DOC Driver 1.0 Block Device (BD)
Software Developer Kit (SDK)
Developer Guide Rev. 0.3

November 2006
96-UM-0806-00
# TABLE OF CONTENTS

**About This Developer Guide**................................................................................................................................. 4

1. **Introduction** ............................................................................................................................................................ 4
   1.1 DOC Driver Block Device (BD) SDK .................................................................................................................. 4
   1.2 DOC Driver BD SDK Architecture ................................................................................................................... 4

2. **DOC Driver Overview** ............................................................................................................................................. 6
   2.1 Interface Terms and Definitions .......................................................................................................................... 6
   2.2 Directory Structure of the DOC Driver SDK ...................................................................................................... 7

3. **Getting Started** ...................................................................................................................................................... 10
   3.1 System Customization ......................................................................................................................................... 10
   3.2 Application Customization .................................................................................................................................. 11

4. **System Customization** ......................................................................................................................................... 12
   4.1 Customizing Compiler and Processor Definitions ............................................................................................ 12
   4.2 Customizing System Services ................................................................................................................................ 20
      4.2.1 FLSYSTEM Module ...................................................................................................................................... 20
      4.2.2 System Functions Description .................................................................................................................... 20
   4.3 Customizing mDOC H3 Memory Register Access Routines ............................................................................. 22
      4.3.1 Basic Terms ................................................................................................................................................. 22
      4.3.2 Overview .................................................................................................................................................... 22
      4.3.3 Run-Time Access Type Customization ....................................................................................................... 22
   4.4 Customizing System Specific HW Performance Accelerators and Features .................................................. 24
      4.4.1 mDOC H3 Synchronous Burst Mode .......................................................................................................... 24
      4.4.2 Interrupts for Flash Erase and Write Operations .......................................................................................... 25
      4.4.3 Overview .................................................................................................................................................... 26
      4.4.4 Terminology ................................................................................................................................................. 26
      4.4.5 SW DMA and HW DMA .............................................................................................................................. 26
      4.4.6 Configuring DOC Driver to Work with SW DMA ....................................................................................... 27
      4.4.7 Configuring DOC Driver to Work with HW DMA ....................................................................................... 27
      4.4.8 DMA Configuration Notes .......................................................................................................................... 30

5. **Application Customization** ................................................................................................................................... 31
   5.1 Customizing the DOC Driver Compilation Flags .............................................................................................. 31
   5.2 Registering the mDOC H3 Device .......................................................................................................................... 38
   5.3 Run-Time Customization and Environment Variables .......................................................................................... 39
      5.3.1 Block Device Environment Variables ........................................................................................................ 39
   5.4 Replacing Malloc/Free........................................................................................................................................... 42
6. DOC Driver Application Programmer’s Interface (API) .........................................................44
  6.1 Introduction ..................................................................................................................44
  6.2 Interface Terms ............................................................................................................45
  6.3 DOC Driver Function Group List ..................................................................................45
  6.4 Block Device Functions ...............................................................................................47
    6.4.1 Volume/Partition Manipulation ..............................................................................47
    6.4.2 Device Formatting .................................................................................................54
    6.4.3 Block Device I/O (Absolute Sector I/O) ...............................................................61
    6.4.4 Using Special Functions .......................................................................................64
  6.5 Writing and Reading the Initial Program Loader (IPL)...................................................65
  6.6 Special DOC Driver Functions ....................................................................................69
    6.6.1 Global DOC Driver Calls ....................................................................................69
    6.6.2 Low Level Information .........................................................................................71
    6.6.3 Internal Device Information .................................................................................72
  6.7 Device Formatting ........................................................................................................75
  6.8 Accessing mDOC H3 Extended Functionality ..............................................................83
    6.8.1 Hardware Read/Write Protection ..........................................................................85
    6.8.2 OTP Operations ...................................................................................................93
    6.8.3 Power Saving Mode .............................................................................................96
    6.8.4 Recovery from a Reset Assertion or Power Off ..................................................97
    6.8.5 HW Configuration API .......................................................................................98

7. DOC Driver Status Codes .................................................................................................100

8. Extended Information .......................................................................................................102
  8.1 Hardware Read/Write Protection .................................................................................102
  8.2 Security-Enabling Features .......................................................................................103
  8.3 Power Modes .............................................................................................................103

9. Additional Information and Tools .....................................................................................107

How to Contact Us ..........................................................................................................108
About This Developer Guide

This guide outlines the steps taken by the developer when working with the DOC Driver Software Development Kit (SDK). It begins with general explanations about DOC Driver SDK and its architecture, followed by an explanation of the package contents and customization options. Finally, it provides detailed descriptions and explanations of the DOC Driver SDK API.

1. INTRODUCTION

1.1 DOC Driver Block Device (BD) SDK

DOC Driver Block Device (BD) SDK is a source code package designed to support the mDOC H3 family of devices. It is API compatible with TrueFFS 7.1, which is the migration version of TrueFFS that supports the mDOC H3 G3, G4 and H1 families of devices as well as the mDOC H3 devices.

The API exported by the SDK enables the user to implement a complete block-device driver. A block-device driver is software that handles sector-based read and write requests by address (sector numbers). Typically, block-device drivers operate in conjunction with an operating system, or with a device-independent file system incorporated into an operating system (such as the FAT file system in the Windows operating systems).

In addition to the basic block-device functionality offered by the SDK, a subset of the API functions is defined as the standard extended-functions interface. These functions enable the driver to provide the OS with the following additional features (described in detail later on):

- Low-level formatting
- General information
- Access to special partitions
- Setting environment variables
- Installation of customized memory access routines
- Special features such as read/write protection, and a One-Time Programming (OTP) area

1.2 DOC Driver BD SDK Architecture

The SDK consists of two major layers: Translation layer and H3 Host SDK.

The translation layer translates the DOC Driver API into the internal mDOC H3 Host SDK API. The mDOC H3 Host SDK is designed to communicate with the mDOC H3 device. It translates the calls it gets into mDOC H3 protocol, which is composed of device specific commands, and sends the commands to the device through the physical layer. Access to the physical layer is implemented in the HAL (HW Abstraction Layer).
The Embedded TrueFFS (ETFFS) running within mDOC H3 device includes 4 layers: MTD, TL, Block Device emulation and I/F. These layers provide similar functionality to that of TrueFFS running on the host side (for support of mDOC H3 G3/P3, G4, H1 devices).

---

**Figure 1: DOC Driver Block Device SDK - Basic Block Diagram**
2. DOC DRIVER OVERVIEW

2.1 Interface Terms and Definitions

Table 1 describes the terms used in this manual.

<table>
<thead>
<tr>
<th>Interface Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>The physical location where an mDOC H3 device can reside. The maximum number of sockets is defined by the <code>FL_SOCKETS</code> definition in <code>FLCUSTOM.H</code>.</td>
</tr>
<tr>
<td>Flash Unit or Block</td>
<td>Smallest erasable area of the flash.</td>
</tr>
<tr>
<td>Physical Drive / Physical Device</td>
<td>mDOC H3 device inserted in a socket. The number of physical drives depends on the number of sockets registered by the socket components in the DOC Driver application. Physical drives are numbered from zero, and serve as the four least significant bits (LSB 0-3) of the drive handle (see below).</td>
</tr>
<tr>
<td>Partition / Volume</td>
<td>A partition is part of a physical drive handled as an independent unit. A physical device can contain up to 14 partitions (in addition to IPL and OTP).</td>
</tr>
<tr>
<td>Logical Partition</td>
<td>Part of the partition, typically described by an entry in the Master Boot Record. Like regular hard disk, every partition can contain many Logical Partitions, each one having completely independent file trees.</td>
</tr>
<tr>
<td>Drive Handle</td>
<td>A handle identifying the partition on which an operation should be performed, specified by the <code>irHandle</code> field of the <code>ioRequestPacket</code> structure. This drive handle is composed of the physical drive number (socket number) (LSB 0-3), partition number (LSB 4-7) and in some routines it can also indicate the logical partition number (LSB 8-11).</td>
</tr>
</tbody>
</table>
2.2 Directory Structure of the DOC Driver SDK

DOC Driver SDK is organized according to the directory structures defined in Table 2. To compile a DOC Driver application, add the relevant C file for your application. In addition include the main header file: `BLOCKDEV.H`. The rest of the headers will be included automatically.

<table>
<thead>
<tr>
<th>File or Directory</th>
<th>Description</th>
<th>When Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC\</td>
<td>The SDK source files</td>
<td>Always</td>
</tr>
<tr>
<td>_COMMON.H</td>
<td>Definitions used by the external API</td>
<td>Always</td>
</tr>
<tr>
<td>_DOCHAPI.H</td>
<td>Definitions used by the external API</td>
<td>Always</td>
</tr>
<tr>
<td>BDDEFS.H</td>
<td>Private type definitions for internal use</td>
<td>Always</td>
</tr>
<tr>
<td>BDKEMUL.H</td>
<td>Type definitions for internal use</td>
<td>Always</td>
</tr>
<tr>
<td>BLOCKDEV.H</td>
<td>DOC Driver API header</td>
<td>Always</td>
</tr>
<tr>
<td>DEFS.C</td>
<td>Translation layer implementation</td>
<td>Always</td>
</tr>
<tr>
<td>DEFS.H</td>
<td>Translation layer definitions</td>
<td>Always</td>
</tr>
<tr>
<td>DOC_DRV.C</td>
<td>Translation layer implementation</td>
<td>Always</td>
</tr>
<tr>
<td>DOCBDK.H</td>
<td>Type definitions for internal use</td>
<td>Always</td>
</tr>
<tr>
<td>DOCH_API.C</td>
<td>H3 API routines implementation file</td>
<td>Always</td>
</tr>
<tr>
<td>DOCH_API.H</td>
<td>H3 API declarations and definitions</td>
<td>Always</td>
</tr>
<tr>
<td>DOCH_ATA.C</td>
<td>ATA driver layer implementation file</td>
<td>Always</td>
</tr>
<tr>
<td>DOCH_ATA.H</td>
<td>ATA driver layer declarations and definitions</td>
<td>Always</td>
</tr>
<tr>
<td>DOCH_FUNC.H</td>
<td>H3 API routine declarations, used internally by other files.</td>
<td>Always</td>
</tr>
<tr>
<td>DOCH_SYS.H</td>
<td>Declaration of H3 HAL layer routines</td>
<td>Always</td>
</tr>
<tr>
<td>DOCHSTUB.C</td>
<td>Implementation of routines needed by FAT filter</td>
<td>Always</td>
</tr>
<tr>
<td>DOCHSTUB.H</td>
<td>Declarations for DOCHSTUB.C file</td>
<td>Always</td>
</tr>
<tr>
<td>DOCHTL.C</td>
<td>Translation layer implementation</td>
<td>Always</td>
</tr>
<tr>
<td>DOCHTL.H</td>
<td>Translation layer definitions</td>
<td>Always</td>
</tr>
<tr>
<td>DOCSOC.C</td>
<td>Socket layer module for mDOC H3</td>
<td>Always</td>
</tr>
<tr>
<td>DOCSYS.C</td>
<td>Customizable memory access routines</td>
<td>Only when using runtime access layer customization (FL_NO_USE_FUNC is not defined)</td>
</tr>
<tr>
<td>DOCSYS.H</td>
<td>Header of the customizable memory access routines</td>
<td>Always</td>
</tr>
<tr>
<td>_DOCSYS.H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>File or Directory</td>
<td>Description</td>
<td>When Needed</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DOSFORMT.C</td>
<td>FAT formatter, logical partitions manipulations</td>
<td>For placing FAT file system format on your mDOC H3.</td>
</tr>
<tr>
<td>DOSFORMT.H</td>
<td>Defines FAT formatter structures and data types</td>
<td></td>
</tr>
<tr>
<td>FATFILT.C</td>
<td>Implementation of the FAT filter</td>
<td>If you use the ffCheckBeforeWrite() routine.</td>
</tr>
<tr>
<td>FATFILT.H</td>
<td>Type definitions for the FAT filter</td>
<td></td>
</tr>
<tr>
<td>FLBASE.C</td>
<td>Basic type-handling routines</td>
<td>Always</td>
</tr>
<tr>
<td>FLBASE.H</td>
<td>Basic type definitions</td>
<td>Always</td>
</tr>
<tr>
<td>FLBUFFER.H</td>
<td>Global buffer data structure definition</td>
<td>Always</td>
</tr>
<tr>
<td>FLCHKDFS.H</td>
<td>Validity check for user customization flags (defined in FLCUSTOM.H and FLSYSTEMS.H)</td>
<td>Always</td>
</tr>
<tr>
<td>FLCOMMON.H</td>
<td>Type definitions and macros common to all DOC Driver packages</td>
<td>Always</td>
</tr>
<tr>
<td>FLOCTL.C</td>
<td>Standard extended function (IOCTL) interface for block device package</td>
<td>Example code for a generic interface to DOC Driver extended functions.</td>
</tr>
<tr>
<td>FLOCTL.H</td>
<td>Type definitions for the standard extended function (IOCTL) interface for block device package</td>
<td></td>
</tr>
<tr>
<td>FLMALLOC.C</td>
<td>Implementation of dynamic allocation routine</td>
<td>In case your system does not support dynamic allocation</td>
</tr>
<tr>
<td>FLMALLOC.H</td>
<td>Forward declaration of the dynamic allocation routines</td>
<td></td>
</tr>
<tr>
<td>FLSTDCMP.H</td>
<td>Forward declaration of the registration routines</td>
<td>Always</td>
</tr>
<tr>
<td>FLSTRUCT.H</td>
<td>Structures common to blockdev interface and IOCTL interface calls</td>
<td>Always</td>
</tr>
<tr>
<td>FLSYSFUN.H</td>
<td>Function prototypes for system-specific functions</td>
<td>Always</td>
</tr>
<tr>
<td>FLSYSTYP.H</td>
<td>Basic types used by the DOC Driver customizable system routines. This file can be included without including the rest of the DOC Driver files in case you need to implement one of the routines in FLSYSTEM.H like the DMA or dynamic allocation routines. An example of such an implementation can be found in FLMALLOC.C and FLMALLOC.H files.</td>
<td>Always</td>
</tr>
<tr>
<td>FLTL.H _FLTL.H</td>
<td>Translation layer definitions</td>
<td>Always</td>
</tr>
<tr>
<td>GEOMETRY.C</td>
<td>Calculation of the C/H/S per given partition capacity.</td>
<td>Always</td>
</tr>
<tr>
<td>HAL_NOR.C</td>
<td>HAL routines implementation for the NOR interface.</td>
<td>When using NOR access layer (DOCH_USE_FUNC is defined)</td>
</tr>
<tr>
<td>HAL_NOR.H</td>
<td>Declarations and definitions for HAL_NOR.C</td>
<td>When using NOR access layer.</td>
</tr>
<tr>
<td>File or Directory</td>
<td>Description</td>
<td>When Needed</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>HIB.H</td>
<td>Definitions of HIB registers</td>
<td>Always</td>
</tr>
<tr>
<td>PART_INF.H</td>
<td>Internal structures for specific partitions information.</td>
<td>Always</td>
</tr>
<tr>
<td>TFFS_API.C</td>
<td>Implementations file of the translation layer between DOC Driver API and H3 Host SDK API</td>
<td>Always</td>
</tr>
<tr>
<td>TFFS_API.H</td>
<td>Definitions and declarations for TFFS_API.C</td>
<td>Always</td>
</tr>
<tr>
<td>SIM_AL\</td>
<td>H3 Simulator source files</td>
<td>Always</td>
</tr>
<tr>
<td>ATA_CMDS.C</td>
<td>ATA Commands simulation</td>
<td>When using H3 simulation</td>
</tr>
<tr>
<td>ATA_CMDS.H</td>
<td>ATA Commands simulation definitions</td>
<td>When using H3 simulation</td>
</tr>
<tr>
<td>HAL_SIM.C</td>
<td>H3 simulator access layer implementation</td>
<td>When using H3 simulation</td>
</tr>
<tr>
<td>HAL_SIM.H</td>
<td>H3 simulator access layer definitions</td>
<td>When using H3 simulation</td>
</tr>
<tr>
<td>SIM_DEV.C</td>
<td>H3 simulation implementation file</td>
<td>When using H3 simulation</td>
</tr>
<tr>
<td>SIM_DEV.H</td>
<td>Declarations and definitions for SIM_DEV.C</td>
<td>When using H3 simulation</td>
</tr>
<tr>
<td>CUSTOM\</td>
<td>Reference code for DOC Driver application customization.</td>
<td>Always</td>
</tr>
<tr>
<td>FLCUSTOM.C</td>
<td>mDOC H3 registration routines.</td>
<td>Always</td>
</tr>
<tr>
<td>FLCUSTOM.H</td>
<td>DOC Driver feature customization.</td>
<td>Always</td>
</tr>
<tr>
<td>SYSTEMS\</td>
<td>System-specific DOC Driver customization code.</td>
<td>Always</td>
</tr>
<tr>
<td>SYSTEM\VC60\</td>
<td>Code-specific reference to PC targets, compiled using Visual Studios 6.0.</td>
<td>Always</td>
</tr>
<tr>
<td>FLSYSTEM.C</td>
<td>System-specific routines needed by DOC Driver (reference code). Refer to section 4.</td>
<td>Always</td>
</tr>
<tr>
<td>FLSYSTEM.H</td>
<td>Customization definitions specific to the host system, such as compiler, CPU, OS (reference code). Refer to section 4.</td>
<td>Always</td>
</tr>
<tr>
<td>MapMemAPI.h</td>
<td>Interface to Windows XP driver for msystems Evaluation Board (EVB)</td>
<td>Always</td>
</tr>
<tr>
<td>MapMemory.dll</td>
<td>Windows XP driver for msystems Evaluation Board (EVB)</td>
<td>Always</td>
</tr>
<tr>
<td>Install shield\Setup.exe</td>
<td>Installation wizard for Windows XP driver for msystems Evaluation Board (EVB)</td>
<td>Always</td>
</tr>
<tr>
<td>Examples\</td>
<td>Sample projects for Visual Studio 6.0</td>
<td>Always</td>
</tr>
<tr>
<td>LOWLEVEL</td>
<td>Examples for writing applications using DOC Driver BD SDK</td>
<td>Always</td>
</tr>
<tr>
<td>LOWLEVEL.C</td>
<td>Sample use of block-device interface functions. This application mounts a volume, writes virtual sectors, reads them back and verifies their content on the partition.</td>
<td>Always</td>
</tr>
<tr>
<td>FLCUSTOM.C</td>
<td>DOC Driver registration routines. For the LOWLEVEL example</td>
<td>Always</td>
</tr>
<tr>
<td>FLCUSTOM.H</td>
<td>DOC Driver feature customization. For the LOWLEVEL example</td>
<td>Always</td>
</tr>
<tr>
<td>VC6_DiskOnChip\</td>
<td>LOWLEVEL example project.</td>
<td>Always</td>
</tr>
<tr>
<td>LOWLEVEL.DSP</td>
<td>LOWLEVEL example workspace.</td>
<td>Always</td>
</tr>
<tr>
<td>LOWLEVEL.DSW</td>
<td>LOWLEVEL example workspace.</td>
<td>Always</td>
</tr>
</tbody>
</table>
3. **Getting Started**

The DOC Driver SDK is supplied as a source code package that can be compiled by any ANSI-C or C++ compiler. The DOC Driver SDK is written in a form that can be ported to any target, and easily configured and customized.

Many features and properties of the DOC Driver software can be configured in compile-time customization options. These options make DOC Driver very flexible, both for meeting application needs and ensuring that the resulting code includes only the options required, minimizing the impact on RAM, ROM and performance.

Customization is performed using C definitions and functions in several customization source-code files, which are then compiled with the application.

DOC Driver customization is divided into two major steps, described in detail in the following subsections:

- **System customization** - Consists of customizing the DOC Driver SDK for your computer platform, independent of the specific DOC Driver application.

- **Application customization** - Adapts the needs of your DOC Driver block device or file system application. Specify this customization in the source files `FLCUSTOM.C`, `FLCUSTOM.H`.

### 3.1 System Customization

System customization is performed by:

- Customizing compiler and processor definitions.

- Selecting the appropriate system related customization options for compiling the DOC Driver code and for generating the code that will run on the processor. Specify this customization in the source file `FLSYSTEM.H`.

- Customizing system routines.

  - Provide the DOC Driver SDK package with services for the particular computer system and OS: Current date/time and other timing functions, memory management, and task synchronization (on systems where these issues are relevant). Code the needed routines in the `FLSYSTEM.C` source file.

- Customizing mDOC H3 memory (registers) access routines.

  - Provide access to the mDOC H3 memory window for specific environment. The code and declarations of these functions can be found in `DOCSYS.C` and `DOCSYS.H`. Either use the variety of access routines supplied by DOC Driver can be used or platform-specific code. Both options are available at runtime using the environment variables mechanism.

- Customize system specific HW performance accelerators.
Some systems may provide HW mechanisms for accelerating the mDOC H3 access time such as DMA and Burst modes. Interrupts can also be used to improve the system responsiveness. These HW mechanisms can be utilized by DOC Driver using a set of customizable macros. Specify this customization in the source file `FLSYSTEM.H`.

This topic is discussed in greater detail in Section 4.

### 3.2 Application Customization

Perform application customization by:

- Customizing DOC Driver services requirements.
- Specify the limits and features that the application requires. For example, define the maximum number of simultaneously mounted disk partitions or decide whether to compile the support of the HW protection features. Customization is done by setting the various DOC Driver definitions found in the `FLCUSTOM.H` code reference files.

- Registering supported components.
- The Registration consists of coding the `flRegisterDochParams()` routine in `FLCUSTOM.C`.

- Declaring and setting of different modes of operation through the environment variables mechanism.

- When using the environment variables mechanism the following global variable must also be declared: Unsigned char `flUse8Bit` and code the `flSetEnvVar()` routine to set its default value. An example of such a routine can be found in the `FLCUSTOM.C` example file.

This topic is discussed in greater detail in Section 5.
4. SYSTEM CUSTOMIZATION

4.1 Customizing Compiler and Processor Definitions

To customize the DOC Driver code to match your target processor, you need:

- An ANSI-C or C++ compiler that can generate code for your target processor.
- To customize the options in **FLSYSTEM.H** to produce code that compiles and executes correctly.
- Some of the customization options in **FLSYSTEM.H** require you to code your own platform specific implementation. In most of these cases DOC Driver will use a dedicated macro instead of a routine, and you will need to define it in **FLSYSTEM.H**. The actual implementation can be coded in either **FLSYSTEM.C**, any other C file or even in one of your standard libraries. You could also use **FLSYSTEM.H** for forward declarations, as this file is included by all other DOC Driver files.
- In case a DOC Driver specific definition, such as **FLByte** or **FLWord**, is required for the customized routine declaration, **FLSYSTYP.H** should be included (See DOC Driver malloc implementation in **FLMALLOC.C** and **FLMALLOC.H** for examples).

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mDOC H3 memory window start address.</td>
<td>The macro <strong>FL_DOC_ADDRESS</strong> defined in <strong>FLSYSTEM.H</strong> should be configured with the exact start address of mDOC H3 memory window, as it must be supplied when registering the mDOC H3 device.</td>
</tr>
<tr>
<td>NULL constant</td>
<td>The <strong>NULL</strong> pointer is usually defined simply as 0 (zero). Some compilers do not accept this and require an alternative definition such as *(void <em>) 0</em>. Ensure that <strong>FLSYSTEM.H</strong> contains the appropriate definition.</td>
</tr>
<tr>
<td>Little-Endian/Big-Endian</td>
<td>Some structures use the Little-Endian (Intel) format for integers, in which low-order digits are in lower addresses. If your processor uses the Big-Endian (Motorola) format, in which the high-order digits are in the lower addresses, uncomment the definition of <strong>FL_BIG_ENDIAN</strong>. The memory access layer is also affected by the Endian architecture, since the compiler handles variable casting differently.</td>
</tr>
<tr>
<td>Byte swap support</td>
<td>You can force the mDOC H3 HW to swap it's lower and upper 8 bits data bus, using the <strong>FL_SWAP_BYTES</strong> compilation flag. This feature is convenient for instance where the mDOC H3 was not properly connected to the host CPU.</td>
</tr>
</tbody>
</table>
## Memory Manipulation

Macros

The library routines `memcpy`, `memset`, and `memcmp` are standard for ANSI-C compilers. However, there may be variations in which the `#include` directive is needed in each file in which these routines are used, and variations such as Borland’s `_fmemcpy`, `_fmemset`, and `_fmemcmp` routines also exist.

To overcome such nuances, the DOC Driver code uses memory-handling macros called `tffscpy`, `tffsset`, and `tffscmp`, with prototypes identical to the standard `memcpy`, `memset` and `memcmp`.

## Pointer Arithmetic

You are required to code the following pointer-arithmetic macros (or short routines) for your processor or compiler:

* **physicalToPointer** *(address, length, driveNo)*

  This macro/routine will return a pointer to the physical address specified by the "address" parameter.

  In a real-mode Intel 80x86 architecture, your implementation should convert a linear address (such as 0xD8000) to a segment:offset far pointer (such as 0xD800:0).

  In a system that uses virtual addressing, this macro would be more complex: It would have the task of mapping the physical address to virtual address space, and would have to know the correspondence between physical and virtual addresses in order to return a pointer. For this reason, the second parameter “length” is supplied, showing the amount of space starting at the physical address space that should be mapped to the virtual address space.

* **addToFarPointer** *(pointer, increment)*

  This macro defines how to add an offset to a pointer and return a new pointer.

  In most cases this can be coded simply as `((char *) (pointer)) + (increment)`, but sometimes this may be inappropriate. For example, when you memory resides on different non consecutive banks.

* **pointerToPhysical** *(ptr)*

  This macro does the opposite of the previous macro `physicalToPointer`.

* **freePointer** *(ptr, size)*

  On some virtual-memory systems, the virtual pointer to the mDOC H3 physical memory must be freed before DOC Driver exits. If your platform uses a flat memory model, simply code this macro as 1, or as an empty macro i.e. do nothing.

Note that this macro is different from the `FL_FREE` macro used for dynamic memory allocation.
### Item Description

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed / Unsigned Char</td>
<td>The DOC Driver code assumes the default for type char is signed (default for most C compilers). If the default for your compiler is signed, further changes are not required. Otherwise, override the default by using compilation switches or parameters.</td>
</tr>
</tbody>
</table>
| Mutex Data Type       | A mutex is a type of semaphore that manages access to a non-sharable resource by enforcing mutual exclusion between agents that want to “own” it.  
On multithreaded or multitasking systems, DOC Driver can handle concurrent requests as long as each request targets a different physical drive. However, calls to the same physical drive must be handled serially, and access to the physical drive is managed by a mutex.  
A quick and easy way of implementing a mutex is by using an integer that “counts” zero when the resource is free and non-zero otherwise.  
The reference code for the `FLSYSTEM.H` and `FLSYSTEM.C` files includes such a basic mutex implementation. If your system does not have multiple tasks working with DOC Driver, or if the basic mutex implementation suits your needs, use the definitions provided in the reference code.  
If you intend to have multiple tasks accessing DOC Driver and the native methods of either your Operating System or your own mutual exclusion are available, define the mutex data type as a typedef for `FLMutex` in `FLSYSTEM.H`. In addition, you must provide implementations for the various mutex functions in `FLSYSTEM.C`. (Refer to Section 4.2 for the specific functions.) |
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
</table>
| Memory Allocation | DOC Driver uses a number of tables and buffers. These tables and buffers can be statically allocated during compilation or dynamically allocated using a heap manager. Dynamic allocation is typically more efficient in terms of RAM usage, since memory is allocated only when required. Alternatively, dynamic allocation requires the use of several runtime library routines, such as malloc, which adds to the code size. If you use `malloc` and `free`, the standard C routines for memory allocation, define:  
```c
#define FL_MALLOC malloc
#define FL_FREE free
```
You must also add the correct `#include` directive for these functions (usually `<malloc.h>` or `<stdlib.h>`). At this point, define any other routines for memory allocation, provided they have the same function prototypes as `malloc` and `free`. In case your system does not contain an implementation of dynamic memory allocator you can use DOC Driver static heap implementation. For more details see section 5.4. Notes: For backwards compatibility reasons, if `MALLOC` is defined while `FL_MALLOC` is not, `FL_MALLOC` receives the same definition as `MALLOC`, therefore dynamic allocation is still used. If you do not define the `FL_FAR_MALLOC` macro, DOC Driver will use the `FL_MALLOC` definition. |
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug Prints</td>
<td>DOC Driver uses several macros for reporting debug information, such as errors and progress. Customize the following six macros according to the level of debug information your application requires:</td>
</tr>
<tr>
<td></td>
<td>The macros are divided into 2 sets:</td>
</tr>
<tr>
<td></td>
<td>Simple string print macros, such as <code>cputs</code></td>
</tr>
<tr>
<td></td>
<td>- <code>DBG_PRINT_ERR(zone, string)</code> – Prints errors</td>
</tr>
<tr>
<td></td>
<td>- <code>DBG_PRINT_FLOW(zone, string)</code> – Prints progress</td>
</tr>
<tr>
<td></td>
<td>- <code>DBG_PRINT_WRN(zone, string)</code> – Prints important information</td>
</tr>
<tr>
<td></td>
<td>Print macros with variable type of parameters, such as <code>printf</code></td>
</tr>
<tr>
<td></td>
<td>- <code>DBG_PRINT_ERR_PRM(zone,(string))</code> – Prints errors</td>
</tr>
<tr>
<td></td>
<td>- <code>DBG_PRINT_FLOW_PRM(zone,(string))</code> – Prints progress</td>
</tr>
<tr>
<td></td>
<td>- <code>DBG_PRINT_WRN_PRM(zone,(string))</code> – Prints important information</td>
</tr>
<tr>
<td></td>
<td>The zone parameter is supplied by DOC Driver to indicate the module using the macro. You can use zone parameter to customize the print level at runtime or to focus your debugging. The list of debug zones is defined in <code>FLYSTYP.H</code>.</td>
</tr>
<tr>
<td></td>
<td>You can customize all these macros as <code>printf</code> for full debug information, as a conditioned <code>printf</code> or not at all for a released binary.</td>
</tr>
<tr>
<td></td>
<td>For customization convenience, the strings passed to <code>DBG_PRINTF_XXX_PRM</code> routines were enclosed with the <code>FLTXT</code> macro, in case ASCII to Unicode conversion is required. This macro will be ignored if it was not customized.</td>
</tr>
<tr>
<td>Debug Prints from ETFFS</td>
<td>To support debug prints from ETFFS the following macro should be customized: <code>DBG_PRINT_ATA(zone, (FLByte *))</code></td>
</tr>
<tr>
<td></td>
<td>In order to enable the print-outs the environment variable <code>FL_ENV_ATA_DEBUG</code> must be set to <code>FL_ON</code> (see Section 5.3.1).</td>
</tr>
<tr>
<td>Extended Functions First</td>
<td>When using DOC Driver IOCTL functions (extended functions) you must define the code of the first DOC Driver block-device and file system IOCTL functions using the <code>FL_IOCTL_START</code> definition. After that, the functions get consecutive increasing numbers.</td>
</tr>
<tr>
<td>Function Number</td>
<td>This number should be outside the range of the standard IOCTL codes used by your operating system.</td>
</tr>
<tr>
<td>Default Calling Convention</td>
<td>C compilers usually use the C calling convention to routines (cdecl), but often can also use the Pascal calling convention, which is somewhat more economical in code size. Some compilers also have specialized calling conventions, which may be suitable. Use compiler switches or insert a <code>#pragma</code> here to select your preferred convention.</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>Non-Default Calling Convention</td>
<td>The <code>NAMING_CONVENTION</code> definition is used when the calling application uses a different convention than the one used to compile DOC Driver. The <code>NAMING_CONVENTION</code> definition is added as a qualifier to all DOC Driver exported API. A good example for its use is a C++ application that uses DOC Driver, which was compiled using standard C. If all of your code is compiled with the same convention, simply leave the <code>NAMING_CONVENTION</code> defined as an empty macro.</td>
</tr>
<tr>
<td>Release all system resources</td>
<td>Upon exit, DOC Driver calls the <code>FL_SYS_FUNC_RELEASE</code> macro. This macro is not used in most systems.</td>
</tr>
</tbody>
</table>
| Yielding the CPU | DOC Driver utilizes the `FL_TFFS_SLEEP(handle,Microseconds)` macro to yield the CPU while waiting for time-consuming operations like a flash erase. *Handle* argument describes the socket and disk partition numbers (0-3 LSB describe the socket number while bits 4-7 describe the disk partition number). *Microseconds* argument indicates the recommended time to yield the CPU. Notes:  
  - You can also use this routine for your interrupt handler as it is called before polling the ready busy signal. More info in the HW acceleration section.  
  - In case of error, this macro will be called with microseconds equal to zero and interrupt no longer be asserted. |
| Removing backward-compatibility basic type definitions | DOC Driver defines the byte, word, dword, Sbyte, Sword and Sdword basic types. Since these names are used by several major OSs, such as Windows and VxWorks, it uses an alternative set of type definitions with less general names (See the FLxxx types below). As many DOC Driver-based applications already use these types, DOC Driver still automatically defines them. If these definitions cause your compiler to report redefinition errors, simply define `FL_DISABLE_OLD_TRUEFFS_TYPES` and DOC Driver will automatically remove the definition. |
| Basic variable types used by DOC Driver. The default of these types can be found in `FLCOMMON.H` | |
## Access Layer Customization

### Using Macros to Access mDOC H3

By default, DOC Driver uses routines to access mDOC H3 memory registers. For improved performance, you can customize your own access macros by un-commenting the `FL_NO_USE_FUNC`.

The macros are as follows:

- `flDirectWrite8BitReg(flash, offset, val)`
- `flDirectWrite16BitReg(flash, offset, val)`
- `flDirectRead8BitReg(flash, offset)`
- `flDirectRead16BitReg(flash, offset)`
- `flReadEvenNumberOfBytes(flash, offset, dest, count)`
- `flWriteEvenNumberOfBytes(flash, offset, src, count)`
- `flRead512Bytes(flash, offset, dest)`
- `flWrite512Bytes(flash, offset, src)`
- `flDocWindow(flash)`

### Note:

- SW DMA can be implemented by customizing the `flReadEvenNumberOfBytes` and `flWriteEvenNumberOfBytes` macros.
- A default implementation for 16 bit platforms is available.

### Platforms that map the mDOC H3 to I/O ports may require supplying the DOC Driver access layer with macros for accessing the I/O space.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FLWRITE_IO_BYTE</strong></td>
<td>Write 8 bits to mDOC H3</td>
</tr>
<tr>
<td><strong>FLWRITE_IO_WORD</strong></td>
<td>Write 16 bits to mDOC H3</td>
</tr>
<tr>
<td><strong>FLWRITE_IO_DWORD</strong></td>
<td>Write 32 bits to mDOC H3</td>
</tr>
<tr>
<td><strong>FLREAD_IO_BYTE</strong></td>
<td>Read 8 bits from mDOC H3</td>
</tr>
<tr>
<td><strong>FLREAD_IO_WORD</strong></td>
<td>Read 16 bits from mDOC H3</td>
</tr>
<tr>
<td><strong>FLREAD_IO_DWORD</strong></td>
<td>Read 32 bits from mDOC H3</td>
</tr>
<tr>
<td><strong>TFFSCPY_FROM_IO_8_BITS</strong></td>
<td>Read from IO using 8-bit operands</td>
</tr>
<tr>
<td><strong>TFFSCPY_FROM_IO_16_BITS</strong></td>
<td>Read from IO using 16-bit operands</td>
</tr>
<tr>
<td><strong>TFFSCPY_TO_IO_8_BITS</strong></td>
<td>Write to IO using 8-bit operands</td>
</tr>
<tr>
<td><strong>TFFSCPY_TO_IO_16_BITS</strong></td>
<td>Write to IO using 16-bit operands</td>
</tr>
<tr>
<td><strong>TFFSSET_IO_8_BITS</strong></td>
<td>Set IO using 8-bit operands</td>
</tr>
<tr>
<td><strong>TFFSSET_IO_16_BITS</strong></td>
<td>Set IO using 16-bit operands</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Enable burst mode     | mDOC H3 support burst mode (synchronous access) in read access only. To enable read burst interface mode from the mDOC H3 side make sure FL_USE_BURST_MODE_READ is defined. Note that burst mode usually has some parameters, which should be properly customized using the following definitions.  
  #define BURST_LENGTH XXX /* Number between 0-10 */  
  #define BURST_LATENCY XXX /* Number between 0-7 */  
  #define BURST_HOLD XXX /* FL_OFF/FL_ON */  
  #define BURST_CLK_INV XXX /* FL_OFF/FL_ON */  
  #define BURST_WAIT_STATE XXX  
  DOC Driver will inform the system of the beginning and end of every burst transfer through the use of 2 customizable macros:  
  FL_HOST_ENTER_BURST_MODE() and  
  FL_HOST_EXIT_BURST_MODE() |
| HW DMA                | To utilize the HW DMA you will need to customize the FL_DMA_CONFIG(bDMA_ParamsPtr) macro. More details in section 4.4. |
4.2 Customizing System Services

4.2.1 FLSYSTEM Module

DOC Driver makes use of several functions that may be implemented differently in each system. Most of these functions relate to system time (clock, date, etc.), mutex and critical-section handling.

These functions should be implemented in **FLSYSTEM.C**. Note that these functions are prototyped in **FLSYSFUN.H** and not in **FLSYSTEM.H**, which contains system-dependent definitions and macros.

Some system specific routines were actually added as macros in **FLSYSTEM.H**. However those macros can be implemented as a prototype for a more complex implementation in **FLSYSTEM.C**.

The implementation of the **FLSYSTEM.C** functions is tied to the target system and cannot be provided for by DOC Driver. The actual implementation is usually a simple matter for someone familiar with the target systems. A full-featured DOC Driver system requires correct implementation of all of the functions declared in **FLSYSFUN.H**. However, the following considerations may shorten and simplify the customization effort.

Several system functions can actually be implemented in a system-independent manner, unless some special considerations apply to the target system. When special considerations do not apply, the **FLSYSTEM.C** reference code supplied with the DOC Driver system can be used as is. For example, the mutex-related functions are coded in the reference example and can be used on almost every system.

Most system functions are required only when using an optional DOC Driver feature or component, and are not required in the context of all applications. For fast prototyping, many functions can be marginally implemented without significantly affecting overall functionality.

4.2.2 System Functions Description

System functions are the system services defined by **FLSYSFUN.H** and implemented in **FLSYSTEM.C**, as defined in Table 4.
### Table 4: System Function Definitions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>flSysFunInit</strong>()</td>
<td><code>flSysFunInit() does not provide service to DOC Driver. It enables you to perform necessary initializations for other system functions, such as initializing the time and date for the </code>flCurrentTime()<code>and</code>flCurrentDate()<code>routines. The routine is called only once, during DOC Driver initialization, from the</code>flInit()` routine found in ABST_LYR.C.**</td>
</tr>
<tr>
<td><code>extern void flSysfunInit (void);</code></td>
<td></td>
</tr>
</tbody>
</table>
| **flCreateMutex**() | Creates and initializes a mutex object. Once initialized, the mutex is not owned by anyone. The `flTakeMutex()` function takes hold of the mutex preventing others from retaking it. **

`extern FLStatus flCreateMutex (FLMutex mutex);` **
/* Pointer to mutex object */**

Returns '0' on success, otherwise failure. |
| **flDeleteMutex**() | Destroys a mutex object. This function frees any resources taken by `flCreateMutex()`. **

`extern void flDeleteMutex (FLMutex mutex);` **
/* Pointer to mutex object */**

| **flTakeMutex**() | Attempts to take ownership of a mutex. If the mutex is currently not owned, TRUE is returned and the mutex becomes owned. If the mutex is currently owned, FALSE is returned and ownership is not taken. **

`extern Boolean flTakeMutex (FLMutex mutex);` **
/* Pointer to mutex object */**

Returns TRUE if ownership taken, FALSE otherwise. |
| **flFreeMutex**() | Frees ownership of a mutex. **

`extern void flFreeMutex (FLMutex mutex);` **
/* Pointer to mutex object */**

| **flCurrentTime**() | Returns the current DOS-format time. **

`extern unsigned flCurrentTime(void);` |
| **flCurrentDate**() | Returns the current DOS-format date. **

`extern unsigned flCurrentDate(void);` |
4.3 Customizing mDOC H3 Memory Register Access Routines

4.3.1 Basic Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>mDOC H3 Memory Window</td>
<td>mDOC H3 is a memory-mapped device that is mapped into a window of at least 8KB in the CPU address space.</td>
</tr>
<tr>
<td>Memory Window Base Address</td>
<td>The address where the mDOC H3 memory window starts.</td>
</tr>
<tr>
<td>Minimal Bus Width</td>
<td>The minimum number of bits that can be accessed in a single software call. mDOC H3 minimum bus width is 16bit.</td>
</tr>
</tbody>
</table>

4.3.2 Overview

mDOC H3 devices can be connected to systems with various bus widths and configurations. For this purpose, implementation of DOC Driver memory access routines resides in a special module - `DOCSYS.C` and `DOCSYS.H`. DOC Driver includes several ready-made memory access implementations for common bus configurations related to the mDOC H3 bus width.

Some buses are restricted to minimum data width, which may be greater than the mDOC H3 bus width. To solve this, the mDOC H3 address lines should be connected to the system address lines with a suitable shift. In addition, DOC Driver should be configured to compensate for this shift.

mDOC H3 access configurations can be customized in one of two ways:

1. **Runtime (recommended):** More flexible and easier to customize. Taking this approach you may:
   - Specify your platform limitations and let DOC Driver install the most suitable access routines from the DOC Driver (already customized) library of routines.
   - Instruct the DOC Driver to call your own custom-made routines.

2. **Compile time:** Better performance and smaller code size.

The required method is selected using the `FL_NO_USE_FUNC` compilation flag in `FLSYSTEM.H`.

Please note that even when using DOC Driver runtime customization model the access layer implementation in `DOCSYS.C` will eventually call the `FLREAD_IO_xxx` and `FLWRITE_IO_xxx` macros coded in `FLSYSTEM.H`.

4.3.3 Run-Time Access Type Customization

To use the runtime customization of the DOC Driver access routine, make sure that the `FL_NO_USE_FUNC` definition is commented out in the `FLSYSTEM.H` customization file.

The runtime access mechanism provides an easier and safer customization. However, this mechanism usually results in a larger code.
4.3.3.1 mDOC H3 Run-Time Access Type Customization

Using DOC Driver Built-In Routines

DOC Driver uses the `flBusConfig` global array to automatically select the best access routine for your target platform. The `flBusConfig` array consists of the following flags:

- Minimum number of bits that can be accessed by the bus to mDOC H3’s memory window in a single fetch instruction.
  
  ```
  FL_BUS_HAS_16BIT_ACCESS /* Bus can access 16-bit */
  FL_BUS_HAS_32BIT_ACCESS /* Bus can access 32-bit */
  ```

- Number of address shifts performed on the mDOC H3 address pins.
  
  ```
  FL_NO_ADDR_SHIFT /* No address shift */
  FL_SINGLE_ADDR_SHIFT /* Single address shift */
  ```

The `flBusConfig` default value is controlled by the `FL_DEFAULT_BUS_CONFIGURATION` definition in `FLSYSTEM.H`. If this value is not defined, DOC Driver will use the following value:

```
FL_BUS_HAS_16BIT_ACCESS | FL_BUS_HAS_32BIT_ACCESS | FL_NO_ADDR_SHIFT
```

This means that DOC Driver is configured by default for a platform that has no address shift, and supports both 16-bit and 32-bit access.

Note: For improved flexibility, the `flBusConfig` variable is actually an array of such variables. Each cell of the `flBusConfig` array controls its corresponding socket.

Changing the default `flBusConfig` value is necessary when your target platform does not support one of the access types (16- and 32-bit), or requires an address shifting. To do so, make sure `FL_ENVIRONMENT_VARS` is defined (in `FLCUSTOM.H`) and call either of the following routines:

- To choose a different configuration (`flBusConfig` value) for a specific socket call:
  
  ```
  FLStatus NAMING_CONVENTION flSetEnvSocket(FLEnvVars variableType, FLByte socket, FLDword value, FLDword *prevValue)
  ```

- To set a different configuration for all the sockets in the system:
  
  ```
  FLStatus NAMING_CONVENTION flSetEnvAll(FLEnvVars variableType , FLDword value, FLDword *prevValue)
  ```

Both routines use the following arguments:

- `variableType` = Always `FL_MTD_BUS_ACCESS_TYPE`.
- `value` = New `flBusConfig` value.
- `prevValue` = Returns the previous `flBusConfig` value.
4.4 Customizing System Specific HW Performance Accelerators and Features

mDOC H3 devices support interfaces to HW acceleration mechanisms such as DMA, burst mode, asynchronous access and interrupts. This section reviews the HW accelerators and explains how they are to be controlled by utilizing the DOC Driver.

4.4.1 mDOC H3 Synchronous Burst Mode

mDOC H3 devices support switching into synchronous bus interface while transferring bursts of data from the device to the host (Read Access), allowing mDOC H3 to output data on every clock.

This mode can be used to improve performance since cycle time is reduced compared to asynchronous mode. Setting up the burst usually takes a bit longer than a normal bus cycle, but once it was initialized the rest of the cycles in that burst are much faster. This type of data transfer is not used for regular mDOC H3 register access, but can be very useful when reading the flash page data which is 0.5K-2K consecutive bytes.

In order to support this mode DOC Driver will call `FL_HOST_ENTER_BURST_MODE (Socket)` (user defined macro) before reading bursts of data. This macro will switch mDOC H3 bus interface to be synchronous interface.

After the transaction ends DOC Driver will call `FL_HOST_EXIT_BURST_MODE (Socket)` (user defined macro) in order to return mDOC H3 to asynchronous mode.

Enabling the read burst support in mDOC H3 transfers requires some mDOC H3 HW support, and is enabled using the `FL_USE_BURST_MODE_READ` definition (`FLSYSTEM.H`)

The user should define the following burst protocol parameters (`FLSYSTEM.H`):

```
#define BURST_HOLD FL_ON/FL_OFF
  • FL_OFF: Data is held for one clock cycle.
  • FL_ON: Data is held for two clock cycles (used for fast clocks).

#define BURST_LENGTH integer number between 2 – 5
  • Number of 16 bits used per burst cycle:
    PlatformBurstLength equals to (2^BURST_LENGTH) - number of bytes/words in burst cycle.

#define BURST_LATENCY integer number between 0 – 7
  • This is the number of clock cycles between the time when the mDOC H3 samples CE# asserted and when the first word of data is output by the mDOC H3 device. This number of clock cycles is equal to BURST_LATENCY + 2. Valid values are integers between 0-7

#define BURST_WAIT_STATE integer number between 0 – 3
```
The number of clocks from the one before last clock of the burst until the assertion of CS. Should be between 0 and 3 (included).

4.4.2 Interrupts for Flash Erase and Write Operations

After passing user data to the mDOC H3, DOC Driver polls the mDOC H3 controller until the write operation is completed (default). Users may want to use this time to perform other tasks. DOC Driver allows users to customize the `FL_TFFS_SLEEP (Handle, TimeInMicroSec)` macro that is called while mDOC H3 starts writing and can be used to carry out other tasks while flash is busy writing. Sleep time is given by `TimeInMicroSec` (time in micro seconds).

In addition DOC Driver may configure the mDOC H3 device to assert an interrupt signal indicating the completion of the flash operation. Please refer to section 6.8.5 for further details.

Notes:
1. Before calling `flHwConfig` the initial mode can be determined by the `FL_IRQ_RB_INIT(wSockeNo)`.
2. In order to properly use this feature, you will be required to code your own interrupt service routine that will cause the `FL_TFFS_SLEEP (handle, time)` to return to the DOC Driver code.
DMA Support

4.4.3 Overview

Working with mDOC H3 requires the transaction of data between mDOC H3 device and host memory. This task can either be carried out by the CPU or via the DMA (Direct Memory Access) Controller. This unique HW module is dedicated to the transfer of data between peripheral to RAM without requiring CPU intervention.

The advantages of using DMA are:

- **Release of CPU resources** - DMA controller offload data transfers from CPU to dedicated HW, thus freeing the CPU to perform other tasks.

- **Performance boost** - DMA controller can make the transaction faster than SW (Since the DMA controller is a HW state machine implementation designed especially for this task)

- **Reduced power consumption** – CPU core can run at reduced clock or even remain in sleep mode until the transaction is completed.

4.4.4 Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA physical channel</td>
<td>Transfers are made through a physical channel that can be thought of as a pipe that connects a source and a destination for the duration of a transfer. Data flows through this pipe from the source to the destination.</td>
</tr>
<tr>
<td>Transfer descriptor</td>
<td>Each channel is controlled by a set of configuration registers, where the software sets up the transfer parameters such as length, source and destination addresses. This set of registers is called the transfer descriptor.</td>
</tr>
<tr>
<td>Element</td>
<td>The smallest level of data that can be sent in a channel (Byte/Word/DWord/Burst of X bytes)</td>
</tr>
<tr>
<td>Frame</td>
<td>A frame is a set of N elements.</td>
</tr>
<tr>
<td>Block</td>
<td>A block is defined as X frames which need to be transferred between source and destination in a single transaction.</td>
</tr>
</tbody>
</table>

4.4.5 SW DMA and HW DMA

In SW DMA mode, both the mDOC H3 device and the DOC Driver SW components work as normal (mDOC H3 is not alerted to being in DMA mode), but the host CPU uses the DMA to transfer data to and from the mDOC H3 device using DMA. In this case the DMA moves each time one frame in a channel.

In HW DMA mode, the processor sets up a DMA channel as a synchronized transfer (demand driven by DMA_REQ#). The channel waits for a DMA_REQ# signal from the mDOC H3 device before each frame transaction starts. HW DMA moves blocks rather than one frame each time. At the end of the transaction, the mDOC H3 device signals the host (via the IRQ# pin) that the transaction has completed successfully.
4.4.6 Configuring DOC Driver to Work with SW DMA

Customization is done by replacing the access layer that reads and writes 512 bytes to and from the mDOC H3 device. Macros must be applied for the Read/Write of 512 bytes to and from mDOC H3 and to and from the host to work using DMA channel.

This is done by configuring the following macros in `FLSYSTEM.H`:

```c
#define flWrite512Bytes(flash,offset,src) \n    DMA_SwWriteFrame(flashPtr->win,offset,src)
#define flRead512Bytes(flash,offset,dest) \n    DMA_SwReadFrame(flashPtr->win,offset,dest)
```

In addition the following flag MUST be defined (in `FLSYSTEM.H`) to enable DMA (both HW and SW):

```c
#define FL_USE_DMA_ON_WRITE
```

The functions used in the above macros for DMA utilization should do the following:

1. **DMA_SwReadFrame**:
   - Using DMA transaction move 512 bytes from `flashPtr->win + offset` (mDOC H3 memory address) to `dest` (RAM memory address)
   - Wait until DMA completes the transaction.
2. **DMA_SwWriteFrame**:
   - Using DMA transaction, moves 512 bytes from `src` (RAM memory address) to `flashPtr->win + offset` (mDOC H3 memory address).
   - Waits until DMA completes the transaction.

4.4.7 Configuring DOC Driver to Work with HW DMA

mDOC H3 supports HW DMA feature for both read and write operations.

In order to support the HW DMA mode the `FL_DMA_CONFIG` macro (`FLSYSTEM.H`) must be implemented:

```c
#define FL_DMA_CONFIG(DMA_Params_P) - Where DMA_Params_P is a pointer to variable from type DMA_Params_S struct defined in `FLCOMMON.H` as following:
```

```c
typedef struct
{
```
FLWord wDiskOnChip_Offset;
FLWord wFrameSize;
FLWord wFramesInBlock;
FLWord wFramesXferred;
FLByte FAR0 *bDiskOnChip_BasePtr;
FLByte FAR0 *bDestAddrPtr;
FLByte bOpType;
FLByte blrHandle;
FLBoolean fDmaStatus;
void *DrvParamsPtr;
}
} DMA_Params_S;

**FL_DMA_CONFIG** macro is called by DOC Driver with the following types, defined by the value of **bOpType** field from **DMA_Params_S** structure:

**bOpType = DMA_OPEN_CHANNEL:**
1. This call is done once when initializing DOC Driver
2. In this case SW should allocate a DMA channel between the CPU and the mDOC H3
3. **fDmaStatus** field should be set to TRUE, when failures occurred; or to FALSE otherwise.

**bOpType = DMA_FREE_CHANNEL:**
1. This call is done once before DOC Driver exits
2. In this case SW should free the DMA channel allocated between the CPU and the mDOC H3
3. **fDmaStatus** field should be set to TRUE, when failures occurred; or to FALSE otherwise.

**bOpType = DMA_CONFIG_TRANSACTION** - Used for READ transactions only.
1. This call is done before each HW DMA read transaction
2. In this case SW should configure the transfer descriptor to prepare for HW DMA transactions between mDOC H3 and the CPU
3. The Transfer descriptor (variable from type DMA_Params_S) should be configured as following:
   - **bDiskOnChip_BasePtr** – (IN) mDOC H3 base address
   - **wDiskOnChip_Offset** – (IN) Offset inside mDOC H3 memory area to read the data from.
   - **bDestAddrPtr** – (IN) Destination address
• **wFrameSize** - (IN) Frame size in bytes

• **wFramesInBlock** - (IN) how many frames the transaction contains

• At the end of this call, the DMA controller should be ready to begin DMA transaction as soon as **DMA_REQ#** is asserted.

4. **fDmaStatus** field should be set to TRUE, when failures occurred; or to FALSE otherwise.

   **bOpType = DMA_WAIT_FOR_TRANSACTION_END** – Used for READ transactions only.

   1. This call waits to the DMA read transaction to finish.
   2. The macro should return only upon the assertion of the mDOC H3 interrupt or expiration of user defined timeout.
   3. mDOC H3 cannot be accessed in the middle of a DMA transaction.
   4. If an error condition occurs (for example - timeout), the macro returns TRUE in **fDmaStatus** field.

   5. **wFramesXferred** field from the variable from type DMA_Params_S should be set to the number of frames were successfully transferred. In case of ECC errors (in non mDOC H3 devices), this value may be smaller than the entire block.

   **bOpType = DOCH_DMA_CONFIG_TRANSACTION_HOST_TO_DEVICE** – Used for WRITE transaction only.

   1. This call is done before each HW DMA write transaction.
   2. In this case SW should configure the transfer descriptor (variable from type **DMA_Params_S**) to prepare for HW DMA transactions between the CPU and mDOC H3
   3. The Transfer descriptor should be configured as follows:

   • **bDestAddrPtr** – Destination address
   • **bDiskOnChip_BasePtr** – (IN) mDOC H3 base address
   • **wDiskOnChip_Offset** – (IN) Offset inside mDOC H3 memory area to write to.
   • **wFrameSize** – (IN) Frame size in bytes.
   • **wFramesInBlock** – (IN) Number of frames the transaction contains

   4. At the end of this call, the DMA controller should be ready to begin DMA transaction as soon as **DMA_REQ#** is asserted.

   5. **fDmaStatus** field should be set to TRUE, when failures occurred; or to FALSE otherwise.

   **bOpType = DOCH_DMA_WAIT_FOR_WRITE_TRANSACTION_END DEVICE** - Used for WRITE transaction only

   1. This call waits to the DMA write transaction to finish.
   2. The macro should return only upon the assertion of the mDOC H3 interrupt or expiration of user defined timeout.
3. mDOC H3 cannot be accessed in the middle of a DMA transaction.

4. If an error condition occurs (for example - timeout), the macro returns TRUE in fDmaStatus field.

5. **wFramesXferred** field from the variable from type DMA_Params_S should be set to the number of frames were successfully transferred.
   - The **bIrHandle** field of **DMA_Params_S** structure indicates the socket handle used by DOC Driver for the specific socket in use.
   - The **DrvParamsPtr** field of **DMA_Params_S** structure is an optional pointer saved internally in the DOC Driver pointing to the private record used by the DMA implementation module. This field is not used by DOC Driver, but it may be used to simplify some DMA implementations.

Please refer to *flHwConfig()* API described in section 6.8.5 for further details about enabling HW DMA support, configuring the interrupt types (edge or level) and setting their polarity.

In addition the following flag MUST be defined to enable DMA (both HW and SW):

```
#define FL_USE_DMA_ON_WRITE (flsystem.h)
```

### 4.4.8 DMA Configuration Notes

- Before calling *flHwConfig* the initial DMA mode can be determined by the
  *FL_DMA_INIT(wSockeNo)* macro (*flsystem.h*). When configured to
  *FL_DMA_HW_ENABLED* (or function, returns *FL_DMA_HW_ENABLED*), DOC Driver will call
to init routine as part of system initialization. Otherwise – Configure *FL_DMA_INIT* to
  *FL_DMA_HW_DISABLED*.

- DMA may be used with Burst Mode. Please, refer to section mDOC H3 Synchronous Burst Mode for definition of Burst Mode support and configuration.

- Additional configurations related to working with DMA:
  ```
  #define DOCH_DMA_CHECK_BUFFER(bufferPtr, dwSectorCount)
  ```
  Should be customized to user function, which checks, that the buffer for DMA transaction is valid (alignment for example). This function must be from the following prototype:

  ```
  FLBoolean check_buffer_function( void * bufferPtr, FLDword dwSectorCount);
  ```

  Where:

  - **bufferPtr** is a pointer to user buffer destination address that will be used in DMA transaction (*wFramesInBlock* field from **DMA_Params_S**)
  - **dwSectorCount** is a number of frames in the buffer pointed to by bufferPtr
    (*bDestAddrPtr* field from **DMA_Params_S**).

  The function must return TRUE, when buffer is OK, FALSE otherwise.
5. APPLICATION CUSTOMIZATION

5.1 Customizing the DOC Driver Compilation Flags

Most of the DOC Driver features and implementation limits can be customized to reduce code size and RAM requirements. For example, the maximum number of sockets and volumes, support for cache and so on. Note that many of the DOC Driver API functions are available only when the appropriate customization option is set.

To build a DOC Driver application that contains only the features and implementation limits you need, set the DOC Driver service customization options (in `FLCUSTOM.H`) as described in Table 7.

### Table 7: Block Device Service Requirement Definitions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General Package Customization</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software versions</td>
<td>Defines the strings to be returned by the <code>flVolumeInfo()</code> function in the <code>driverVer</code> and <code>OSAKVer</code> fields.</td>
<td>The string must not be greater than 10 characters. Other than being returned by the <code>flVolumeInfo()</code> routine this value has no actual affect on the DOC Driver software.</td>
</tr>
<tr>
<td></td>
<td>#define driverVersion “OS Less”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#define OSAKVersion “1.0.0”</td>
<td></td>
</tr>
<tr>
<td><strong>Package Limitations Customization</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of physical sockets</td>
<td>Defines the maximum number of supported mDOC H3 sockets.</td>
<td>The actual number of sockets depends on which socket controllers are actually registered and the number of physical devices in the system.</td>
</tr>
<tr>
<td></td>
<td>#define FL_SOCKETS 1</td>
<td></td>
</tr>
<tr>
<td>Total number of partitions</td>
<td>Defines the maximum number of partitions supported on all the sockets in the system (including IPL and OTP partitions).</td>
<td>The actual number depends on the number of mounted sockets and the flash format placed on each device.</td>
</tr>
<tr>
<td></td>
<td>#define FL_VOLUMES (16*FL_SOCKETS)</td>
<td></td>
</tr>
</tbody>
</table>
## Feature Description Notes

### DOC Driver heap size

DOC Driver uses the `FL_MALLOC` and `FL_FREE` routines to allocate the memory it requires. However, if your native system does not have standard libraries, you may use the DOC Driver implementation for dynamic memory allocation for those routines called `flMalloc` and `flFree`.

In such cases you will need to define the heap size for those routines using the following definition at compile time:

```c
#define FL_TRUEFFS_HEAP_SIZE 200000
```

More details in section 5.4.

### DOC Driver heap pointer

Optionally you can specify the location of the DOC Driver SDK heap using this definition:

```c
#define FL_TRUEFFS_HEAP_POINTER 0x8000000
```

When this definition is available, DOC Driver will not allocate a static heap, but will simply use the given pointer.

## Features IN/OUT Customization

### Read only

Use the `#define FL_READ_ONLY` directive. When this option is selected, only API functions that do not involve write operations to the media can be called.

This option can be used to create read-only applications with low RAM/ROM requirements.

### Formatting

Selects or deselects the following flash formatting routines:

- `flFlashFormat ()`
- `flEraseBD ()`
- `flUnformat ()`

```c
#define FL_FORMAT_VOLUME
```

Note that file system formatting is separated from the flash format routine, hence it is compiled using a different compilation flag.

### Block device functionalities

Selects or deselects the absolute sector access function.

```c
#define FL_ABS_READ_WRITE
```

### Use environment variables

When selected, environment variables are supported.

```c
#define FL_ENVIRONMENT_VARS
```

For further information, refer to the section on environment variables. For further information, see section 5.2.

### Support IOCTL interface

Enables standard Extended Functions interface support.

```c
#define FL_IOCTL_INTERFACE
```

For further information, refer to the section on extended functions interface and to the *DOC Driver 1.0 Extended Functions Developer Guide*.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware OTP functions</td>
<td>Compiles the code for accessing the OTP partition, Unique ID and customer ID.</td>
<td>#define HW_OTP</td>
</tr>
<tr>
<td>Hardware write protection</td>
<td>Compiles the hardware protection manipulation functions.</td>
<td>#define HW_PROTECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware protection is maintained even when using a DOC Driver-based driver or an application not compiled with HW_PROTECTION.</td>
</tr>
<tr>
<td>Debugging of defective HW</td>
<td>Compile the code for reading after every write operation.</td>
<td>#define FL_VERIFY_WRITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When FL_ENVIRONMENT_VARS is defined the read after write mode is controlled using the dedicated environment variable.</td>
</tr>
<tr>
<td>Application exit</td>
<td>Selects whether or not the flExit() function is to be included.</td>
<td>#define FL_EXIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If your application exits, that is, if it does not run indefinitely, call the flExit() routine before exiting from the application.</td>
</tr>
<tr>
<td>Compile extended disk information routine</td>
<td>DOC Driver provides an extended information routine for internal msystems utilities named flGetExtendedDiskInfo().</td>
<td>#define FL_EXTENDED_DISK_INFO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>As this is an internal function, backward compatibility for this function is not guarantied.</td>
</tr>
<tr>
<td>Remove IPL functionality</td>
<td>Removes read and write IPL functions.</td>
<td>#define NO_IPL_CODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that once written, the IPL is visible through mDOC H3 memory window regardless of the compilation flag.</td>
</tr>
<tr>
<td>Additional Customizations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOCH_ACCESS_NANOSEC</td>
<td>The NOR access layer access time in nanoseconds.</td>
<td></td>
</tr>
<tr>
<td>DOCH_DPD_DEFAULT_ACTIVE_MODE</td>
<td>Defines the active mode to be set to device while initialization. Possible values are:</td>
<td>#define DOCH_WM_NORMAL - Auto DPD disabled; Work mode is Turbo mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DOCH_WM_LOW_FREQ - Auto DPD disabled; Work mode is PowerSave mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DOCH_WM_NORMAL_AND_AUTO_STBY - Auto DPD enabled; Work mode is Turbo mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DOCH_WM_LOW_FREQ_AND_AUTO_STBY - Auto DPD enabled; Work mode is PowerSave mode.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>DOCH_DPD_DEFAULT_INACTIVE_MODE</td>
<td>DOCH_IM_IDLE - mDOC H3 Standby power mode.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_IM_DPD - mDOC H3 Deep Power Down mode.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_IM_IDLE_2_DPD - mDOC H3 moves into Standby mode when inactive, and goes into Deep Power Down mode after a configurable timeout.</td>
<td></td>
</tr>
<tr>
<td>DOCH_DPD_DEFAULT_DPD_TIMEOUT</td>
<td>Active to inactive timeout.</td>
<td></td>
</tr>
<tr>
<td>DOCH_32K_SLIDING_WINDOW</td>
<td>Should be defined for msystems PCI EVB</td>
<td></td>
</tr>
<tr>
<td>ATA_MAX_NUM_DEVICES</td>
<td>Maximum number of supported devices per socket. Possible values – 1 or 2.</td>
<td></td>
</tr>
<tr>
<td>DOCH_UNSERVICED_INTERRUPT</td>
<td>SDK calls this macro, when suspects an interrupt had occurred and wasn't serviced by SDK (should clear HW).</td>
<td></td>
</tr>
<tr>
<td>DOCH_SET_WINDOW_OFFSET</td>
<td>This macro should be customized to function, which sets the window offset (used for PortaDOC and msystems PCI EVB)</td>
<td></td>
</tr>
<tr>
<td><strong>DOCH HW Registers Defaults</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOCH_BURST_WRITE_MODE_DEFAULT</td>
<td>Defines default value of BURST WRITE MODE register. Logical OR of one value from each of the following groups:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 1:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_DISABLE - Must be defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 2:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_HOLD_1_CLK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_HOLD_2_CLK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 3:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_LEN_4_CYC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_LEN_8_CYC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_LEN_16_CYC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_BURST_LEN_32_CYC</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>DOCH_BURST_READ_MODE_DEFAULT</strong></td>
<td>Defines default value of BURST READ MODE register.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Logical OR of one value from each of the following groups:</td>
<td></td>
</tr>
</tbody>
</table>
|                                  |   Group 1:  
|                                  |     **DOCH_BURST_DISABLE** - Must be defined                               |       |
|                                  |     Group 2:  
|                                  |     **DOCH_BURST_HOLD_1_CLK**                                              |       |
|                                  |     **DOCH_BURST_HOLD_2_CLK**                                              |       |
|                                  |     Group 3:  
|                                  |     **DOCH_BURST_LEN_4_CYC**                                               |       |
|                                  |     **DOCH_BURST_LEN_8_CYC**                                               |       |
|                                  |     **DOCH_BURST_LEN_16_CYC**                                              |       |
|                                  |     **DOCH_BURST_LEN_32_CYC**                                              |       |
| **DOCH_IPL_CTRL_DEFAULT**        | Defines IPL value of the IPL control register.                             |       |
|                                  | Logical OR of one value from each of the following groups:                  |       |
|                                  |   Group 1:  
|                                  |     **DOCH_IPL_WRITE_DISABLE**                                             |       |
|                                  |     **DOCH_IPL_WRITE_READY**                                               |       |
|                                  |     **DOCH_IPL_WRITE_ENABLE**                                              |       |
|                                  |   Group 2:  
|                                  |     **DOCH_IPL_ALL_CS_ENABLED**                                            |       |
|                                  |     **DOCH_IPL_CLOSE_2LOW_IPLS**                                           |       |
|                                  |     **DOCH_IPL_CLOSE_2HIGH_IPLS**                                          |       |
|                                  |     **DOCH_IPL_CLOSE_ALL_IPLS**                                            |       |
| **DOCH_WARM_BOOTCTRL_DEFAULT**   | Defines the value of WARM BOOT CONTROL register.                           |       |
|                                  | Logical OR of one value from each of the following groups:                  |       |
|                                  |   Group 1:  
|                                  |     **DOCH_WARM_RST_POLARITY_HIGH**                                        |       |
|                                  |     **DOCH_WARM_RST_POLARITY_LOW**                                         |       |
|                                  |   Group 2:  
<p>|                                  |     <strong>DOCH_WARM_RST_BURST_ON</strong>                                             |       |
|                                  |     <strong>DOCH_WARM_RST_BURST_NO_CHNG</strong>                                        |       |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCH_POWER_DOWN_DEFAULT</td>
<td>Defines the value of POWER DOWN register. Logical OR of one value from each of the following groups:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 1:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DPD_PIN_POL_HIGH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DPD_PIN_POL_LOW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 2:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DPD_PIN_ENABLED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DPD_PIN_DISABLED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 3:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DPD_WAKEUP_HOST_CE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DPD_WAKEUP_ASYNC_CLK</td>
<td></td>
</tr>
<tr>
<td>DOCH_DMA_CTRL_DEFAULT</td>
<td>Defines the value of DMA CONTROL register. Logical OR of one value from each of the following groups:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 1:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DMA_REQ_POL_LOW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DMA_REQ_POL_HIGH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 2:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DMA_REQ_EDGE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DMA_REQ_LEVEL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group 3:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_DMA_PULSE_WIDTH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#define DOCH_DMA_PULSE_WIDTH -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defines the width of the DMA_REQ signal, which calculated as PULSE_WIDTH * ICMU_CLK (cycle, period is equal ~20ns), maximum 32 ICMU clocks. Note: If the value is zero the DMA_REQ signal will not be asserted.</td>
<td></td>
</tr>
<tr>
<td>DOCH_DMA_NEGATION_CTRL_DEFAULT</td>
<td>Defines the value of DMA NEGATION register. DMA programmable negation. Possible values are</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0-1023: Number of clocks before SMA_request signal will be negated (at the end of each sector).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: Dma negation must be smaller than DRQ_LEN</td>
<td></td>
</tr>
<tr>
<td>DOCH_SLOCK_DEFAULT</td>
<td>Defines the value of Software. Lock register. Possible values are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_SLOCK_OFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOCH_SLOCK_ACTIVE</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------</td>
</tr>
</tbody>
</table>
| DOCH_ENDIAN_CTRL_DEFAULT | Defines the value of ENDIAN CONTROL register. Possible values are:  
HIB_END_SWAP_ON – Bytes swap ON  
HIB_END_SWAP_OFF – Bytes swap OFF | |
| DOCH_OPERATION_MODE_CTRL_DEFAULT | Logical OR of one value from each of the following groups:  
Group 1:  
HIB_ADDR_SHIFT  
HIB_NO_ADDR_SHIFT  
Group 2:  
HIB_PIPE_ACCESS  
HIB_NON_PIPE_ACCESS | |
5.2 Registering the mDOC H3 Device

The mDOC H3 device must be registered through a call to `flRegisterDochParams()`, located in `FLCUSTOM.C`.

```c
FLStatus flRegisterDochParams(void)
{
    flRegisterDOCH_Hal_NOR();
    /* Register Socket */
    flRegisterDOCH3SOC (FL_DOC_ADDRESS);
    return flOK;
}
```

Note: `FL_DOC_ADDRESS` specifies the start of the memory window of the mDOC H3 device.
5.3 Run-Time Customization and Environment Variables

In addition to the compile-time definitions, there are also customization options that can be selected at runtime. These options are controlled by the environment variables software mechanism.

For each environment variable, there is an internal DOC Driver variable controlled by a dedicated DOC Driver API.

5.3.1 Block Device Environment Variables

- To configure the mDOC H3’s block device environment variables during runtime, ensure that `FL_ENVIRONMENT_VARS` is defined (in `FLCUSTOM.H`) and call the respective routine from the following list:
- Use `flSetEnvVolume()` to set the environment variables for a specific partition.

  ```c
  FLStatus flSetEnvVolume(FLEnvVars variableType, FLByte socket, FLByte volume, FLDword value, FLDword *prevValue)
  ```
- Use `flSetEnvSocket()` to set the environment variables for a specific mDOC H3 (selected with the socket handle).

  ```c
  FLStatus flSetEnvSocket(FLEnvVars variableType, FLByte socket, FLDword value, FLDword *prevValue)
  ```
- Use `flSetEnvAll()` to set the environment variables for all mDOC H3 devices on your system.

  ```c
  FLStatus flSetEnvAll(FLEnvVars variableType, FLDword value, FLDword *prevValue)
  ```

Notes:
1. `*prevValue` returns the previous value of the environment variable to a user variable (passed by reference).
2. Default values are denoted by square parentheses, for example, `[FL_XX]`.
3. Defaults can be overwritten by using the compile time default macros mentioned for each of the variables listed in Table 8.
### Table 8: Default Variable Values

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Value*</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL_VERIFY_WRITE_BDTL</td>
<td>[FL_OFF]</td>
<td>Selects the verify write operation mode for all partitions: FL_OFF: Full protection against power failure in the block device level. See section 8. FL_ON: Perform verification of every write operation. Usually used for HW debugging. Setting the default at compile time can be done using: FL_DEFAULT_VERIFY_WRITE_MODE</td>
</tr>
<tr>
<td>FL_SET_AUTO_DPD_MODE</td>
<td>[FL_OFF]</td>
<td>Set Automatic Deep Power Down Mode at block device level. FL_OFF: DPD mode is controlled by the user using DOC Driver extended function calls and is not used automatically by mDOC H3 device. FL_ON: mDOC H3 will automatically enter the DPD mode after every block device operation. Setting the default at compile time can be done using: FL_DEFAULT_BD_DPD_MODE</td>
</tr>
<tr>
<td>FL_SET_ACTIVE_DPD_MODE,</td>
<td>Default values:</td>
<td>See section 8.3 for explanation. The default values (compilation time customization) may be changed by defining: DOCH_DPD_DEFAULT_ACTIVE_MODE, DOCH_DPD_DEFAULT_INACTIVE_MODE and DOCH_DPD_DEFAULT_DPD_TIMEOUT in ffsystem.h file.</td>
</tr>
<tr>
<td>FL_SET_INACTIVE_DPD_MODE</td>
<td>[DOCH_WM_NORMAL_AND_AUTO_STBY] For Active mode: [DOCH_IM_IDLE_2_DPD] For Timeout: 0</td>
<td></td>
</tr>
<tr>
<td>FL_SET_TIMEOUT_DPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FL_ENV_ATA_DEBUG</td>
<td>[FL_OFF]</td>
<td>Set Debug mode of mDOC H3. Each command sent to the device will automatically retrieve and print debug messages created by ETFFS.</td>
</tr>
</tbody>
</table>
For backwards-compatibility reasons, the following environment variable is defined as a global variables by the user (usually in `FLCUSTOM.C`) and therefore do not have any default values. However the DOC Driver `flInit()` routine calls the function `flSetEnvVar(void)`, which initializes these environment variables. The user **must** implement this function, typically in `FLCUSTOM.C`.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Corresponding Variable Type*</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flUse8Bit</code></td>
<td><code>FL_DOC_8BIT_ACCESS</code></td>
<td><code>FL_OFF</code></td>
</tr>
<tr>
<td></td>
<td>Set by:</td>
<td><code>FL_ON</code></td>
</tr>
<tr>
<td></td>
<td><code>flSetEnvAll()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Before calling:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>flInit()</code></td>
<td></td>
</tr>
</tbody>
</table>
5.4 Replacing Malloc/Free

DOC Driver uses macros `FL_MALLOC/FL_FREE` and `FL_FAR_MALLOC/FL_FAR_FREE` to allocate and de-allocate its buffers. Typically `FLSYSTEM.H` includes the following definitions:

```c
#define FL_MALLOC malloc
#define FL_FREE(a)  free((void *)a)
```

However, occasionally these definitions are not adequate. For example, if you are implementing a kernel-level block device, you might need to use kernel-level allocation routines.

If you do not want DOC Driver to use your library or system allocation routines, then DOC Driver can use an internal memory-allocation module.

To use this internal malloc implementation, you need to make the following changes:

- Add module `FLMALLOC.C` to your project
- Replace the `FL_MALLOC` and `FL_FREE` definitions with:

```c
#include "flmalloc.h"
#define FL_MALLOC flMalloc
#define FL_FREE(a) flFree((void *)a)
```

Note: User application can also use the `flMalloc` as the `FL_TRUEFFS_HEAP_SIZE` is increased by the necessary amount. Beware: if user application experiences a memory corruption problem, sharing the heap will increase the chance of DOC Driver buffers corruption.

You can also use the `FL_TRUEFFS_HEAP_POINTER` to specify the location of the heap physical memory. In this case, DOC Driver will not allocate the memory statically, but will simply use the supplied pointer.
DOC Driver also exports a variety of routines for inspecting heap:

**Table 10: FL_MALLOC/FL_FREE Definitions**

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define FL_TRUEFFS_HEAP_SIZE</td>
<td>Place the amount of heap here. Size depends on many factors.</td>
</tr>
<tr>
<td>#define FL_TRUEFFS_HEAP_POINTER</td>
<td>Optionally point to your free RAM address space.</td>
</tr>
<tr>
<td>#define FL_HEAP_WALK</td>
<td>If you wish to use one of the routines described below: flHeapWalk, flHeapChk, flDisplayHeap, flBiggestFreeMemoryChunk</td>
</tr>
<tr>
<td>#define FL_DISPLAY_HEAP</td>
<td>If you wish to use the flDisplayHeap routine - For debug purposes. Routine FLBoolean flDisplayHeap(void); uses macro flHeapPrint defined in FLMALLOC.H</td>
</tr>
<tr>
<td>#define FL_REPORT_BIGGEST_MEMCHUNK</td>
<td>If you wish to use the flBiggestFreeMemoryChunk routine. Routine FLDword flBiggestFreeMemoryChunk(void); returns the maximum block that can be allocated, which is different from the total free space in the heap, due to possible heap de-fragmentation</td>
</tr>
</tbody>
</table>

Routine *FLBoolean flHeapWalk* (flInspectHeapFunction, void * extraInfo) provides the user with means of inspecting heap blocks. The definition of the call-back routine flInspectHeapFunction and heap block information structure *flMemChunk* is located in FLMALLOC.H. flDisplayHeap and flBiggestFreeMemoryChunk provide examples of flHeapWalk use.

*flHeapChk* is a macro that invokes flHeapWalk in a way that does not use callback routine.

In the event the heap routine detects heap corruption, it uses the *flHeapPrint* macro to report the problem, and returns failure (NULL from flMalloc, 0 from flBiggestFreeMemoryChunk, FALSE from flFree, flHeapWalk/flHeapChk or flDisplayHeap).
6. **DOC Driver Application Programmer’s Interface (API)**

6.1 **Introduction**

The DOC Driver BD Software Developers Kit (SDK) is designed to be integrated and linked to a driver or an application. DOC Driver exports an Application Programmer’s Interface (API) providing full block-device functionality, in addition to various special capabilities like write protection, formatting, and so on. The `BLOCKDEV.H` header file defines this API.

All calls to DOC Driver (with the exception of the `flExit()`, `flInit()`, `flBuildGeometry()`, `flSetEnvLogicalVolume()`, `flSetEnvVolume()`, `flSetEnvSocket()`, `flSetEnvAll()`, `flGetDocBusRoutine()` and `flSetDocBusRoutine()` functions take the following form:

```c
status = SomeOperation(&ioRequestPacket);
```

Where:

- **SomeOperation** = Name of the DOC Driver function called.
- **IoRequestPacket** = A parameter structure supplied by the caller.

The structure is defined by the `Ioreq` type defined in `FLCOMMON.H`, as follows:

```c
typedef struct {
    FLHandle   irHandle;  /* The drive handle or file handle of the operation */
    FLDword    irFlags;   /* Function-specific flags*/
    void *     irPath;    /* Pointer to Unicode string file path and name */
    void *     irData;    /* Pointer to user-buffer */
    FLSDword   irLength;  /* Additional argument */
    FLSDword   irCount;   /* Additional argument */
} IOreq;
```

And sometimes using the following aliases:

```c
#define irSectorCount irCount    /* Used by sector read & write calls */
#define irSectorNo irLength
#define irByteCount irCount      /* Used by physical read & write calls */
#define irAddress irLength
```

All DOC Driver functions take the same parameter structure, although the meaning of the parameter fields varies from function to function. The `Ioreq` fields are described on a function-specific basis.
Status: Completion status of the DOC Driver operation. The status is defined with an `FLStatus` type, and its values are found in `FLCOMMON.H`. A successful operation always returns the value `flOK` (0). Other values indicate an error or a special condition.

6.2 Interface Terms

It is important to read and master the interface terms (see Section 2.1) before further reading of the DOC Driver API.

6.3 DOC Driver Function Group List

DOC Driver functions can be broken down into the groups described in Table 11. The following sections describe the usage and API of the each of the functions listed below.

**Table 11: Block Device Functions**

<table>
<thead>
<tr>
<th>DOC Driver Function Group</th>
<th>Sub Group</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Device Functions</td>
<td>Volume/Partition Manipulation (see Section 6.4, page 47)</td>
<td>flAbsMountVolume()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flDismountVolume()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flCheckVolume()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flVolumeInfo()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flCountVolumes()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flSectorsInVolume()</td>
</tr>
<tr>
<td></td>
<td>Device Format (see Section 6.4.1, page 53)</td>
<td>flFlashFormat()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flEraseBD()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flUnformat()</td>
</tr>
<tr>
<td>Block Device I/O</td>
<td></td>
<td>flAbsDelete()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flAbsSecureDelete()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flAbsRead()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flAbsWrite()</td>
</tr>
<tr>
<td>Special Functions (see Section 6.4.4, page 64)</td>
<td></td>
<td>ffCheckBeforeWrite()</td>
</tr>
<tr>
<td>IPL support</td>
<td>Writing and readin Initial Program Loader (IPL) (see Section 6.5, page 65)</td>
<td>flWriteIPL()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flReadIPL()</td>
</tr>
<tr>
<td>Special DOC Driver functions (see Section 6.6, page 69).</td>
<td>Global DOC Driver calls (see Section 6.6.1, page 69)</td>
<td>flInit()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flExit()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flBuildGeometry()</td>
</tr>
<tr>
<td>DOC Driver Function Group</td>
<td>Sub Group</td>
<td>Function Name</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>DOC Driver runtime configuration routines (see application customization section 4.3.3.1)</td>
<td></td>
<td>flSetEnvLogicalVolume()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flSetEnvVolume()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flSetEnvSocket()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flSetEnvAll()</td>
</tr>
<tr>
<td></td>
<td>Low level information (See section 6.6.2 page 70)</td>
<td>flGetPhysicalInfo()</td>
</tr>
<tr>
<td></td>
<td>Internal Device Information (See section 6.6.3 page 72)</td>
<td>flGetExtendedDiskInfo()</td>
</tr>
<tr>
<td>Logical format functions.</td>
<td>Device Format (see Section 6.7 on page 75)</td>
<td>flFindLogicalPartition()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flFormatFS()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flCreateLogicalPartitions()</td>
</tr>
<tr>
<td></td>
<td>IOCTL Functions</td>
<td>flIoctl()</td>
</tr>
<tr>
<td></td>
<td>Hardware Read/Write Protection (see Section 6.8.1, page 85)</td>
<td>flIdentifyProtection()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flInsertProtectionKey()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flRemoveProtectionKey()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flHardwareProtectionLock()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flChangeProtectionKey()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flChangeProtectionType()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flApplyStickyLock()</td>
</tr>
<tr>
<td></td>
<td>One Time Programming (OTP) Operations (see Section 6.8.2, page 93)</td>
<td>flOTPSize()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flOTPWriteAndLock()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flOTPRead()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flGetUniqueID()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flGetCustomerID()</td>
</tr>
<tr>
<td>Accessing mDOC H3 Extended Functionality (see Section 0, page 75).</td>
<td>Power Saving Mode</td>
<td>flDeepPowerDownMode()</td>
</tr>
<tr>
<td></td>
<td>Recover from reset assertion or turning of the power</td>
<td>flRecoverFromPowerLoss()</td>
</tr>
<tr>
<td></td>
<td>HW Configuration</td>
<td>flHwConfig</td>
</tr>
</tbody>
</table>
6.4 Block Device Functions

6.4.1 Volume/Partition Manipulation

flAbsMountVolume

Function

Mounts the partition, preparing it for further access.

This function verifies only the physical format and does not check the FAT format.

Mounting the device is mandatory before any block device operations can be performed.

The function applies only to a specific partition. If the physical drive has several partitions, each partition must be mounted individually.

Ioreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| irHandle  | LSB 0-3: Socket number  
            LSB 4-7: Partition number |
| irData    | Pointer to progress callback routine in case \textit{FL\_REPORT\_MOUNT\_PROGRESS} compilation flag is defined in \texttt{FLCUSTOM.H}. |

Returns

<table>
<thead>
<tr>
<th>FLStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{flOK}</td>
<td>on success, otherwise failed.</td>
</tr>
</tbody>
</table>

\texttt{FLCUSTOM.H} definitions required

None

Note: Partition can be mounted several times (presumably by separate applications or threads). It must be dismounted the same number of times before being considered as dismounted.
flDismountVolume

Function
Dismounts the partition. This call is required for every mount call that was performed.

Notes:
1. The flExit function dismounts all volumes. Therefore if flExit() is called, there is no need to call flDismountVolume().
2. Format procedures will not be performed as long as the media is in use (i.e. mounted). Therefore, if a partition was mounted, make sure to call flDismountVolume() before calling format API's

Ioreq Parameters

- irHandle
  LSB 0-3: Socket number
  LSB 4-7: Partition number

Returns

- FLStatus
  flOK on success, otherwise failed.

FLCUSTOM.H definitions required
No
flCheckVolume

*Function*
Checks if the partition is currently mounted.

*Ioreq Parameters*

- **irHandle**
  - LSB 0-3: Socket number
  - LSB 4-7: Partition number

*Returns*

- **FLStatus**
  - *flOK* if the partition is mounted, non-zero value if the partition is not mounted

**FLCUSTOM.H** definitions required

No
**flVolumeInfo**

**Function**

Returns general information about the partition, for example, where the socket resides, high-level and low-level geometry of the volume and the software versions in use.

A *VolumeInfoRecord* structure containing the above information is returned to a user buffer.

The *VolumeInfoRecord* structure is defined in **BLOCKDEV.H** as follows:

```c
typedef struct {
  FLWord logicalSectors;  /* Number of logical sectors in the partition
                             (including hidden sectors and boot sectors) */
  FLWord bootAreaSize;    /* obsolete, always 0 */
  FLWord baseAddress;     /* Physical base address of the memory window */
  unsigned short flashType;  /* FL_H3 – mDOC H3 device */
  FLWord dwPhysicalSize;  /* capacity of unformatted media, in sectors */
  FLWord dwPhysicalUnitSize;  /* Flash Unit size, in sectors */
  FLByte DOCTYPE;          /* FL_H3 – mDOC H3 device */
  FLByte lifetime;         /* obsolete, always 1 */
  FLByte driverVer[10];   /* Driver layer OS name and version. This field
                           returns the Driver version string definition found in
                           FLCUSTOM.H */
  FLByte OSAKVer[10];     /* DOC Driver version used for this driver. This
                           field returns the OSAK version string definition
                           found in FLCUSTOM.H */
  FLWord cylinders;       /* Media */
  FLWord heads;           /* Geometry */
  FLWord sectors;         /* Parameters */
} VolumeInfoRecord;
/* End structure */
```

**ioreq Parameters**

- **irHandle**
  - LSB 0-3: Socket number
  - LSB 4-7: Partition number

- **irData**
  - Address of buffer that receives the *VolumeInfoRecord* structure
Returns

FLStatus  

f1OK on success, non-zero on failure

FLCUSTOM.H definitions required  
None for the function. Specific information requires specific definitions, as described in the VolumeInfoRecord definition.
**flCountVolumes**

**Function**

Returns the number of partitions in the physical drive.

- The volume does not need to be mounted, nor does this routine actually mount the partition.
- Read-protected partitions will be counted by this function even when they are not accessible.

**Ioreq Parameters**

- **irHandle**
  - LSB 0-3: Socket number

**Returns**

- **irFlags**
  - The number of partitions in the volume
- **FLStatus**
  - **flOK** on success, non-zero on failure

**FLCUSTOM.H definitions required**

- None
flSectorsInVolume

Function
Returns the number of sectors in the partition.

- The volume must be mounted prior to calling this routine.
- Block-Device flSectorsInVolume returns all of the virtual media sectors, regardless of the logical partitions existing on it.
- It might be important to note that Cylinder X Heads X Sectors may produce a lower size than actual sectors in the Disk reported by this routine.

I/Oreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irHandle</td>
<td>LSB 0-3: Socket number</td>
</tr>
<tr>
<td></td>
<td>LSB 4-7: Partition number</td>
</tr>
</tbody>
</table>

Returns

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irLength</td>
<td>The number of sectors in the partition</td>
</tr>
<tr>
<td>FLStatus</td>
<td>flOK on success, non-zero on failure</td>
</tr>
</tbody>
</table>

FLCUSTOM.H definitions required
None
6.4.2 Device Formatting

Most mDOC H3 devices are shipped as unformatted media (raw flash) and require a formatting operation, allowing you to dictate the way you want your data to be stored on mDOC H3. Examples of such options are the percent of the media you want to use, protection attributes, enhanced performance partitions etc.

The following routines are the APIs for these formatting capabilities:

- `flFlashFormat()` routine supplies full mDOC H3 block device formatting capabilities.
- `flEraseBD()` routine erases a specific partition.
- `flUnformat()` routine erases the entire flash disk, returning it to its virgin state.

The code supporting this entire section can be removed by commenting out the `FL_FORMAT_VOLUME` compilation flag.

**flFlashFormat**

**Function**

Formats a physical device by performing a combination of the following actions:

- Formats the device and divide it into partitions.
- Set the protection attributes of the partitions.
- Sets enhanced performance ratio's

**Notes:**
1. Formatting destroys all existing data.
2. All format scenarios require all partitions to be in the dismounted state, and leave them in dismounted state (subsequently requiring a mount call).
3. DOC Driver support up to 14 partitions (in addition to IPL and OTP).
4. All protection keys must be inserted before calling this routine.
5. After placing the OTW protection on a partition, the media can no longer be formatted.
6. After formatting, 92%-93% of media capacity will be available for user data (partitions). The remaining 7%-8% are used for OTP area, IPL, bad blocks and flash management. The reserved area as well as the area available for user data, are fixed per capacity, and do not depend on format parameters.

**Function Structure Definitions**

Formatting is controlled by a set of parameters defined in a `FormatParams3` structure and passed through the `ixData` pointer. This structure is defined in `FLSTRUCT.H` and contains the fields described in Table 12.

Table 12: TL Formatting Section
FLByte percentUse;  
This field is obsolete.

FLByte noOfBDTLPartitions;  
Indicates the number of partitions (1-14).

FLByte noOfBinaryPartitions;  
Obsolete – Must be set to 0.

BDTLPartitionFormatParams3  
* BDTLPartitionInfo;  
Partition information record array (see definition below).

FLByte noOfBDTLPartitions;  
This field is obsolete.

FLByte noOfBinaryPartitions;  
Obsolete – Must be set to 0.

BDTLPartitionFormatParams3  
* BDTLPartitionInfo;  
Partition information record array (see definition below).

FLByte cascadedDeviceNo;  
Reserved set to 0.

FLByte noOfCascadedDevices;  
Reserved set to 0.

FLDword FP_0_RFU_0  
Reserved set to 0.

FLDword FP_1_RFU_0  
Reserved set to 0.

FLProgressCallback  
progressCallback;  
Pointer to user-defined progress callback routine.  
Obsolete - Must be set to NULL.

FLDword bootImageLen  
Reserved set to 0.

FLDword vmAddressingLimit;  
Reserved set to 0.

FLWord embeddedCISlength;  
Reserved set to 0.

FLByte * embeddedCIS;  
Reserved set to 0.

Partition parameters are passed through the BDTLPartitionInfo field. This field is an array of BDTLPartitionFormatParams3 structures defined in FLFORMAT.H with the following fields:
FLDword length;

The size of the usable storage space. The size is rounded upwards to a multiple of a flash unit size. The size of the last partition is automatically calculated, and that parameter is ignored.

FLDword fastAreaLength;

Length/Percentage of the fast area (depending on fastAreaLengthType field).

It is customary to specify the length in percentages and then use one of the following predefined values:

- **TL_BASIC_BOOST** - This is the recommended mode of operation where you sacrifice only 1% of the media space.
- **TL_ENHANCED_PERFORMANCE** - Enhanced performance partition, sacrificing half the capacity for maximum performance gain.
- **TL_NO_BOOST** – Full media utilization, at the expense of lower performance.

The fast area usually takes twice the space of the normal area. This means that a 1MB in the length field and 50% in the fastAreaLength field will produce a 1MB partition that will occupy 1.5MB of the flash media.

It might be useful to note that the FAT area usually takes 1/256 of the media divided by cluster size. In other words using fast area on 1% of the media is more than enough to cover the entire FAT area.

Notes:

- Size is rounded to flash unit size.
- Covering the FAT area of the partition will usually help only the first logical partition.

FLWord lengthType

Sets the length field type using one of the following types:

- **FL_LENGTH_IN_BYTES** 0
- **FL_LENGTH_IN_SECTORS** 1

FLWord fastAreaLengthType

Sets the fastAreaLength field type using one of the following types:

- **FL_LENGTH_IN_BYTES** 0
- **FL_LENGTH_IN_SECTORS** 1
- **FL_LENGTH_IN_PERCENTS** 3

FLWord

Reserved, set to 1
fastAreaVirtualFactor

FLWord noOfSpareUnits;
Reserved, set to 0

FLDword BDTLFP_0_RFU_0
Reserved, set to 0

FLDword BDTLFP_1_RFU_0
Reserved, set to 0

FLDword BDTLFP_2_RFU_0
Reserved, set to 0

FLDword BDTLFP_3_RFU_0
Reserved, set to 0

FLByte protectionKey[8]; The key for the protection.

FLByte protectionType; A combination of the following flags:

  PROTECTABLE This partition can receive protection attributes
  READ_PROTECTED Protect against read operations
  WRITE_PROTECTED Protect against write operations
  LOCK_ENABLED Enables the hardware lock signal
  CHANGEABLE_PROTECTION This partition protection is changeable after format

A partition cannot be defined as OTW protected at the formatting stage. To use this feature, you should format the partition as CHANGEABLE_PROTECTION and use the relevant API to change protection attributes to OTW after placing data on it.

More information regarding the protection attributes can be found in the hardware protection section.
Standard Formatting Parameters

DOC Driver contains standard definition that can be used as a basic format example. This example definition will format the media with a single unprotected disk partition.

The IOreq structure fields sent to flFlashFormat()

IOreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irHandle</td>
<td>LSB 0-3: Socket number</td>
</tr>
<tr>
<td></td>
<td>LSB 4-7: Must be 0</td>
</tr>
<tr>
<td>irData</td>
<td>Address of format parameter structure</td>
</tr>
<tr>
<td>irFlags</td>
<td>TL_NORMAL_FORMAT</td>
</tr>
<tr>
<td></td>
<td>0 Reset the protection logic and thus new protection data will take affect</td>
</tr>
<tr>
<td></td>
<td>immediately.</td>
</tr>
<tr>
<td></td>
<td>TL_LEAVE_SOME_PARTITIONS</td>
</tr>
<tr>
<td></td>
<td>9 Leave some of the previously existing partitions. Create new ones</td>
</tr>
<tr>
<td></td>
<td>afterwards according to format parameters structure.</td>
</tr>
<tr>
<td></td>
<td>TL_DO_NOT_PERFORM_DOWNLOAD</td>
</tr>
<tr>
<td></td>
<td>32 Does not reset the protection logic, and thus new protection data will</td>
</tr>
<tr>
<td></td>
<td>not take affect, until the next reset, or SW download operation.</td>
</tr>
</tbody>
</table>

Returns

<table>
<thead>
<tr>
<th>FLStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flOK</td>
<td>on success, non-zero on failure</td>
</tr>
</tbody>
</table>

FLCUSTOM.H definitions required

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL_FORMAT_VOLUME</td>
</tr>
</tbody>
</table>
flEraseBD

Function

This routine physically erases one partition, erasing all of its data. This API can be used to make sure all previous data is physically erased and can not be reconstructed.

Ioreq Parameters

- **irHandle**: LSB 0-3: Socket number
  LSB 4-7: Partition number.
- **irData**: Reserved (set to NULL)
- **irFlags**: Reserved (set to 0)
- **irLength**: Reserved (set to 0)
- **irPath**: Pointer to progress call back routine , or NULL

Returns

- **FLStatus**: FL_OK on success, non-zero on failure

FLCUSTOM.H definitions required

- FL_FORMAT_VOLUME
**flUnformat**

**Function**

This routine physically erases the entire media returning it to its virgin state. The routine will physically erase the entire media including IPL area, protection data structures and the raw flash blocks returning them to their virgin state, similar to the way they were shipped from the foundry.

This routine should not be used for normal handling of the mDOC H3. In case you need to erase the device, simply call the `flFlashFormat()` routine again with your new parameters.

The routine can be used for debugging purposes. For example you can use this routine if you need to make sure that your build procedure is working properly and you want to re-run it while you can not replace the actual mDOC H3 unit.

**Notes:**

1. This routine cannot open protected areas such as OTP area, OTW partition or a protected partition.
2. The routine leaves the device without any kind of logical format, which will cause future mount operation to fail until it will be reformatted using the `flFlashFormat()` routine.
3. When called with OTP locked – will return flHWProtection error and leave the data on the mDOC H3 without any change. Otherwise – will delete all data, including IPL.

**IOreq Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irHandle</td>
<td>LSB 0-3: Socket number</td>
</tr>
<tr>
<td></td>
<td>LSB 4-7: The partition number must be set to 0</td>
</tr>
<tr>
<td>irData</td>
<td>Reserved (set to NULL)</td>
</tr>
<tr>
<td>irFlags</td>
<td>Reserved (set to 0)</td>
</tr>
<tr>
<td>irLength</td>
<td>Reserved (set to 0)</td>
</tr>
</tbody>
</table>

**Returns**

<table>
<thead>
<tr>
<th>FLStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>f1OK</code> on success, non-zero on failure</td>
</tr>
</tbody>
</table>

**FLCUSTOM.H definitions required**

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL_FORMAT_VOLUME</td>
</tr>
</tbody>
</table>
6.4.3 Block Device I/O (Absolute Sector I/O)

The interface to a partition is exported as a block device, enabling read, write and delete operations to an array of logical sectors. The following routines are the API of those capabilities. The code supporting this entire section can be removed by commenting out the FL_ABS_READ_WRITE compilation flag.

The routines in this section can be used only after the required volume is mounted using the Block-Device-level flAbsMountVolume() routine.

For improved performance when using FAT file system, see the ffCheckBeforeWrite() routine in Section 6.4.4, page 64.

flAbsDelete

Function

Marks one or more consecutive absolute sectors as logically deleted. This function is not required except in special circumstances, as explained below.

The function flAbsDelete() informs ETFFS that these sectors are deleted and therefore become candidates for space reclamation.

ETFFS does not guarantee that the sectors will actually be erased.

Ioreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irHandle</td>
<td>LSB 0-3: Socket number</td>
</tr>
<tr>
<td></td>
<td>LSB 4-7: Partition number</td>
</tr>
<tr>
<td>irSectorNo</td>
<td>First absolute sector to delete</td>
</tr>
<tr>
<td>irSectorCount</td>
<td>Number of absolute sectors to delete</td>
</tr>
</tbody>
</table>

Returns

FLStatus

f1OK on success, non-zero on failure

FLCUSTOM.H definitions required

FL_ABS_READ_WRITE
flAbsSecureDelete()

Function

This routine has the same API and functionality as the `flAbsDelete()`.

The difference is that it makes sure that there are no traces of the previous sectors data on the flash media.

This routine takes much longer to execute, but may come in handy, when deleting sensitive information. It is recommended to use it on a large range of sectors and not on single sectors as it requires more time for the normal write operation.

flAbsRead

Function

Reads one or more consecutive absolute sectors.

Ioreq Parameters

- **irHandle**
  - LSB 0-3: Socket number
  - LSB 4-7: Partition number

- **irSectorNo**
  - First absolute sector to read

- **irSectorCount**
  - Number of absolute sectors to read

- **irData**
  - Buffer to read to

Returns

- **FLStatus**
  - `flOK` on success, non-zero on failure

FLCUSTOM.H definitions required

- `FL_ABS_READ_WRITE`
**flAbsWrite**

**Function**
Writes one or more consecutive absolute sectors.

**Ioreq Parameters**
- **irHandle**: LSB 0-3: Socket number
  LSB 4-7: Partition number
- **irSectorNo**: First absolute sector to write
- **irSectorCount**: Number of absolute sectors to write
- **irData**: Buffer to write to

**Returns**
- **FLStatus**: FL_OK on success, non-zero on failure

**FLCUSTOM.H** definitions required
- **FL_ABS_READ_WRITE**
6.4.4 Using Special Functions

**ffCheckBeforeWrite**

**Function**

Checks if the sector(s) that are about to be written using the routine `flAbsWrite()` are FAT sector(s). If so, the function activates the FAT filter.

The FAT filter detects deleted clusters by comparing new FAT entries to old FAT entries, and physically marks all sectors in deleted clusters as deleted. These deleted sectors are reclaimed in the next garbage collection process. This improves performance by increasing the available free flash media space for consecutive writes.

When mounting the device with the `flAbsMountVolume()` routine (the common case for block device drivers) the FAT filter must be activated. To activate the FAT filter simply call `ffCheckBeforeWrite()` immediately before calling `flAbsWrite()` using the parameters (Ioreq) required by `flAbsWrite` (Ioreq). The FAT filter mechanism ignores non-FAT partitions.

If the partition was further partitioned by FAT (using utilities such as FDISK) and there is more than one FAT partition, mount the device using `flAbsMountVolume()`. The `ffCheckBeforeWrite()` routine monitors FAT table access of all the FAT-formatted partitions.

Note: The `ffCheckBeforeWrite()` function is not a standard DOC Driver API, and is not protected by the internal mutex mechanism. Therefore, in a multi-processing environment, an external mutex should be added to avoid multiple entrances.

**Ioreq Parameters**

- `irHandle` LSB 0-3: Socket number
  LSB 4-7: Partition number
- `irSectorNo` First absolute sector to write (sector numbers are counted starting with the DOS boot sector, whose sector number is 0)
- `irSectorCount` Number of absolute sectors to write
- `irData` Buffer to write from

**Returns**

- FLStatus
  - `flOK` on success, non-zero on failure

**FLCUSTOM.H** definitions required

- `FL_ABS_READ_WRITE`
6.5 Writing and Reading the Initial Program Loader (IPL)

flWriteIPL

Function

Writes the user buffer to the IPL partition.

1. Virtual RAM mode:
   - When the device is configured to work in 8KB window (FL_IPL_128K_WINDOW_MODE not specified), the 8KB window becomes an SRAM like area, filled with 8KB of IPL code/data. Maximum possible IPL data size is 8KB.
   - When the device is configured to work with a 128KB window (FL_IPL_128K_WINDOW_MODE specified), the entire 128KB is SRAM like. When crossing 32KB segments, device BUSY signal will be asserted until the data is loaded to the internal SRAM. Maximum possible IPL data size is 128KB.

2. Normal mode:
   - When device configured to work in 2KB window (FL_IPL_128K_WINDOW_MODE not specified), 2KB window becomes an SRAM like area, filled with 2KB of IPL code/data. Maximum possible IPL data size is 2KB.
   - When device configured to work with 128K window (FL_IPL_128K_WINDOW_MODE specified) 32KB window, from the offset of 0KB inside 128KB becomes an SRAM like area, filled with 32KB of IPL code/data. Maximum possible IPL data size is 32KB.

3. Paged mode:
   - Maximum possible IPL data size in paged mode is 254KB
   - The Paged RAM Boot feature uses the IPL SRAM as two 1KB sections. The first section (offset 0) provides constant data, while the other section (offset 0x400) can be downloaded sequentially with flash data. One application of this feature is to support Secure Boot requirements. The Paged RAM Boot feature does not support XIP (unlike the Virtual RAM Boot feature. In non-alternate mode (FL_IPL_ALTERNATE_MAP not specified), those 2KB are always found at the start of the mDOC H3 window, regardless of window size (128KB or 2KB).
   - After a hardware or software reset, mDOC H3 initializes the first 2KB of RAM from data stored in the IPL area at mDOC H3. The Paged RAM Boot feature permits 1KB of the internal SRAM to be downloaded upon receiving a command sequence from one of many 1KB virtual pages. Since the DOC H3 BUSY# output is not asserted by a page-load operation, a polling procedure is required to determine when the download is complete. An XIP operation from the DOC H3 RAM is not supported during this polling operation, so it must be executed from system RAM or ROM instead.
• When multiple DOC H3 devices are cascaded, Paged RAM downloads occur only on the first mDOC H3 device in the cascaded configuration (device-0). The other cascaded device moves to Reset mode when a Paged RAM download is initiated.

• Only sequential page load is supported: With each request, the next 1KB page is loaded. Direct page load (with page index parameter) is not supported.

• The following procedure must be performed to load a new RAM page:
  o Write the value 71H to the Paged RAM Command Register, and then specify the page to be downloaded by writing to the Paged RAM Select Register. No other write cycles are permitted between these two write cycles.
  o Perform 20 read cycles from address device 400H to provide a time delay to ensure compatibility with future products.
  o Begin polling device address 400H until the least significant bit no longer toggles as follows:

    do { 
        X = DOC[400H] & 1;
        Y = DOC[400H] & 1;
    } while (X != Y)

  o The results of the first iteration of the loop (the 21st and 22nd read cycles) should indicate the following:
    o Device should return a different value on D[0] in these two read cycles, regardless of the download duration or the delay from the download command to the polling sequence.

• After exiting the loop, the RAM contains the 1KB of requested data.

Ioreq Parameters

irHandle
  LSB 0-3: Socket number
  LSB 4-7: Partition number must be 0

irData
  Pointer to user buffer containing the IPL data

irLength
  Size of the user buffer. The IPL size limit is mode and memory window size dependent.

irFlags
  FL_IPL_MODE_NORMAL
  FL_IPL_VIRTUAL_RAM_MODE Use the virtual RAM mode
  FL_IPL_ALTERNATE_MAP Use the alternate paged RAM mode
FL_DOC_IPL_PAGED_RAM_MODE  Use the paged RAM mode.

FL_IPL_128K_WINDOW_MODE  Set mDOC H3 to 128KB memory window mode.

FL_IPL_NO_ADDRESS_SHIFT_MODE  Set mDOC H3 to no address shift mode.

FL_IPL_SWAP_BYTES_MODE  Set mDOC H3 to HW swap bytes mode.

irCount

Start sector offset of the IPL update. It is user's responsibility to set at least once a 0 offset so that the routine will erase the previous IPL code.

When writing IPL in segments, use buffers in 2KxN (N positive integer) chunks, in this case irCount must be a multiple of 4.

Returns

FLStatus

flOK on success

flFeatureNotSupported if the volume does not support this option

flHWProtection if the area is currently protected

Any other read/write failures

NO_IPL_CODE  must not be defined

FLCUSTOM.H definitions required
**flReadIPL**

**Function**
Reads the IPL to a user buffer.

This routine will read the IPL code/data directly from the device. This data will be available in the XIP area after next reboot.

Calling this routine with a partition number other than 0 returns an `flBadDriveHandle` error code.

**IOreq Parameters**
- **irHandle**
  - LSB 0-3: Socket number
  - LSB 4-7: Partition number must be 0
- **irData**
  - Pointer to user buffer that will contain the IPL data
- **irLength**
  - Size of the user buffer. The IPL buffer size:
- **irFlags**
  - Read modes:
    - `FL_IPL_MODE_NORMAL`
    - `FL_IPL_VIRTUAL_RAM_MODE`
    - `FL_IPL_ALERNATE_MAP`
    - `FL_DOC_IPL_PAGED_RAM_MODE`
    - `FL_IPL_128K_WINDOW_MODE`
    - `FL_IPL_NO_ADDRESS_SHIFT_MODE`
    - `FL_IPL_SWAP_BYTES_MODE`
- **irCount**
  - Start sector offset of the IPL read operation. `irCount` must be a multiple of 4 (or 0).

**Returns**
- **irFlags**
  - Returns the IPL mode
    - `FL_IPL_MODE_NORMAL`
    - `FL_IPL_VIRTUAL_RAM_MODE`
    - `FL_IPL_ALERNATE_MAP`
    - `FL_DOC_IPL_PAGED_RAM_MODE`
    - `FL_IPL_128K_WINDOW_MODE`
    - `FL_IPL_NO_ADDRESS_SHIFT_MODE`
    - `FL_IPL_SWAP_BYTES_MODE`
- **irCount**
  - In case IPL is not written – Returns zero.
  - In case IPL is written – Returns the maximum possible IPL size according to mDOC window and IPL type.

**FLStatus**
- `flOK` on success.

**FLCUSTOM.H** definitions required
- `NO_IPL_CODE` must not be defined
6.6 Special DOC Driver Functions

6.6.1 Global DOC Driver Calls

flInit

Function

Initializes the DOC Driver system, sockets, and timer for the specific partition.

This function is optional, and if not called, is activated automatically on the very first DOC Driver call. Calling this function after initialization has no effect unless flExit() was called.

Notes:
1. The routine actually tries to access the flash media.
2. This function does not conform to the standard DOC Driver function prototype. Its prototype is FLStatus flInit (void);

Parameters

None

Returns

FLStatus flOK on success, non-zero on failure

FLCUSTOM.H definitions required

None

flExit

Function

Last call before DOC Driver exits; dismounts all volumes and closes all files.

It is strongly recommended to call this function before shutting down the system. While this is not mandatory and DOC Driver will function correctly even if this is not done, calling flExit() frees all systems resources and may result in faster mounting the next time the system is turned on.

Note: This function does not conform to the standard DOC Driver function prototype. Its prototype is: void flExit(void);

Parameters

None

Returns

None

FLCUSTOM.H definitions required

FL_EXIT
flBuildGeometry

Function

Calculates the C/H/S representation of a specific number of virtual sectors.

This API can be used by applications that require a Cylinder/Head/ Sector geometry for the block device used (in contrast to just the number of sectors it export).

This calculation is just a recommendation for such a representation. Different systems may have different geometry limitations which may not be compatible with the proposed one.

The routine prototype is as follows:

```c
void NAMING_CONVENTION flBuildGeometry (FLDword capacity, FLDword * cylinder, FLDword * heads, FLDword sectors, FLBoolean oldFormat, FLWord wIrHandle)
```

Both routines use the following arguments:

**Input**

- `capacity` = Size of the media in bytes.
- `oldformat` = Should be set to FALSE.
- `wIrHandle` = Relevant partition handle (optional parameter).

**Output**

- `cylinder` = Number of cylinders.
- `heads` = Number of heads.
- `sectors` = Number of sectors.

Note: This function does not conform to the standard DOC Driver function prototype. Its prototype is: `void flExit(void).`
6.6.2 Low Level Information

flGetPhysicalInfo

Function

Returns information about the physical flash media.

A PhysicalInfo structure is returned to the caller’s buffer. The PhysicalInfo structure is defined as follows:

```c
typedef struct {
    FLWord type;          // Obsolete, always 0
    FLByte mediaType;     // Type of media (see below).
        FL_H3 15 mDOC H3
    FLDword dwUnitSize;   // Flash unit size in sectors.
    FLDword dwMediaSize;  // Media size (of unformatted device), in sectors.
    FLDword dwChipSize;   // Same as Media size
    FLDword interleaving; // Reserved, always 0
} PhysicalInfo;
```

Ioreq Parameters

irHandle

LSB 0-3: Socket number.
LSB 4-7: Partition number must be 0

irData

Address of buffer that receives PhysicalInfo structure

Returns

FLStatus

f1OK on success, non-zero on failure

irLength

mDOC H3 memory window base address

FLCUSTOM.H definitions required.
6.6.3 Internal Device Information

flGetExtendedDiskInfo

Function

This routine is used by internal msystems utilities, and as such, is subject to future changes.

The routine supplies very low-level information on the device and uses the
FLExtendedDiskInfo structure to retrieve the information.

Note: This routine receives the size of the struct in the dwStructSize field, allowing future revisions of this routine to report additional information.

The FLExtendedDiskInfo structure is defined in FLTL.H as follows:

```c
typedef struct {
    FLDword  dwStructSize       // INPUT - Size of the give structure – sizeof(dwStructSize)
    FLDword  dwTLType           // Obsolete, always 0
    FLDword  dwFormatFlags      // Obsolete, always 0
    FLDword  dwTrueFFSVersion   // Obsolete, always 0
    FLDword  dwFlashTechnology  // Obsolete, always 0
    FLDword  dwType             // Device type – FL_H3 (for mDOC H3 device)
    FLDword  dwSubType          // Obsolete, always 0
    FLByte   bMediaType         // Obsolete, always 0
    FLByte   bNoOfFloors        // Number of floors in the device – cascaded controllers.
    FLByte   bNoOfPlanes        // DOCH_NUM_OF_PLANES (by default 1)
    FLByte   bNoOfBanks         // DOCH_NUM_OF_BANKS (by default 1)
    FLByte   bDataBusWidth      // 16 bits
    FLByte   bSectorsPerPage    // DOCH_SECTORS_PER_PAGE (by default 4)
    FLByte   bSharedSectors     // DOCH_SHARED_SECTORS (by default 32)
    FLByte   bFastAreaSharedSectors
    FLByte   bMaxRelatedSectors // Obsolete, always 0
    FLDword  dwIPLSize          // IPL size (valid only after format)
    FLByte   bChangeableProtectedAreas // Number of changeable protection areas (valid only after format)
} FLExtendedDiskInfo;
```
FLByte btotalProtectedAreas  Total number of changeable protection areas (valid only after format)
FLDword dwUnitsInFirstFloor  Obsolete, always 0
FLDword dwUnitSize  Flash Unit size.
FLDword dwMaxBadPercentage  Obsolete, always 0
FLByte bNoOfBinaryPartitions  Obsolete
FLByte bNoOfDiskPartitions  Number of partitions on the media.
FLByte bBlockMultiplierBits  Obsolete, always 0
FLByte dwPercentUsed  Obsolete, always 0
FLByte bHeaderUnits  Obsolete, always 0
FLWord wHeaderLocation  Obsolete, always 0
FLByte bProgrammerNamePtr[4]  Programmer name

Fields that belong to the specific partition
FLDword dwVirtualUnits  Number of Flash Units in a partition.
FLDword dwFastUnits  Number of Flash Units in fast area of the partition
FLDword dwFirstQuickMountUnit  Obsolete
FLDword dwFirstUnit  Always 0
FLDword dwLastUnit  Same as dwVirtualUnits
FLDword dwSpareUnits  Always 0
FLDword dwTransferUnits  Always 0
FLDword dwPartitionFlags  See the Partition Flags list:

PROTECTABLE  0x1 Partition that can accept protection.
CHANGEABLE_PROTECTION  0x40 Partition that can change its protection attributes.

Fields that are valid only after mount
FLDword dwNormalAreaUnitSize  Flash Unit size
FLDword dwFastAreaLogicalUnitSize  Fast Flash Unit size
FLDword fastAreaVirtualFactor  Factor between size of fast Flash Unit and normal Flash Unit (power of 2)
FLByte bMinSectorsForFolding Always 0
FLByte bMinSectorsForNextWrite Always 0
FLByte bFastAreaMinSectorsForNextWrite Always 0
FLDword dwUsedUnits Always 0
FLDword dwFreeUnits Always 0
FLByte bNeededFreeUnits Always 0
FLDword dwUMDBBT Sector Obsolete
} FLExtendedDiskInfo;

Ioreq Parameters
irHandle LSB 0-3: Socket number
LSB 4-7: Partition number must be 0. The partition number to be used is supplied through the irCount field.
irFlags Must be set to FL_DISK_PARTITION.
irCount Partition number to be analyzed by the routine (1-15).
irLength Floor number to report for (0-3).
Use FL_ALL_FLOORS to report the information of the combined media header.
irData Pointer to user buffer containing the FLExtendedDiskInfo record.

Returns
FLStatus f1OK on success.
flBadDriveHandle if the partition number is not zero.
flFeatureNotSupported if the device does not support the routine.
flBadParameter if the given arguments are illegal.
6.7 Device Formatting

Logical format routines traditionally do not belong to the block device package. But due to ETFFS algorithm, a significant performance improvement can be achieved by formatting the FAT file system according to flash characteristics as page size and flash unit size. Therefore, DOC Driver block device package was extended with FAT format routines:

- `flFormatFS()` - Places FAT format on a specific logical partition.
- `flCreateLogicalPartitions()` - Creates, deletes and modifies logical partitions, and also enumerates the free space within the partitions (in other words, it provides functionality close to that provided by FDISK or similar disk-management utilities).

All format scenarios require affected logical partitions to be in the dismounted state, and remain in a dismounted state (subsequently requiring a mount call).
**flFormatFS**

**Function**

Performs a FAT high-level format of a single logical partition, using structure FATFormatParams as input:

**FATFormatParams**

- **FLWord noOfRootEntries**
  
  Number of root entries to be created; up to 65520 is valid (though time of search in such large directory may become an issue)
  
  Number 0 will create number of entries depending on the media size. Specifically, the number of root directory sectors will be 0.5% of the total sectors in logical partition, but no more than 15.
  
  For all but the smallest logical partitions, that means 240 root dir entries.

- **FLWord noOfSectrsPerCluster**
  
  Number of 512-byte sectors per FAT cluster. When setting this value to `FS_USE_DEFAULT_CLUSTER_SIZE` (0), DOC Driver will automatically determine the best size for your logical partition. Note that if `MIN_CLUSTER_SIZE` is defined, it will override the automatic selection.

- **FLByte noOfFATs**
  
  Number of FATs if FAT format will be executed (1 recommended)

- **FLByte ptType**
  
  Output field: upon successful completion, it is set to the correct type of logical partition (several different types correspond to FAT format):
  
  Either **FAT12_PARTIT, FAT16_PARTIT, DOS4_PARTIT** or **FAT32_PARTIT**

- **FLByte reserved1**
  
  0

- **FLByte reserved2**
  
  0

- **FLByte volumeID[4]**
  
  Partition identification number.

- **FLByte * volumeLabel**
  
  Volume label; If NULL, no label.
  
  If this pointer is not NULL, it should point to 11-byte ASCII array, not ASIIZ string.

**Notes:**

1. To place FAT format on a partition, the media must be partitioned first:
   
   - Low level or Block device level. This means that partition sizes were already set using the `flFlashFormat()` routine.
- Logical partition level. This means that the partition was already divided to logical drives usually by writing an MBR sector. This can be done using the `flCreateLogicalPartitions()` routine.

2. Optionally after FAT format is completed, the correct partition type (FAT12/FAT16) will be written in the MBR entry corresponding to that logical partition. You can disable this mode using the `FL_DO_NOT_UPDATE_MBR` flag.

3. It is therefore recommended to set cluster size to 0 and allow DOC Driver to select the optimal one in order to optimize the performance.

4. If the cluster size is set to 0, and FAT-12 support is not present, DOC Driver will attempt to decrease the cluster size on small partitions, as to remain within FAT-16 format; otherwise future attempt to mount such partition will fail.

5. If the cluster size is set to 0, and the partition is very large, DOC Driver will attempt to increase the cluster size to remain within FAT-16 format; otherwise the format procedure will fail.

6. `volumeID` is supposed to be a unique number; in a special case when all bytes of `volumeID` are zero, DOC Driver will create a number from current date and time (using user-supplied routines `flCurrentDate` and `flCurrentTime`).
Standard Formatting Parameters

DOC Driver contains standard definitions that can be used as a basic format example. This sample definition will format the logical partition with a maximum of 512 files in the root directory, let DOC Driver set the best cluster size automatically and use "MSYS" as the volume ID.

The following **STD_FAT_FORMAT_PARAMS** definitions are used to initialize the FATFormatParams record. The definition is coded in **FLFORMAT.H** with the following values:

- 512, noOfRootEntries
- **FS_USE_DEFAULT_CLUSTER_SIZE**, noOfSectrsPerCluster
- 1, noOfFATs
- 0, ptType
- 0, reserved1
- 0, reserved2
- \{0,0,0,0\}, volumeID[4]
- NULL, volumeLabel
The IOreq structure fields sent to flFormatFS()

**IOreq Parameters**

- **irHandle**
  - LSB 0-3: Socket number
  - LSB 4-7: Partition number
  - LSB 8-11: Logical number

- **irFlags**
  - **FL_MEDIA_WITHOUT_MBR** 0x1000
    - Assume the logical partition starts at sector 0 of the disk. This means that the disk will not have an MBR which is the common case for removable media.
  - **FL_MOUNT_ON_GIVEN_RANGE** 0x2000
    - Do not parse the MBR in order to locate start sector and length of the logical partition. Instead use the specific values supplied by the user through the `irSectorNo` and `irSectorCount` fields.
    - Logical partition number in this case is ignored.
  - **FL_DO_NOT_UPDATE_MBR** 0x4000
    - Do not mark the partition type (FAT16 or FAT12) in the MBR.

- **irSectorNo**
  - First absolute sector of logical partition (this field is valid only if `FL_MOUNT_ON_GIVEN_RANGE` is set).

- **irSectorCount**
  - Number of sectors in logical partition (this field is valid only if `FL_MOUNT_ON_GIVEN_RANGE` is set).

- **irData**
  - Address of the *FATFormatParams* structure.

**Returns**

- **FLStatus**
  - `flOK` on success, non-zero on failure.

**FSCUSTOM.H** definitions required

- `FS_FORMAT_VOLUME`
flCreateLogicalPartitions

Function

Creates, deletes and modifies logical partitions; also returns free space within a partition.

The routine accepts an array of LogicalPartitionParams structures in the irData field. The structure is defined in FLFORMAT.H file as follows:

LogicalPartitionParams

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLDword length</td>
<td>Length of the partition to be created.</td>
<td></td>
</tr>
<tr>
<td>FLWord lengthType</td>
<td>How the length parameter is interpreted</td>
<td>0</td>
</tr>
<tr>
<td>FLDword lengthType</td>
<td>depends on the lengthType field, and on the type and order of the partition (see below).</td>
<td></td>
</tr>
<tr>
<td>FLWord pType</td>
<td>Logical partition types.</td>
<td></td>
</tr>
<tr>
<td>FLDbyte flags</td>
<td>Logical partition types</td>
<td></td>
</tr>
<tr>
<td>FLDbyte flags</td>
<td>FL_MARK_PARTITION_BOOTABLE</td>
<td>0x80</td>
</tr>
</tbody>
</table>

The order in which partitions are described in an array of LogicalPartitionParams structures is important:

1. All logical partitions described before the extended partition, will be created as primary partitions (in MBR). Up to 3 primary partitions can exist if an extended partition is present, up to 4 otherwise.
2. Only primary partitions can be marked bootable (and only one of them).
3. If an extended partition is not present, the last primary partition will automatically take all the remaining free space on the media (its length parameter will be ignored).
4. Only one extended partition can be described. If present, it will automatically take all the remaining free space on the media (its length parameter will be ignored).
5. All logical partitions described after the extended partition, will be created as logical drives (in a chain of extended partitions). They cannot be marked as bootable. Their number is limited by the MAX_PARTITION_DEPTH definition, and does not have a hard limit.

Note: Any partition type can be given. If you are planning to format FAT partitions using flFormatFS, there is no need to specify the correct partition type (other than being different from extended partition type 5), it can be written by flFormatFS automatically.

**Standard Formatting Parameters**

DOC Driver contains standard definitions that can be used as a basic format example. This sample definition will format the media using a single logical partition on the entire media space reporting itself as a FAT16 media.

The following STD_LOG_PARTITION_PARAMS definitions can be used to initialize the LogicalPartitionParams record. The definitions are coded in FLFORMAT.H with the following values:

```
0,                  // Length
FL_LENGTH_IN_BYTES  // lengthType
DOS4_PARTIT,        // pType
FL_MARK_PARTITION_BOOTABLE // Flags
```

The IOreq structure fields sent to flCreateLogicalPartitions()

**IOreq Parameters**

- **irHandle**: LSB 0-3: Socket number, LSB 4-7: Partition number
- **irFlags**: 0 reserved
- **irData**: Pointer to array of LogicalPartitionParams structures
- **irCount**: Specify the number of LogicalPartitionParams structures

**Returns**

- **FLStatus**: flOK on success, non-zero on failure

**FSCUSTOM.H** definitions required: FS_CREATE_LOGICAL_PARTITIONS
**flFindLogicalPartition**

**Function**

Routine returns information about the location of the particular logical partition inside a partition. This API is only necessary if you want to use Abs Block I/O (which addresses the whole partition). File access and Log Block I/O address specific logical partitions automatically.

The routine uses the following structure:

**logPartitionInfo**

- **SectorNo bootSectorNo**
  Location of the boot sector within the partition.
- **SectorNo sectorsInVolume**
  Total number of sectors in logical partition.
- **SectorNo MBR**
  Location of MBR with the record of this logical partition. It is 0 for Primary partitions, but is non-zero for extended partitions.
- **int MBRslot**
  Number of Partition structures inside MBR.

The MBR information can be used to obtain the cylinder/head/sector information of the partition. Note that “starting sector” field for extended partitions is relative to the extended partition’s start.

**IOreq Parameters**

- **irHandle**
  LSB 0-3: Socket number.
  LSB 4-7: Partition number.
  LSB 8-11: Logical Partition number.

- **irData**
  logPartitionInfo structure.

**Returns**

- **FLStatus**
  f1OK on success, non-zero on failure.

**FSCUSTOM.H definitions required**

- FS_FIND_LOGICAL_PARTITION.
6.8 Accessing mDOC H3 Extended Functionality

In addition to the block device interface, the DOC Driver BD exports a set of features unique to mDOC H3 devices, such as power down mode (DPD), software and hardware protection, One Time Programming (OTP) area, and Unique ID (UID). These features (also referred to as extended functionalities) are exported in either of two ways:

- A DOC Driver routine: Best suited for customers using the DOC Driver BD SDK.
- The `fliOctl()` routine with a generic interface: Best suited for DOC Driver BD based drivers.

The best approach is to simply call the appropriate routine, however, compiled binary device drivers usually find the single function approach easier to implement. The reason is that most user accesses to the driver go through the native file-system API. The file system requires the device driver to implement read sector, write sector and general purpose device I/O control. The read sector and write sector are usually exported to the user through file-oriented calls, while the extended functionalities are available through the same single general-purpose routine.

Implementing the single routine enables driver implementers to be compatible with msystems drivers and use the same documentation and H files available with the DOC Driver SDK.

fliOctl Function

Function

Standard entry point and interface to all DOC Driver extended functions (I/O control).

`fliOctl()` has a generic interface that enables easy implementation, regardless of the extended function interface that the operating system defines.

A subset of the DOC Driver API is defined as an extended function interface. It is recommended to use `fliOctl()` with DOC Driver-based drivers in addition to the standard read block and write block calls.

`fliOctl()` receives the code number of the extended function to call, as defined in `FLIOCTL.H` and has the following structure:

```c
typedef struct {
    void *inputRecord;
    void *outputRecord;
} fliOctlRecord;
```

The fields of this structure (`inputRecord` and `outputRecord`) are specific to the extended function, and are defined separately for each function in the files `FLIOCTL.H`.

Since all msystems drivers are based on the DOC Driver SDK, the extended functions of the VxWorks driver are similar to the ones supplied by Windows Mobile.
The full DOC Driver SDK source supersedes the IOCTL interface, so when building a DOC Driver application (in contrast to building a device driver) there is no reason to use the more complex I/O control interface (*flIOctl()*) Simply call the SDK routines directly.

For more information on the extended functions interface and a list of the extended functions, refer to the *DOC Driver 1.0 Extended Functions Developer Guide*.

**ioreq Parameters**

- **irHandle**
  
  LSB 0-3: Drive number  
  LSB 4-7: Partition number

- **irFlags**
  
  Extended function code number

- **irData**
  
  Pointer to *flIOctlRecord*

**Returns**

- **FLStatus**
  
  *flOK* on success, non-zero on failure

**FLCUSTOM.H definitions required**

- **FL_IOCTL_INTERFACE**
  
  Note that some extended functions may require additional definitions. Refer to the description of each extended function for further details.
6.8.1 Hardware Read/Write Protection

The functions described in this section perform hardware read/write protection-related operations.

Method of Operation

mDOC devices, enable you to define partitions that are key protected (in hardware) against write operations, or against both read and write operations. Their size and protection attributes (read/write/changeable/lock/key) are predefined in the media formatting stage (DFORMAT utility or the `flFlashFormat()` function) in the following two ways:

- Defining up to one partition as changeable, meaning that its attributes are fully configurable (from read to write, change the key, change the lock state or any combination) at any time.
- Defining a partition as unchangeable, meaning that its attributes cannot be changed unless the media is reformatted.

mDOC H3 has an additional hardware safety measure. If the Lock option is enabled (using one of the extended functions), and the mDOC H3 LOCK pin is set, then the protected partition has an additional hardware lock preventing the use of the key, meaning that not even using the correct key will provide access to the protected partitions.

Each protected partition has its own unique attributes: Key, read\write protection and the hardware LOCK signal enable state (the safety chain). DOC Driver exports several routines that enable changing these attributes: Change key, change protection type (read\write protected) and change hardware LOCK state (enabled or not).

The only way to write or read from the corresponding protected partition is to use the insert key call (not even format will remove the protection). This is also true for modifying the protection attributes (key, read/write protection, and Lock Enable state). The key can be removed in one of the following configurations:

- Power down
- Removing of the protection key through the use of `flRemoveProtectionKey`

Some devices also support the Sticky Lock and/or the One Time Write (OTW) protection features:

- The Sticky Lock is a dedicated register accessed through DOC Driver API, which prevents the insertion of the key until the device receives a hardware reset signal. The bit is common to all the protected partitions on the device, but is enabled only if the partition enabled the LOCK property.
- OTW protection can be placed on an existing partition formatted as protectable. Once this property is placed in the partition, it will be write-protected forever, with no possibility of ever writing to it again.

Notes: 1. All hardware protection routines are applicable to both mounted and unmounted volumes.
2. Dismounting a volume does not remove its key.
3. If the partition is protected and the LOCK pin is asserted and enabled, there is no way to remove the protection by software (not even by the disable lock call). The mDOC H3 LOCK pin must be negated first.

4. If the partition is OTW–protected, it can never be un-protected, even when negating the LOCK pin.

5. In order to make a partition changeable, the specific flag must be added to the `protectionType` field of the format record.
f1IdentifyProtection

Function
Returns the protection state of a specific partition with the following information:
- Whether the partition can be protected and if so, if the protection attributes are changeable
- Protection type: None\read\write\both read and write
- Protection status: Key currently inserted or not inserted
- Hardware LOCK signal status: Enabled\disabled\asserted\not asserted

Ioreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irHandle</td>
<td>LSB 0-3: Socket number</td>
</tr>
<tr>
<td></td>
<td>LSB 4-7: Partition number</td>
</tr>
</tbody>
</table>

Returns

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>irFlags</td>
<td>PROTECTABLE</td>
</tr>
<tr>
<td></td>
<td>1: Must always be set for any other flags to have an effect.</td>
</tr>
<tr>
<td></td>
<td>READ_PROTECTED</td>
</tr>
<tr>
<td></td>
<td>2: The partition is protected against read operations.</td>
</tr>
<tr>
<td></td>
<td>WRITE_PROTECTED</td>
</tr>
<tr>
<td></td>
<td>4: The partition is protected against write operations.</td>
</tr>
<tr>
<td></td>
<td>LOCK_ENABLED</td>
</tr>
<tr>
<td></td>
<td>8: The hardware LOCK signal is enabled.</td>
</tr>
<tr>
<td></td>
<td>LOCK_ASSERTED</td>
</tr>
<tr>
<td></td>
<td>0x10: The hardware LOCK signal is currently asserted.</td>
</tr>
<tr>
<td></td>
<td>KEY_INSERTED</td>
</tr>
<tr>
<td></td>
<td>0x20: The protection is currently disabled using the key.</td>
</tr>
<tr>
<td></td>
<td>CHANGEABLE_PROTECTION</td>
</tr>
<tr>
<td></td>
<td>0x40: The protection type can be changed without reformatting the media.</td>
</tr>
<tr>
<td></td>
<td>OTW_PROTECTED</td>
</tr>
<tr>
<td></td>
<td>0x80: The key can never be inserted to this partition.</td>
</tr>
<tr>
<td></td>
<td>Note: This flag should be updated together with READ_PROTECTED and/or WRITE_PROTECTED flags</td>
</tr>
<tr>
<td></td>
<td>STICKY_LOCK_ASSERTED</td>
</tr>
<tr>
<td></td>
<td>0x100: The key cannot be inserted to this partition until the next hardware reset.</td>
</tr>
</tbody>
</table>

FLStatus

flOK on success non-zero on failure.
FLCUSTOM.H definitions required

HW_PROTECTION
**flInsertProtectionKey**

**Function**
The partition can be freely accessed while the key is inserted.

**Ioreq Parameters**
- `irHandle` LSB 0-3: Socket number
  LSB 4-7: Partition number
- `irData` Pointer to an 8-byte protection key

**Returns**
- `FLStatus`
  - `flOK` on success, non-zero on failure.

**FLCUSTOM.H definitions required**
- `HW_PROTECTION`

**flRemoveProtectionKey**

**Function**
Remove the protection key to reinstate protection.

**Ioreq Parameters**
- `irHandle` LSB 0-3: Socket number
  LSB 4-7: Partition number

**Returns**
- `FLStatus`
  - `flOK` on success
  - `flNotProtected` if the specified partition is not protected
  - `flFeatureNotSupported` if mDOC H3 does not support hardware protection

**FLCUSTOM.H definitions required**
- `HW_PROTECTION`
**flHardwareProtectionLock**

**Function**

Enables or disables the hardware LOCK signal. When the signal is asserted and enabled, it renders the protection key useless and the protection cannot be removed until the signal is negated. The Lock Enable property is also required for enabling the Sticky Lock mechanism.

**Caution**: If the hardware LOCK signal is asserted while it is enabled, the protection key is useless as long as the signal is on. In this case, you must negate the hardware LOCK signal, because it is impossible to disable it by software calls (not even by calling the format routine).

**Ioreq Parameters**

- **irHandle**
  LSB 0-3: Socket number
  LSB 4-7: Partition number

**Returns**

- **irFlags**
  LOCK_ENABLED
  8: If set (irFlags = 8), enable the LOCK signal.
  If not set, disable the LOCK signal.

- **FLStatus**
  flOK on success, non-zero on failure

**FLCUSTOM.H** definitions required

**HW_PROTECTION**

**flChangeProtectionKey**

**Function**

Modifies the current protection key of a partition to a new key.

**Note**: The key is revoked from the partition after the operation is completed, and the partition becomes protected (if protection attributes are Write or Read and Write protected).

**Ioreq Parameters**

- **IrHandle**
  LSB 0-3: Socket number
  LSB 4-7: Partition number

- **IrData**
  Pointer to the new 8-byte protection key

**Returns**

- **FLStatus**
  flOK on success, non-zero on failure

**FLCUSTOM.H** definitions required

**HW_PROTECTION**
**flChangeProtectionType**

**Function**

Modifies the current protection type (read, write or OTW) of a partition. This function can only be applied to partitions with the **CHANGEABLE** attribute set at format time.

Note: The key is revoked from the partition after the operation is completed, and the partition becomes protected (if protection attributes are Write or Read and Write protected).

**Ioreq Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| irHandle  | LSB 0-3: Socket number  
|           | LSB 4-7: Partition number |
| irFlags   | PROTECTABLE  
|           | 1: Must be added, otherwise the call fails.  
|           | READ_PROTECTED  
|           | 2: The partition is protected against read operations.  
|           | WRITE_PROTECTED  
|           | 4: The partition is protected against write operations.  
|           | OTW_PROTECTED  
|           | 128: This flag should only be set with the PROTECTABLE and WRITE_PROTECTED flags. |

**Returns**

<table>
<thead>
<tr>
<th>FLStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1OK</td>
<td>on success, non-zero on failure</td>
</tr>
</tbody>
</table>

**FLCUSTOM.H definitions required**

- HW_PROTECTION
**flApplyStickyLock**

**Function**
Sets the Sticky Lock bit.

**Notes:**
1. The sticky bit will affect only those partitions that enabled the LOCK property.
2. The sticky bit affects all the partitions of the media (provided the LOCK property is enabled).

**Ioreq Parameters**

- **irHandle**
  - LSB 0-3: Socket number

**Returns**

- **FLStatus**
  - `flOK` on success, non-zero on failure

**FLCUSTOM.H** definitions required

- `HW_PROTECTION`
6.8.2 OTP Operations

mDOC H3 devices have a ROM-like hardware feature known as a One Time Programming (OTP) area. The OTP area is a dedicated 15KB area on the flash that can be programmed once and then permanently locked (by mDOC H3 hardware). The function `flOTPWriteAndLock()` enables writing to the OTP section. This function writes the user buffer to the device and permanently locks the OTP area.

The function `flOTPSize()` returns the lock condition of the area (either locked or not), OTP total area size.

The code supporting this entire section can be removed by commenting out the `HW_OTP` compilation flag defined in `FLCUSTOM.H`.

Notes:
1. The OTP area can always be read, but writing to it is strictly limited. Once this area is written to, it is automatically locked in the hardware. In the locked state (after the first write), all erase and write operations to this area will be prevented by the device hardware. Not even a format operation can change the content of the OTP area.
2. OTP routines called with a partition other than 0 return an `flBadDriveHandle` error status even if the partition exists.
3. OTP write/read size must be a multiple of 512 bytes.

**flOTPSize**

**Function**

Returns maximum possible OTP size if any information written to OTP, otherwise – returns zero.

**Ioreq Parameters**

- `irHandle` LSB 0-3: Socket number
  LSb 4-7: Partition number must be 0

**Returns**

- `irFlags` `LOCKED_OTP` The area is locked.
- `irCount` The size of the OTP area, in bytes.
- `irLength` Returns maximum possible OTP size if any information written to OTP, otherwise – returns zero
- `FLStatus` `flOK` on success, non-zero on failure

**FLCUSTOM.H definitions required** `HW_OTP`
**fIOTPWriteAndLock**

**Function**

Writes data to the OTP area, and permanently locks the area. Writing always starts at the beginning of the OTP area.

**Caution:** Once locked, the OTP area is write-protected and cannot be rewritten, not even by formatting the entire physical drive.

Note: The area will be locked even if the parameter length is smaller than the actual OTP size. The remainder of the space is not used.

**Ioreq Parameters**

- `irHandle`  
  LSB 0-3: Socket number  
  LSB 4-7: Partition number must be 0
- `irData`  
  Pointer to user buffer.
- `irLength`  
  The size to write, in bytes.

**Returns**

- `FLStatus`  
  `flOK` on success, non-zero on failure

`FLCUSTOM.H` definitions required  
`HW_OTP`

**fIOTPRead**

**Function**

Reads data from the OTP area.

**Ioreq Parameters**

- `irHandle`  
  LSB 0-3: Socket number  
  LSB 4-7: Partition number must be 0
- `irData`  
  Pointer to user buffer.
- `irLength`  
  The size to read in bytes.
- `irCount`  
  Offset from the beginning of the OTP area, in bytes.

**Returns**

- `FLStatus`  
  `flOK` on success, non-zero on failure

`FLCUSTOM.H` definitions required  
`HW_OTP`
flGetUniqueID

Function

Returns the 16-byte unique ID (UID) data of mDOC H3.

Note: Calling this routine with a partition number other than 0 returns an flBadDriveHandle error code.

Ioreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| irHandle  | LSB 0-3: Socket number  
LSB 4-7: Partition number must be 0 |
| irData    | Pointer to a 16-byte user buffer. |

Returns

<table>
<thead>
<tr>
<th>FLStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flOK</td>
<td>on success, non-zero on failure</td>
</tr>
</tbody>
</table>

FLCUSTOM.H definitions required

HW_OTP

flGetCustomerID

Function

Returns the customers’ 4-byte ID data.

Note: Calling this routine with a partition number other than 0 returns an flBadDriveHandle error code.

Ioreq Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| irHandle  | LSB 0-3: Socket number  
LSB 4-7: Partition number must be 0 |
| irData    | Pointer to a 4-byte user buffer. |

Returns

<table>
<thead>
<tr>
<th>FLStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flOK</td>
<td>on success, non-zero on failure</td>
</tr>
</tbody>
</table>

FLCUSTOM.H definitions required

HW_OTP
6.8.3 Power Saving Mode

flDeepPowerDownMode

Function

This function places the device in and out of Deep Power-Down mode.

While in this mode, mDOC H3 requires a much lower current, but as a result all input and output
buffers are disabled. This means that the mDOC H3 registers are not connected to the mDOC H3
logic.

Notes:
1. mDOC H3 wakes up in an Active power mode (either Turbo mode or PowerSave
   mode).
2. Calling this routine with a partition number other than 0 returns an
   flBadDriveHandle error code.
3. ETFFS supports by default automatic enter and exit of DPD mode, to further reduce
   the power consumption of the device. (See section 8.3 for more details)

IOreq Parameters

irHandle

LSB 0-3: Socket number
LSB 4-7: Partition number must be 0

irFlags

DEEP_POWER_DOWN
Forces the device into low-power consumption mode. If
this flag is not set the device returns to Normal mode.
EXIT_DEEP_POWER_DOWN (or any other value) will exit
from Deep Power Down mode.

Returns

FLStatus
flOK if the volume supports such a feature, otherwise
flFeatureNotSupported

FLCUSTOM.H definitions required
6.8.4 Recovery from a Reset Assertion or Power Off

**flRecoverFromPowerLoss**

**Function**

This routine performs the necessary device initialization sequences after power up or reset signal assertion. Normally this routine is not needed as ETFFS automatically initializes the mDOC H3 device. This API was added, for full hibernation of the device.

Note: This routine must be called after the maximal power up timing spec of the mDOC H3 has elapsed.

**IOreq Parameters**

- irHandle
  - LSB 0-3: Socket number
  - LSB 4-7: Partition number must be 0

- irFlags
  - FL_PREPARE_FOR_RESUME

**Returns**

- FLStatus
  - **flOK** if the volume supports such a feature, otherwise **flFeatureNotSupported**
6.8.5 HW Configuration API

flHwConfig

Function

This routine enables the control of mDOC H3 HW features.

Among these features are also features related to IRQ signal behavior, DMARQ# signal behavior etc.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irHandle</td>
<td>LSB 0-3: Socket number</td>
</tr>
<tr>
<td>irFlags</td>
<td>Pointer to word describing HW configuration Type.</td>
</tr>
</tbody>
</table>

Allowed types are:

- **FL_IRQ_RB_TYPE (0)**: Configures IRQ signal behavior in case of interrupt when mDOC H3 has completed long accesses to the flash, like program or erase.
- **FL_DMA_TYPE (5)**: Describes the behavior of IRQ# and DMA_REQ# while using HW DMA transactions.

irFlags are different for each HW configuration type chosen as described below:

- **FL_IRQ_RB_TYPE**
  - **FL_INT_RB_ENABLED (1)**: IRQ is enabled when mDOC H3 finishes long transaction.
  - **FL_INT_RB_DISABLED (0)**: IRQ is disabled when mDOC H3 finishes long transaction.

Note: mDOC H3 supports only level interrupts.

- **FL_DMA_TYPE**
  - **FL_DMA_HW_ENABLED (1)**: Enable HW DMA.
  - **FL_DMA_HW_DISABLED (0)**: Disable HW DMA.
  - **FL_DMA_REQ_EDGE (2)**: DMA_REQ# output pin is edge type.
  - **FL_DMA_REQ_LEVEL (0)**: DMA_REQ# output pin is level type.
  - **FL_NEGATED_1_ASSERTED_0 (0)**: DMA_REQ# is normally logic 1 and falls to initiate DMA.
  - **FL_NEGATED_0_ASSERTED_1 (4)**: DMA_REQ# is normally logic 0 and rises to initiate DMA.

Returns **flOK** on success
FLStatus

*FlFeatureNotSupported*: In case feature is not supported for device.
7. **DOC Driver Status Codes**

All DOC Driver calls return a status code of type `FLStatus`, having a numerical value that is zero or positive. These status codes are defined as an enumerated type in `FLCOMMON.H`.

A `flOK` (value ‘0’) status indicates success. Any other value (always positive) indicates an abnormal condition or a failure to execute the requested function.

Table 13 describes the DOC Driver status codes, which are listed in alphabetical order.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flAdapterNotFound</code></td>
<td>During initialization of the socket interface, the socket hardware was not detected or it failed to initialize correctly. Receiving this status means that further DOC Driver calls are impossible. The <code>noOfSockets</code> global variable reports the number of sockets actually found.</td>
</tr>
<tr>
<td><code>flBadDriveHandle</code></td>
<td>The <code>ioreq_irHandle</code> field of a DOC Driver call specified a drive number, but the number is invalid. Either the socket was not found, that the partition does not exist or the specific operation must be performed on partition number 0.</td>
</tr>
<tr>
<td><code>flBadFormat</code></td>
<td>Problems found in the physical format of the flash media volume. The status indicates the following:</td>
</tr>
<tr>
<td><code>flBadFunction</code></td>
<td>Non-existent DOC Driver function was called. This can occur if the code for calling the function is of the form: <code>bdCall(&lt;function code&gt;, &lt;ioreq&gt;)</code> where <code>&lt;function code&gt;</code> is not a valid DOC Driver function.</td>
</tr>
<tr>
<td><code>flBadParameter</code></td>
<td>The DOC Driver function was called with an illegal parameter value.</td>
</tr>
<tr>
<td><code>flBufferingError</code></td>
<td>The FAT filter could not use its internal buffers.</td>
</tr>
<tr>
<td><code>flDataError</code></td>
<td>Uncorrectable error occurred while reading from mDOC H3.</td>
</tr>
<tr>
<td><code>flDriveNotAvailable</code></td>
<td>Call to DOC Driver function refers to a drive that is already busy with another DOC Driver operation on a different execution thread. The call cannot take place until the original operation terminates.</td>
</tr>
<tr>
<td><code>flFeatureNotSupported</code></td>
<td>The specific software or hardware does not support the requested functionality.</td>
</tr>
<tr>
<td><code>flFormatNotSupported</code></td>
<td>DOC Driver cannot mount the volume because its FAT formatting is not supported by the current DOC Driver system. One of the following occurred:</td>
</tr>
<tr>
<td></td>
<td>The sector size of the mounted volume is not the same as the DOC Driver sector size selected in <code>FSCUSTOM.H</code>.</td>
</tr>
<tr>
<td></td>
<td>The volume uses 12-bit FAT, but 12-bit FAT support was not enabled.</td>
</tr>
<tr>
<td><code>flGeneralFailure</code></td>
<td>The status may be reported upon detection of inconsistent state of the device</td>
</tr>
<tr>
<td>Status Code</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| flHWProtection      | It is not possible to read/write from the flash media because hardware protection is enabled in the following areas:  
                           Read/write operations to a read-protected partition  
                           Read operation to the data protection structures  
                           Write operation of the IPL while the protection keys are not inserted  
                           Write/erase operation to a write-protected partition  
                           Write operation to the data protection structures  
                           Change protection attributes without inserting the key  
                           Write/erase operation to the OTP area                  |
| flNonFATformat      | mDOC H3 media contains the proper disk format, but does not have a valid FAT format. This status code is informatory, and the media is mounted. |
| flNoSpaceInVolume   | A file or directory needs to be extended, but there is no available space on the volume. |
| flNotEnoughMemory   | This status may be returned by `flDefragmentVolume()`                         |
| flNotMounted        | A DOC Driver function accessed a partition, but the volume is not mounted. A mount-volume call is required before accessing the drive. One of the following occurred:  
                           An `flAbsMountVolume()` call was never issued for this partition.  
                           The partition that was previously mounted was removed, replaced, reinserted, or dismounted by a DOC Driver call. |
| flNotProtected      | Trying to change the protection attributes of an unprotectable partition.     |
| flNoWriteAccess     | DOC Driver call tried to erase a non-mounted partition.                      |
| flPartitionNotFound | The FAT filter failed to find such a partition in the MBR (Master Boot Record). |
| flSectorNotFound    | The DOC Driver was directed to read or write a sector outside the valid range of the volume. This error is an internal consistency problem, and probably indicates either damage to the FAT format or a device malfunction. |
| flUnchangeableProtection | Trying to change the protection attributes of a volume formatted with unchangeable protection. |
| flUnknownMedia      | Returned by `flAbsMountVolume()` when mounting a disk partition that does not contain the proper physical format. Usually indicates that the volume is unformatted, but may also be caused by malfunction of the device. |
| flWriteFault        | Physical error occurred while writing to mDOC H3.                           |
| flWriteProtect      | It is not possible to write to flash media, because of the following reasons:  
                           Write protection was previously set.  
                           There is no spare flash unit on media. |
| fFlWrongKey         | The wrong key was inserted by `flInsertProtectionKey()`                     |
8. Extended Information

8.1 Hardware Read/Write Protection

mDOC H3 devices enable up to 14 partitions to be hardware-protected against write and/or read operations.

A partition can be either write protected or read and write protected (but it cannot be read protected only).

Defining partition size and protection attributes (read/write/changeable/lock) is done at the media formatting stage, using the DFORMAT utility or format extended function call. You may define partitions as changeable, meaning that the password and attributes are fully configurable (read/write, both, none and vice versa) at any time. Note that unchangeable partition attributes (including the protection key) cannot be changed unless the media is reformatted.

To access a protected partition, you must insert the password (and, in practice, you must use the insertProtectionKey routine). Once the key is entered, the partition can be freely accessed (as if it was not protected).

Warning: If the key is lost, protection cannot be removed, not even by formatting the media.

Sticky Lock

mDOC H3 devices have an additional protection feature called the Sticky Lock. Once this feature is enabled (using the flApplyStickyLock () API), the protection key cannot be inserted until the device receives a hardware reset signal. Once this mode is activated, it affects all protected partitions on the device provided the LOCK mechanism (LOCK_ENABLED) has been enabled.

OTW Protection

OTW is a special write-protection mode. This attribute can be placed on an existing partition formatted as a protectable partition. Once a partition has an OTW-protected attribute, it is no longer possible to write to this partition (even using a protection key), and actually becomes a ROM-like partition.

Warning: If a partition becomes OTW-protected, it is no longer possible to reformat the device.

Hardware LOCK Signal

A hardware signal can disable the use of the key for removal of the partition’s protection. Although this is a hardware signal, it can be enabled and disabled per partition by the function flHardwareProtectionLock(). Note that the LOCK_ENABLED flag is protected by the same key as the partition is protects.

Warning: If the hardware LOCK is asserted, and the function flHardwareProtectionLock() enabled the LOCK signal, it cannot be disabled by software, even by the use of the key. In this condition, mDOC H3 can no longer be formatted (until the signal is negated).
8.2 Security-Enabling Features

mDOC H3 devices offer the following hardware features that can be used in security applications:

- Unique device and customer IDs, burned into each device, making each mDOC H3 unique and identifiable by your application.
- A 15KB ROM-like One Time Programming (OTP) area. You can write information to this area once, and then it automatically locks forever, making the information stored on it available for read only (much like ROM).

8.3 Power Modes

1. mDOC H3 has 2 basic states – Active and inactive. Active mode may be configured to be ‘Turbo’ or ‘PowerSave’. Inactive mode may be configured to ‘Standby’ or ‘DPD’.
2. mDOC H3 may be configured to work in different modes:
   - mDOC H3 always stays at the Active mode and never enters Inactive.
   - mDOC H3 stays in Active mode while executing command and then switches immediately to 'Standby' mode. After user defined timeout mDOC H3 switches to DPD mode.
   - mDOC H3 stays in active mode while executing command and then switches to Inactive mode (either ‘Standby’ or ‘DPD’ depending on user definition) after user defined timeout.
3. Configuration may be done by compilation flags or by settings environment variables:
## Table 14: Active/Inactive Flags and Values

<table>
<thead>
<tr>
<th>Environment variable/Macro Definition</th>
<th>Possible Values and Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL_SET_ACTIVE_DPD_MODE /</td>
<td>DOCH_WM_NORMAL – mDOC H3 works in Turbo mode and never enters inactive mode.</td>
</tr>
<tr>
<td>DOCH_DPD_DEFAULT_ACTIVE_MODE</td>
<td>DOCH_WM_LOW_FREQ – mDOC H3 works in PowerSave mode and never enters inactive mode.</td>
</tr>
<tr>
<td></td>
<td>DOCH_WM_NORMAL_AND_AUTO_STBY – mDOC H3 device works in Turbo mode when executing a command and enters inactive mode after timeout set by FL_SET_TIMEOUT_DPD / DOCH_DPD_DEFAULT_DPD_TIMEOUT has elapsed.</td>
</tr>
<tr>
<td></td>
<td>DOCH_WM_LOW_FREQ_AND_AUTO_STBY – mDOC H3 works in PowerSave mode when executing a command and enters inactive mode after timeout is set by FL_SET_TIMEOUT_DPD / DOCH_DPD_DEFAULT_DPD_TIMEOUT has elapsed.</td>
</tr>
<tr>
<td>FL_SET_INACTIVE_DPD_MODE /</td>
<td>DOCH_IM_IDLE – This value sets the inactive mode to Standby mode.</td>
</tr>
<tr>
<td>DOCH_DPD_DEFAULT_INACTIVE_MODE</td>
<td>DOCH_IM_DPD – This value sets the inactive mode to DPD mode.</td>
</tr>
<tr>
<td></td>
<td>DOCH_IM_IDLE_2_DPD – When this flag is set, mDOC H3 switches into Standby mode right after any operation has completed. After a timeout defined by FL_SET_TIMEOUT_DPD during which no access to mDOC H3 is made, mDOC H3 switches into DPD mode. Setting this value overrides FL_SET_ACTIVE_DPD_MODE for the inactive mode, while leaving the active mode settings as defined in FL_SET_ACTIVE_DPD_MODE.</td>
</tr>
<tr>
<td>FL_SET_TIMEOUT_DPD /</td>
<td>Timeout in miliseconds from Active to Inactive mode or from Standby to DPD mode, rounded up to the nearest multiple of 100 milliseconds.</td>
</tr>
<tr>
<td>DOCH_DPD_DEFAULT_DPD_TIMEOUT</td>
<td>Maximum timeout value is 10 seconds (10,000 milliseconds).</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>For values 0-99 -&gt; Actual time-out is 100 ms</td>
</tr>
<tr>
<td>For values 100-199 -&gt; Actual time-out is 200 ms</td>
<td></td>
</tr>
<tr>
<td>For values 200-299 -&gt; Actual time-out is 300 ms</td>
<td></td>
</tr>
<tr>
<td>Maximum timeout value is 10 seconds (notified in milliseconds).</td>
<td></td>
</tr>
</tbody>
</table>
### Table 15: Active/Inactive Combinations and Timeouts

<table>
<thead>
<tr>
<th>Active /Inactive Modes</th>
<th>DOCH_WM_NORMAL</th>
<th>DOCH_WM_LO_W_FREQ</th>
<th>DOCH_WM_NORMAL_AND_AUTO_STBY</th>
<th>DOCH_WM_LOW_F_REQ_AND_AUTO_STBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCH_IM_IDLE</td>
<td>Device will not move from active (Turbo) to inactive mode automatically.</td>
<td>Device will not move from active (PowerSave) to inactive mode automatically.</td>
<td>Device will move from active (Turbo) to inactive (Standby) mode automatically after user defined timeout in milliseconds.</td>
<td>Device will move from active (PowerSave) to inactive (Standby) mode automatically after user defined timeout in milliseconds.</td>
</tr>
<tr>
<td>DOCH_IM_DPD</td>
<td>Device will not move from active (Turbo) to inactive mode automatically.</td>
<td>Device will not move from active (PowerSave) to inactive mode automatically.</td>
<td>Device will move from active (Turbo) to inactive (DPD) mode automatically after user defined timeout in milliseconds.</td>
<td>Device will move from active (PowerSave) to inactive (DPD) mode automatically after user defined timeout in milliseconds.</td>
</tr>
<tr>
<td>DOCH_IM_IDLE_2_DPD</td>
<td>Device will move from active (Turbo) to Standby mode automatically and immediately after end of last command, and then to DPD mode after user defined timeout in milliseconds, rounded up to nearest multiple of 100 milliseconds.</td>
<td>Device will move from active (PowerSave) to Standby mode automatically immediately after end of last command, and then to DPD mode after user defined timeout in milliseconds, rounded up to nearest multiple of 100 milliseconds.</td>
<td>Device will move from active (Turbo) to Standby mode automatically immediately after end of last command, and then to DPD mode after user defined timeout in milliseconds, rounded up to nearest multiple of 100 milliseconds.</td>
<td>Device will move from active (PowerSave) to Standby mode automatically immediately after end of last command, and then to DPD mode after user defined timeout in milliseconds, rounded up to nearest multiple of 100 milliseconds.</td>
</tr>
</tbody>
</table>
9. ADDITIONAL INFORMATION AND TOOLS

Additional information about mDOC H3, including application notes, can be found at http://www.m-sys.com.

Additional tools and documents are listed in the table below:

Table 16: Additional Documents

<table>
<thead>
<tr>
<th>Document/Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Sheets</td>
<td>Various mDOC H3 products</td>
</tr>
<tr>
<td>Developer Guide</td>
<td>DOC Driver 1.0 Extended Functions Developer Guide</td>
</tr>
<tr>
<td>Developer Guide</td>
<td>DOC Driver 1.0 Software Utilities Developer Guide</td>
</tr>
</tbody>
</table>
HOW TO CONTACT US

USA
msystems, Inc.
555 North Mathilda Avenue, Suite 220
Sunnyvale, CA 94085
Phone: +1-408-470-4440
Fax: +1-408-470-4470

China
msystems China Ltd.
Room 121-122
Bldg. 2, International Commerce & Exhibition Ctr.
Hong Hua Rd.
Futian Free Trade Zone
Shenzhen, China
Phone: +86-755-8348-5218
Fax: +86-755-8348-5418

Japan
msystems Japan Inc.
Asahi Seimei Gotanda Bldg., 3F
5-25-16 Higashi-Gotanda
Shinagawa-ku Tokyo, 141-0022
Phone: +81-3-5423-8101
Fax: +81-3-5423-8102

Europe
msystems Ltd.
7 Atir Yeda St.
Kfar Saba 44425, Israel
Tel: +972-9-764-5000
Fax: +972-3-548-8666

Taiwan
msystems Asia Ltd.
14 F, No. 6, Sec. 3
Minquan East Road
Taipei, Taiwan, 104
Tel: +886-2-2515-2522
Fax: +886-2-2515-2295

Internet
http://www.m-systems.com/mobile

General Information
info@m-systems.com

Sales and Technical Information
technical@m-systems.com

This document is for information use only and is subject to change without prior notice. msystems Ltd. assumes no responsibility for any errors that may appear in this document.

msystems products are not warranted to operate without failure. Accordingly, in any use of the Product in life support systems or other applications where failure could cause injury or loss of life, the Product should only be incorporated in systems designed with appropriate and sufficient redundancy or backup features.

Contact your local msystems sales office or distributor, or visit our website at www.m-systems.com to obtain the latest specifications before placing your order.

© 2006 msystems Ltd. All rights reserved.

msystems, mDOC H3, mDOC H3 Millennium, DiskOnKey, DiskOnKey MyKey, FFD, Fly-By, imDOC H3, iDOC, mmDOC H3, mDOC H3, Mobile mDOC H3, Smart DiskOnKey, SmartCaps, SuperMAP, TrueFFS, umDOC H3, uDOC, and Xkey are trademarks or registered trademarks of msystems Ltd. Other product names or service marks mentioned herein may be trademarks or registered trademarks of their respective owners and are hereby acknowledged. All specifications are subject to change without prior notice.