Application Note
AP-DOC-046

Extended Functions of DiskOnChip Driver Based on TrueFFS® Version 5.0

JULY-2001

91-SR-005-11-7L REV 2.0

Contents

1 Introduction

The basic function of TrueFFS is to provide disk emulation using the DiskOnChip. To do this, TrueFFS provides a standard block-device interface, consisting in essence of the capability for reading and writing logical sectors. This capability is enough to enable file-systems and operating systems to access the DiskOnChip as a storage device.

In addition to the standard storage device functionality, the TrueFFS driver, based on TrueFFS SDK, provides extended functionalities. These functionalities go beyond simple data storage capabilities and include features like: format the media, Read/Write protect, Binary partitions access, Flash defragmentation and other options. All these unique functionalities are available in all TrueFFS based drivers through the standard IO control command of the native file system.

In many operating systems the IOCTL mechanism is the only way you may access a driver running in kernel mode from your applications running in user mode.

All TrueFFS based drivers have the same internal extended functionalities mechanism. However various OS constrains force several variations to the external interface. This document describes the core interface on which specific appendixes supply the variations. These appendixes are included with the installation manuals (or readme files) of the OS specific drivers.

1.1 Glossary

Definition	Description
Socket	Socket is a physical location where a DiskOnChip device can reside.
Physical Drive\Physical Device	Physical drive is a DiskOnChip device placed in a socket.
Partition\Volume	Partition is a part of a physical drive handled as an independent unit. A partition can be either a BDTL (Block Device Translation Layer) partition (i.e. a logical drive partition) or a Binary partition. A physical device can contain up to 4 partitions of any type, provided one of them is a BDTL partition.
Logical Drive \ BDTL Partition \ BDTL Volume	A BDTL (Block Device Translation Layers) partition is a partition formatted and supported by one of TrueFFS' Translation Layers, making it capable of supporting a block device driver and file-system. This partition is accessed by your OS file system through the TrueFFS driver, and is also called the flash disk partition (in simple terms – it is the partition used as a disk).
Binary Partition \ Binary Volume	Partition on the DiskOnChip that usually contains executable code (usually OS loader or boot code). These partitions are not accessed through the file system regular calls, but through its device IO controls.
TL	Translation Layer

Definition	Description
Binary Sub-Partition	All of the Binary Partition blocks are marked with a unique signature. Since TrueFFS-SDK (OSAK) 4.1 a dedicated routine enables changing this signature for a contiguous subset of the partition's blocks, thus creating several separated areas within the Binary Partition. Each such area is called a Binary sub-partition. When first formatted the Binary Partition contain a single sub-partition.
Firmware Space \ EXB Space	M-System provides an EXB file that containing a driver for x86 BIOS platforms. The file can be placed on the DiskOnChip using the Dformat DOS utility or through one of the driver extended functions. Once placed on the media the DiskOnChip will automatically hook int13 as a BIOS expansion and will register as a normal FAT hard-drive.
IPL	Initial Program Loader – This code usually performs minimal system initialization and loads the SPL (see below).
SPL	Secondary Program Loader – This code loads and runs the code found in the first binary partition of the DiskOnChip (the default SPL, intended for x86, loads the TrueFFS BIOS driver).
Quick mount	DiskOnChip Millennium Plus and DiskOnChip 2000 TSOP can be formatted with the quick mount feature. This option saves the RAM conversion table to the flash before dismounting therefore allowing quicker mount time on the next power up. The tables are protected by EDC and a "dirty flag" mechanism, making sure the media is not mounted with bad conversion tables. The downside is slower dismount and the loss of some media space (usually one block per BDTL partition). This feature is usually not needed unless mounting time is critical and the DiskOnChip device is large.
OTP area (or block)	One Time Programming Area – A 6KB ROM-Like block that automatically locks up forever after one write operation is performed in it.

2 Extended Functionality Interface

The TrueFFS driver implements the interface of the extended functionalities through your file system device IO control calls. The device IO control interface can vary from one file system to another, but the general interface has the following 2 steps:

1) Getting a the TrueFFS driver descriptor:

   Drive Handle = File System Get Drive Handle Call (Device Driver Name & Parameters);

2) Initiating an Extended function call:

   File System Call Status = File System Io Control Call (Driver Handle , IO Request Packet);

The first step (Getting the **Drive Handle**) is file system dependent. Its result is a descriptor of the TrueFFS driver that can be used by the device IO control call.

The second step (Initiating an Extended function call) has several attributes common to all File Systems:

File System Call Status – This value indicates whether the call was successfully passed to the driver and whether the driver took responsibility for the call. It usually does not specify the operation status. The operation status is returned as part of the IO *Request Packet* using standard TrueFFS status codes. The complete set of the TrueFFS status codes is available in the IO control H file (flIOCTL.H, included with each driver package).

File System Io Control Call – The function name used to invoke a file system extended functionality (device IO control) call.

IO Request Packet – All TrueFFS extended functionalities receive the following IO request packet:

```
typedef struct {
     FLHandle            irHandle;
     unsigned            irFlags;
     FLSimplePath FAR1 * irPath;             /* Not used */
     void FAR1         * irData;
     long                irLength;           /* Not used */
     long                irCount;            /* Not used */
} Ioreq
```

- **irHandle** - A handle identifying the Partition on which an operation should be performed. When there is only one logical drive no ambiguity can occur, and the drive handle parameter should be 0. This drive handle is composed of the physical drive number (Bits 0-3) and the partition number (Bits 4-7).
    - As both the Binary Partitions and the BDTL partitions are numbered from 0, a Binary Partition can have the same handle as a BDTL partition on the same physical drive. Therefore, Binary and BDTL operation are always handled using different calls, thus avoiding ambiguity.
- **irFlags –** This field controls the type of your extended function. Every extended function is represented by a code defined as enumerated type defined in `FLIOCTL.H`.

  typedef enum{ FL_IOCTL_GET_INFO = FL_IOCTL_START,

```
    FL_IOCTL_DEFRAGMENT,
    FL_IOCTL_WRITE_PROTECT,
    FL_IOCTL_MOUNT_VOLUME,
    FL_IOCTL_FORMAT_VOLUME,
    FL_IOCTL_BDK_OPERATION,
    FL_IOCTL_DELETE_SECTORS,
    FL_IOCTL_READ_SECTORS,          /* Not implemented */
    FL_IOCTL_WRITE_SECTORS,         /* Not implemented */
    FL_IOCTL_FORMAT_PHYSICAL_DRIVE,
    FL_IOCTL_FORMAT_LOGICAL_DRIVE,  /* Not implemented */
    FL_IOCTL_BDTL_HW_PROTECTION,
    FL_IOCTL_BINARY_HW_PROTECTION,
    FL_IOCTL_OTP,
    FL_IOCTL_CUSTOMER_ID,
    FL_IOCTL_UNIQUE_ID,
    FL_IOCTL_NUMBER_OF_PARTITIONS,
    FL_IOCTL_INQUIRE_CAPABILITIES,
    FL_IOCTL_SET_ENVIRONMENT_VARIABLES,
    FL_IOCTL_PLACE_EXB_BY_BUFFER,
    FL_IOCTL_WRITE_IPL,
    FL_IOCTL_DEEP_POWER_DOWN_MODE
} flIOctlFunctionNo;
```

The constant *FL_IOCTL_START* defines the number of the first extended function code to be used by TrueFFS. Typically, each operating system defines a range of extended function codes that are reserved for its use. *FL_IOCTL_START* is therefore defined outside this range (see the OS driver installation manual for more info).

- **irData –** This field should contain a pointer to an flIOctlRecord record. The flIOctlRecord record contains pointers to your specific extended function input and output records:

```
typedef struct {
  void FAR1 *inputRecord;
  void FAR1 *outputRecord;
} flIOctlRecord;
```

Section 3 of this document describes in details the specific input and output records of each extended function.

2.1 Example Source Code

Calling extended functions depends on the operating system, and is described in detail in the relevant Application Note for each specific TrueFFS driver.

2.1.2 Windows CE Code Example

The following example shows the code invoking a write-protect extended function under Windows CE.

Note that CE requires separated input and output buffers and the exact control function number as specific arguments of the “Device IO Control”. This interface is a bit different from the one described above.

The **IO Request Packet** is not a part of the I/O control call. It is replaced by:

- The **inputRecord** structure pointer.
- The **outputRecord** structure pointer.
- The **irFlags** field.

The **irHandle** is supplied as an argument to the binding stage (getting the drive handle).

```
# include <stdio.h>
# include “flioctl.h”

unsigned realBytes;
HANDLE hDriver;
flWriteProtectInput     inputRecord;
flOutputStatusRecord    outputRecord;

/* Binding the DiskOnChip driver */
hDriver = CreateFile(TEXT("DSK1:"),
            GENERIC_READ|GENERIC_WRITE,
            0,
            NULL,
            OPEN_EXISTING,
            0,
            NULL);

/* Preparing the input buffer */
strcpy(inputRecord.Key, “AAAAAAAA”);       /* Write-protect Key */
inputRecord.type  = FL_PROTECT;            /* Type of write protect command */

/* Activating the write protect function */
```

```
if(DeviceIoControl(     hDriver ,                       /* Driver handle */
                  FL_IOCTL_WRITE_PROTECT   /* Extended function code */
                  &inputRecord,                         /* Input buffer  */
                  sizeof(inputRecord),                  /* Input buffer size */
                  &outputRecord,                 /* Output buffer */
                  sizeof( outputRecord),                /* Output buffer size */
                  & realBytes,                          /* Return buffer size */
                  NULL) == FALSE)         /* Overlapped handle not needed */
{
           /* Failure – Print the return status */
           printf("Failed Write-Protect. Status = %d\n", outputRecord.status);
}
```

TrueFFS driver data structures are defined in `FLIOCTL.H` and described in the section below.

3 Extended Functions Argument Structures

This section describes each of the extended functions purpose, usage and relevant I/O structures. As explained in Section 2, all extended function receive a common flIOctlRecord pointing to the respective input and output records:

```
typedef struct {
        void FAR1 *inputRecord;
        void FAR1 *outputRecord;
} flIOctlRecord;
```

The status of the IO control call is returned in the output records of the TrueFFS driver (FLStatus field). FlOK (0) indicates success while any other status indicates some kind of a failure.

Note: File System Call Status indicates whether the call was successfully passed to the driver and whether the driver took responsibility for the call. It does not specify the operation status. The operation status is one of the *outputRecord* fields using standard TrueFFS status codes.

3.1 FL_IOCTL_GET_INFO

Returns general information on the specific BDTL partition, the DiskOnChip socket address, software version, high-level and low-level geometry and estimated lifetime of the media.

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

The *VolumeInfoRecord* structure is defined as:

```
typedef struct {
unsigned long logicalSectors;       /* number of logical sectors in the BDTL partition (including
                                       hidden sectors and boot sectors  */
unsigned long bootAreaSize;  /* boot area size of the entire drive (combines all binary partitions) */
unsigned long baseAddress;          /* physical base address of the memory window */
unsigned short flashType;           /* JEDEC id of the flash              */
unsigned long physicalSize;         /* physical size of the media in bytes  */
unsigned short physicalUnitSize;    /* flash erasable block size in bytes   */
char DOCType;                       /* DiskOnChip types defined in flioctl.h */
        FL_NOT_DOC        - 0    - Not DiskOnChip
        FL_DOC            - 1    - DiskOnChip 2000
        FL_MDOC           - 2    - DiskOnChip Millennium
        FL_DOC2000TSOP    - 3    - DiskOnChip 2000 TSOP
        FL_MDOCP          - 5    - DiskOnChip Millennium Plus
char lifetime;              /* lifetime indicator for the partition (1-10)        */
```

```
                        /* 1   - the media is fresh                          */
                        /* 10 - the media is close to the end of its life    */
char driverVer[10];     /*  driver version (NULL terminated string)          */
char OSAKVer[10];       /* TrueFFS version that driver is based on (NULL terminated string) */
                        /*The following values are of the specific partition */
unsigned long cylinders; /* Media.....                                       */
unsigned long heads;    /* geometry.....                                     */
unsigned long sectors;  /* parameters.....                                   */
} VolumeInfoRecord;     /* end struct */
```

VolumeInfoRecord Parameters	
logicalSectors	Number of logical sectors.
bootAreaSize	Number of physical bytes (not necessarily all usable) reserved for Binary Partition. For more information, see the section 5.1 of this manual.
baseAddress	Physical address in host memory where DiskOnChip window is located.
physicalSize	Amount of raw Flash memory available, in bytes. The total amount of storage space available for data storage will be lower due to formatting overhead, and presence of a Binary Partition.
physicalUnitSize	Size of erasable flash blocks, in bytes.
DOCType	Family of products to which this DiskOnChip belongs: DiskOnChip 2000 (DIP & DIMM), DiskOnChip Millennium, DiskOnChip 2000 TSOP and DiskOnChip Millennium Plus.
lifetime	Since the DiskOnChip is a flash memory, it is limited by the number of erase cycles. This parameter indicates the “lifetime status” (1 through 10), where 1 Indicates the media is fresh, and 10 Indicates that media is close to its end of life. Note that this value is only an estimate based on general lifetime statistics.
driverVer	Version number of TrueFFS driver for this specific operating system. **NOTE:** this is **not** the TrueFFS SDK version number.
OSAKVer	TrueFFS SDK version that TrueFFS driver is based on.

cylinders	Media geometry parameter: number of cylinders.
heads	Media geometry parameter: number of heads
sectors	Media geometry parameter: number of sectors per track.

Input record:

DO NOT CARE

Output record:

typedef struct {

VolumeInfoRecord info; / VolumeInfoRecord is defined in flioctl.h */*

FLStatus status;} flDiskInfoOutput;

3.2 FL_IOCTL_DEFRAGMENT

Regular writing to the flash makes periodic space reclamation, or garbage collection, necessary. TrueFFS performs such space reclamation automatically, usually on an immediate-need basis. This process takes time and slows down both the average and maximum time to perform a write operation.

The defragmentation process, performs early flash space reclamation. If an application needs to write a burst of data and has some idle time before the burst arrives, it can write the data more quickly when applying an early defragmentation call.

If it is necessary to write some data without interruption for space reclamation, defragmentation should be done before starting the write operation. The minimum number of bytes specified should include about 20% extra for FAT and BDTL overhead. For example, if you need to write a 16 KB file, specify about 40 sectors as a defragmentation target.

This function accepts as a parameter the minimum number of sectors that need to be available for immediate write. If the current amount of free writeable space is greater than this parameter, the function returns immediately. If not, it performs garbage collection operations until the amount of free space is at least equal to the quantity requested or there is no more reclaimable space left on the media.

If the amount of desired free space is unknown, a quick garbage collection procedure can be invoked by setting number of sectors needed to be –1. The amount of space that will be reclaimed in this procedure depends on the physical geometry of the media and the distribution of the data on the media. However, it is guaranteed to perform garbage collection in the most efficient manner (that is, the best “space reclaimed to time of operation” ratio).

In all cases, the call will return the actual number of free writable sectors currently available.

Note: This number is **not** the same as the free space on the volume, but represents only the amount of Flash memory that is in the erased state. A volume may be empty of files, yet have no free writeable sectors whatsoever.

To find the currently available space, request a defragmentation for 0 (zero) sectors. Defragmentation is not performed, however, the current number of free sectors is returned. Request a large number of

sectors to perform a general defragmentation of the volume. In this case, defragmentation completes with a failing status, since the number requested cannot be achieved.

Input record:

```
typedef struct {
long requiredNoOfSectors;   /* Minimum number of sectors to make available;          */
                            /* if -1 then a quick garbage collection operation is invoked*/

} flDefragInput;
```

Output record:

```
typedef struct {

long actualNoOfSectors;     /* Actual number of sectors available */

FLStatus status;} flDefragOutput;
```

3.3 FL_IOCTL_WRITE_PROTECT

TrueFFS includes a Key-controlled write-protection feature for DiskOnChip (software protection). Once a DiskOnChip is protected by the Key, it assumes read-only mode. Removing a Key can be done by an authorized user who knows the current Key.

The Key consists of 8 bytes (64-bit), each of which may be any 8-bit code character (2^{64} combinations). The Key is stored on the Flash disk in a manner that is both scrambled and hidden. That is, the Key is encrypted, and it is not possible to read the Flash disk to see the encrypted Key. If the Key is lost or forgotten by the authorized user, the Flash disk can become writable again by downloading all data from it, reformatting it, and uploading it. A new Key can then be enforced.

The same procedure can also be performed by unauthorized users. In this case however, the authorized user is able to determine that the Key was removed or changed.

A Key-protected DiskOnChip is available to an unauthorized user in read-only mode. All data may be read, but not written or modified. An authorized user can write to the Flash disk by temporarily disabling the write-protection (unlock) or permanently removing it (unprotect), depending on the parameters involved. In case the protection was temporarily removed both dismounting the DiskOnChip and system reset will cause the DiskOnChip to return to read-only mode.

The DiskOnChip, as shipped by M-Systems, are by default not Key-protected.

Note: This protection is not as strong as the hardware protection supported by DiskOnChip Millennium Plus devices.

Input record:

```
typedef struct {

unsigned char type;     /* Type of operation: FL_PROTECT / FL_UNPROTECT / FL_UNLOCK */

long Key[2];            /* 8 bytes Key                                              */

} flWriteProtectInput

#define FL_PROTECT      0  - Make the DiskOnChip write-protected.

#define FL_UNPROTECT  1  - Permanently remove the write-protection.
```

#define FL_UNLOCK 2 - Temporarily remove the write-protection.

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

3.4 FL_IOCTL_MOUNT_VOLUME

This function remounts the DiskOnChip. Remounting consists of discarding all in-memory control information kept by the TrueFFS driver, and rebuilding it. The remount consists of a low-level BDTL mount.

This function is not needed except in special circumstances. One of the most common uses of this function is when a user application accesses and modifies the DiskOnChip not via the file system or the TrueFFS driver API. An example of such an application is using the standalone DiskOnChip format utility "DFORMAT". In such a situation the TrueFFS driver is not updated with the DiskOnChip changes, and every further operation would be unreliable or even harmful. Forcing the TrueFFS driver to remount the driver will get updated with the changes, and enables it to operate reliably.

Input record:

typedef struct {

unsigned char type;

} flMountInput;

#define FL_MOUNT 0

#define FL_DISMOUNT 1

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

Note: Non valid arguments will force mounting of the DiskOnChip (identical to FL_MOUNT).

3.5 FL_IOCTL_FORMAT_VOLUME

This extended function is left for backwards compatibility with previous versions of TrueFFS SDK (former name TrueFFS-OSAK). Whenever possible use FL_IOCTL_FORMAT_PHYSICAL_DRIVE instead.

`FL_IOCTL_FORMAT_VOLUME` formats a volume, writing a new and empty file-system. All existing data is destroyed. Optionally, a low-level (flash translation layer) formatting is done.

Input record:

```
typedef struct {
unsigned char formatType;          /* Type of format*/
formatParams FAR1 fp;              /* Format parameters structure*/
} flFormatInput;
```

Options for formatType:

```
#define FAT_ONLY_FORMAT      0  - Perform FAT formatting only without the low-level format.
#define TL_FORMAT            1 – Perform both low-level and FAT format.
#define TL_FORMAT_IF_NEEDED  2 – Perform low-level and FAT format only if the current FAT
                                format is invalid.
#define TL_FORMAT_ONLY       8 – Performa low-level format only.
```

```
typedef struct
{
 /* TL formatting section */
      long int        bootImageLen;
            /* Space to reserve for a boot-image at the start of the
              medium. The BDTL volume will begin at the next higher
              erase unit boundary */

      unsigned int    percentUse;
/* The Translation Layer (TL) performance depends on how full the flash media is, becoming slower as
the media comes closer to 100% full. It is possible to avoid the worst-case performance (at 100% full)
by formatting the media to less than 100% capacity, thus guaranteeing free space at all times. This will
sacrifice some capacity. The standard value used is 98 */

      unsigned int    noOfSpareUnits;
```

```
/* BDTL partitions need at least one spare erase unit to function as a read/write media. That unit is
normally taken from the transfer units specified by the percentUsed field, but it is possible to specify
additional units (which takes more media space). The advantage of specifying spare units is that if all
the transfer units become bad and inerasable, the spare unit enables TrueFFS to continue its read/write
functionality. Conversely, if no spare units are available the media may switch into read-only mode. The
standard value used is 1 */
    unsigned long  vmAddressingLimit;  /* NOR flash formatting (not relevant for DiskOnChip)*/

    FLStatus       (*progressCallback)(int totalUnitsToFormat,int totalUnitsFormattedSoFar);
        /* Progress callback routine; will be called if not NULL.
           The callback routine is called after erasing each unit,
           and its parameters are the total number of erase units
           to format and the number erased so far.
           The callback routine returns a Status value. A value of
           OK (0) allows formatting to continue. Any other value
           will abort the formatting with the returned status code. */

    /* DOS formatting section */

    char    volumeId[4];
        /* Volume identification number */

    char*  volumeLabel;
        /* Volume label string. If NULL, no label */

    unsigned int    noOfFATcopies;
        /* It is customary to format DOS media with 2 FAT copies.
           The first copy is always used, but more copies make it
           possible to recover if the FAT becomes corrupted (a
           rare occurrence). On the other hand, this slows down
           performance and uses media space.
           The standard value to use is 2 */

 /* NOR flash formatting section (not relevant to DiskOnChip) */
```

unsigned int embeddedCISlength;

char embeddedCIS;*

}FormatParams;

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

3.6 FL_IOCTL_DELETE_SECTORS

Marks one or more consecutive absolute sectors as logically deleted. This function is required only in special circumstances. The reason for activating this function is that TrueFFS write performance depends to some degree on the amount of free space on the flash disk. A flash disk that is close to being full will show slower write performance. This is the result of the necessity to perform space reclamation (e.g. garbage collection) more often.

This function is used to increase the amount of logically free space by informing TrueFFS of absolute sectors that it considers used, but actually no longer contain useful data, that is, they can be deleted.

Typically, this function is beneficial for systems that use a non-FAT file system. If the file-system code is available, it is possible to identify the places where the file-system marks disk areas as logically deleted, and inform TrueFFS of this explicitly. In this case the customer is able to add a call to function *FL_IOCTL_DELETE_SECTORS* after deleting logical sectors.

Note: This special handling is **not** necessary when using a FAT file system (sometimes also called a DOS file system), since the TrueFFS driver is automatically aware of space management on FAT file systems.

Input record:

typedef struct {

long firstSector; / First logical sector to delete */*

long numberOfSectors; / Number of sectors to delete */*

} flDeleteSectorsInput;

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

3.7 FL_IOCTL_FORMAT_PHYSICAL_DRIVE

Formats a DiskOnChip. This extended functionality call allows the full range of functionalities introduced by TrueFFS 5.0 and the new M-Systems devices the Millennium Plus and DiskOnChip 2000 TSOP:

1) Divide the device into Binary partitions and BDTL partitions.
2) Perform a low-level binary format and BDTL format.
3) Protect up to two partitions of any kind.
4) Place quick mount format.
5) Place firmware boot file (or just leave space for it in the first binary partition).
6) Write an empty FAT file system onto BDTL partitions.

Notes:

- Formatting destroys all existing data.
- Formatting leaves all the BDTL volumes in the dismounted state, so that a mounting call is necessary afterwards.
- Millennium Plus and DiskOnChip 2000 TSOP support up to 4 partitions of any combination, Binary and BDTL, as long as there is at least one BDTL partition (Other devices support only one BDTL and one Binary partition).
- All H/W protection keys must be inserted before calling this routine.
- Some of the features introduced by TrueFFS 5.0 are ignored unless the *irFlags* field is set with the proper flag. The specific flags are mentioned in their proper fields.
- Formatting is controlled by a set of parameters defined in a *FormatParams2* structure.

Input record:

```
typedef struct {
FormatParams2 fp;          /* Format parameters structure          */
unsigned char formatType;  /* Type of format as defined in flioctl.h */
} flFormatPhysicalInput;
```

Options for *formatType:*

```
#define  TL_QUICK_MOUNT_FORMAT   1 – Apply the quick mount format
#define  TL_LEAVE_BINARY_AREA    8 – Leave the previous Binary partition unchanged
```

The FormatParams2 record is defined as follows:

```
typedef struct
```

```
{
unsigned char       percentUse;
        /* BDTL performance depends on how full the flash media is,
           becoming slower as the media becomes closer to 100% full.
           It is possible to avoid the worst-case performance
           (at 100% full) by formatting the media to less than 100%
           capacity, thus guaranteeing free space at all times. This
           of course sacrifices some capacity. The standard value
           used is 98 */

unsigned char           noOfBDTLPartitions;
                    /* Indicates the number of BDTL partitions (1 to 4). */

unsigned char           noOfBinaryPartitions;
        /* Indicates the number of Binary partitions (0 to 3). 0 will cause formatting with no Binary
        partitions. This value is ignored if the TL_LEAVE_BINARY_AREA flag is set. */
BDTLPartitionFormatParams*      BDTLPartitionInfo;
        /* BDTL partition information array  (see definition bellow) */

BinaryPartitionFormatParams  * binaryPartitionInfo;
     /* Binary partition information array (see definition bellow) */

/*****************************/
/* Special format features section   */
/*****************************/

void   *    exbBuffer;
        /* A buffer containing the EXB file. Optionally this file can
           contain only the first 512 bytes of the file while the rest
           will be sent using consecutive calls to FL_IOCTL_PLACE_EXB_BY_BUFFER */

unsigned long         exbBufferLen;   /* Size of the given EXB buffer */
```

```
unsigned long        exbLen;          /* The specific size to leave for the EXB */

unsigned short       exbWindow;       /* Set explicit DiskOnChip window base */

word                 exbFlags;        /* A combination of the following flags. */
#defineINSTALL_FIRST      1           - Make the device the first hard drive
#defineQUIET              4           - Do not show titles while BIOS expansion is found
#define INT15_DISABLE     8           - Disable INT15 hooking
#define FLOPPY            0x10        - Make device assume drive A: (This will not make it bootable)
#define SIS5598           0x20        - Support Windows NT platforms with SIS5598 VGA Chipset
#define EBDA_SUPPORT      0x40        - Support BIOS with EBDA.
#define NO_PNP_HEADER     0x80        - Do not place the PNP header
#define LEAVE_EMPTY       0x100       - Leave empty space for placing the specific firmware late
unsigned char        cascadedDeviceNo;
        /* Reserved for individual cascaded device formatting 0..n. For
           this value to have any affect the TL_SINGLE_CHIP_FORMATTING flag
           should be set in the flags field. This option is not currently implemented */

unsigned char        noOfCascadedDevices;
        /* This field must be supplied in order to perform a format of
           a single chip that will be eventually assembled as a cascaded
           device. The field should specify the number of DiskOnChips
           that will be eventually cascaded on the target platform.
          This option is currently not implemented */

FLStatus (*progressCallback)(int totalUnitsToFormat,
                             int totalUnitsFormattedSoFar);
        /* Progress callback , will be called if not NULL.
           The callback routine is called after erasing each unit,
           and its parameters are the total number of erase units
           to format and the number erased so far.
```

*The callback routine returns a Status value. A value of OK (0) allows formatting to continue. Any other value will abort the formatting with the returned status code. */*

The BDTLPartitionFormatParams record describes the individual BDTL partitions:

```
typedef struct
{
unsigned long  length;
    /* The size of the usable storage space. The size will be
       rounded upwards to a multiplication of a block size.
       The size of the last partition will be calculated automatically,
       but if the requested size is greater then the remaining space
       an error code will be returned. Requesting zero size for any
        partition but the last will generate an flBadParameters status. */

unsigned int    noOfSpareUnits;
    /* BDTL needs at least one spare erase unit in order to function
       as a read/write media. It is possible to specify more than one
       spare unit, which takes more media space. The advantage of
       specifying more than one spare unit is that if one of the flash
       erase units becomes bad and inerasable in the future, then one
       of the spare units can replace it. In that case, a second spare
       unit enables TrueFFS to continue its read/write functionality,
       whereas if no second spare unit is available the media goes into
       read-only mode. The standard value used is 1 */

unsigned char flags;
#define TL_FORMAT_FAT        2  /* Add FAT format on the media              */
#define TL_OLD_FORMAT        4  /* Try formatting with 1 sector per cluster */

unsigned char  volumeId[4];  /* DOS partition identification number          */
```

```
unsigned char*          volumeLabel;  /*DOS partition label string. If NULL, no label */

unsigned char  noOfFATcopies;
      /* It is customary to format DOS media with two FAT copies. The
         first copy is always used, but more copies make it possible
         to recover if the FAT becomes corrupted (a rare occurrence).
         On the other hand, this slows down performance and uses media
         space. The standard value used is 2. */

unsigned char  protectionKey[8];   /* The key for the protection*/

unsigned char  protectionType;
      /* PROTECTABLE                - 1 - Must be added for any protection attribute */
      /* READ_PROTECTED             - 2 - Protect against read operations            */
      /* WRITE_PROTECTED            - 4 - Protect against write operations           */
      /* LOCK_ENABLED               - 8 - Enables the hardware lock signal           */
      /* CHANGEABLE_PROTECTION      - 64 - Protection type can be changed            */
}BDTLPartitionFormatParams;
```

The BinaryPartitionFormatParams record describes the individual BDTL partitions:

```
typedef struct
{
unsigned long length;          /* Required number of usable bytes on the  partition.*/

unsigned char sign[4];         /* signature of the binary  partition to format. */

unsigned char signOffset;
                /* offset of the signature. This value should always be 8, but it
                   can also accept 0 for backwards compatibility reasons. Note that
                   if the offset is 0 EDC\ECC is neutralized */
```

unsigned char protectionKey[8]; / The key for the protection*/*

unsigned char protectionType;

/ PROTECTABLE - 1 - Must be added for any protection attribute */*

/ READ_PROTECTED - 2 - Protect against read operations */*

/ WRITE_PROTECTED - 4 - Protect against write operations */*

/ LOCK_ENABLED - 8 - Enables the hardware lock signal */*

/ CHANGEABLE_PROTECTION - 64 - Protection type can be changed */*

}BinaryPartitionFormatParams;

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

3.8 Hardware Protection

The extended functionality calls described in this section perform hardware read/write protection related operations, and consequently can be used only with DiskOnChip devices that have the required hardware support (currently DiskOnChip Millennium Plus only).

Different functions handle Binary and BDTL partitions. The usage is identical only that BDTL partitions use FL_IOCTL_BDTL_HW_PROTECTION while binary partition use FL_IOCTL_BINARY_HW_PROTECTION.

Method of operation

The DiskOnChip Millennium Plus enables you to define two partitions that will be key protected (in hardware), against any combination of read or write operations. Defining their size and protection attributes (read/write/changeable/lock), is done in the media formatting stage (DFORMAT utility or the format extended function call). You may define one partition as "changeable" i.e. its password and attributes are fully configurable (from read to write, both or none and vise versa) at any time. Note that "un-changeable" partition attributes cannot be changed unless the media is reformatted.

The DiskOnChip Millennium Plus has an additional H/W safety mechanism. If the Lock option is enabled (using one of the extended functions), and the Lock pin of the DiskOnChip is set, then the protected partition will have an additional H/W lock preventing the use of the key, i.e. not even using the correct key will provide access to the protected partitions.

A good analogy to the way the DiskOnChip hardware protection works would be the following:

You can decide whether or not to install a lock on your door (this is done in the DFORMAT stage) and whether you want to add a safety chain on it (Lock enabled). However at any given point in time you can decide whether to leave the "key" inside (insert key) allowing free access or removing it, leaving the door closed. No protection violation operation can be done without inserting the key. If you installed a safety chain on your door you can always use it (assert the DiskOnChip LOCK pin) therefore preventing access even if someone has the correct key or even if the key is currently inserted. If the safety chain was not installed during the format stage (or later on when the partition is changeable) then the DiskOnChip LOCK pin is ignored by the H/W protection logic.

Note: The target volume does not have to be mounted before calling a H/W protection routine.

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

A change of any of these attributes causes a reset of the protection mechanism and consequently the removal of all the devices protection keys. Care should be taken to avoid interference between different protected partitions. For example, a key inserted into one partition will be removed when another partition is instructed to change its protection attributes (changing key for example).

The only way to write or read from a read or write protected partition is to use the insert key call (not even format will remove the protection). This is also true for modifying its attributes (key, read, write and lock enable state).The key is removed in each one of the following events:

- Power down.

- Change one of the protection attributes (not necessarily to the same partition).
- Write operation to the IPL area using FL_IOCTL_WRITE_IPL.
- Removing of the protection key through the use of the functionalities.

Note: In order to make a partition changeable the specific flag must be added to the protectionType field of the format record. Without it, functions attempting to change protection attributes will return error codes.

Note: 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 DiskOnChip LOCK pin must be negated first.

Note: Only one partition in a device can be changeable.

3.8.1 FL_IOCTL_BDTL_HW_PROTECTION

Not all DiskOnChip devices support the H/W protection feature. To find out, call the function FL_IOCTL_INQUIRE_CAPABILITIES with option SUPPORT_HW_PROTECTION

As of the writing of these lines the only device with H/W protection features is the DiskOnChip Millennium Plus.

The hardware protection extended function is subdivided into sub-functions. All of the sub-functions use the same record for input and output.

Input record:

typedef struct {

unsigned char protectionType;/ See PROTECTION_GET_TYPE table bellow */*

unsigned char key[8]; / The key */*

Unsigned char type; / One of the following flags: */*

#define PROTECTION_INSERT_KEY - 0 - Insert key (disabling protection)

#define PROTECTION_REMOVE_KEY - 1 - Remov key (restoring protection)

#define PROTECTION_DISABLE_LOCK - 2 - Do not enable the H/W LOCK pin

#define PROTECTION_ENABLE_LOCK - 3 - Enable the H/W LOCK pin

#define PROTECTION_GET_TYPE - 4 - Get the current protection status

#define PROTECTION_CHANGE_KEY - 5 - Change the protection key

#define PROTECTION_CHANGE_ TYPE - 6 - Change the protection type (read \ write protected)

} flProtectionInput;

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

Note: Some protection operations return values in the *flProtectionInput* record.

Following are the action types and the required arguments in the *flProtectionInput* record:

PROTECTION_INSERT_KEY

Inserts the key to a protected area

FlProtectionInput Parameters	
key	The key to be inserted

Note: Inserting a wrong key to a partition that already has a key inserted does not fail.

PROTECTION_REMOVE_KEY

Remove the key from a protected partition

PROTECTION_DISABLE_LOCK

Disable the DiskOnChip LOCK pin signal effect on the Key.

PROTECTION_ENABLE_LOCK

Enables the DiskOnChip LOCK pin signal effect on the Key.

PROTECTION_CHANGE_KEY

Change the Key of a protected partition.

FlProtectionInput Parameters	
key	The new Key for the protection area

CHANGE_PROTECTION_TYPE

Change the protection type.

FlProtectionInput Parameters			
protectionType	PROTECTABLE	1	Must be added for the operation to succeed.
	READ_PROTECTED	2	Partition is protected against read operation
	WRITE_PROTECTED	4	Partition is protected against write operation

PROTECTION_GET_TYPE

Gets a protected partition status

FlProtectionInput Parameters			
type	Action type		
protectionType	PROTECTABLE	1	The partition can be protected
	READ_PROTECTED	2	Partition is protected against read operation

	WRITE_PROTECTED	4	Partition is protected against write operation
	LOCK_ENABLED	8	The hardware LOCK signal is enabled
	LOCK_ASSERTED	16	The hardware LOCK signal is currently asserted
	KEY_INSERTED	32	Protection is temporarily removed
	CHANGEABLE_PROTECTION	64	The protection attributes of the partition can be changed without a full reformatting of the media
FlProtectionOutput			
status	flOK – succeeded flNotProtected – not a protected partition		

3.8.2 FL_IOCTL_BINARY_HW_PROTECTION

This IOCTL function is identical to the function *FL_IOCTL_BDTL_HW_PROTECTION*, only for Binary Partitions.

3.9 FL_IOCTL_OTP

The functions described in this section perform standard operations on the OTP area. Not all M-Systems devices support this feature. To find out, call the function *FL_IOCTL_INQUIRE_CAPABILITIES* with option *SUPPORT_OTP_AREA* (see inquire capabilities extended functionality).

The DiskOnChip Millennium Plus has a ROM-Like hardware feature (referred to as “OTP”). This feature provides a dedicated 6KB area on the flash that can be programmed once and then locked forever (by the DiskOnChip hardware). Writing to the OTP section can be done only once (EDC is automatically added) after which the area is H/W protected against write and erase operations. The total size of the area , the actually used size and the locked state of the area can be retrieved in addition to normal read of the area.

The OTP extended function is subdivided into sub functions.

A pointer to *flOtpInput* structure is passed to all of the OTP sub-functions. The same record is sent both as input and as output.

Input record:

```
typedef struct {
unsigned long    length;      /*  Length to read/write/size in bytes      */
unsigned long    usedSize;    /*  The written size of the area in bytes   */
unsigned char    lockedFlag;  /*  The area condition (LOCKED_OTP)          */
```

unsigned char FAR1 buffer; /* pointer to user buffer */*
word type; / One of the types bellow */*
} flOtpInput; / flOtpOutput is the same */*

#define OTP_SIZE 1 - Get OTP statistics.
#define OTP_READ 2 – Read from the OTP area.
#define OTP_WRITE_LOCK 3 - Write and permanently lock the OTP area.

Output record:

typedef struct {
FLStatus status;
} flOutputStatusRecord;

OTP_SIZE

Get size of the OTP area.

FlOtpInput Parameters	
type	OTP_SIZE
FlOtpOutput	
status	flOK – succeeded
length	The length of the OTP area in bytes
usedSize	The used size of the OTP area in bytes
lockedFlag	The area current state: #define LOCKED_OTP 1 – The area is currently locked.

OTP_READ

Read from OTP area to a user buffer

FlOtpInput Parameters	
type	OTP_READ
length	The length to read in bytes
usedSize	The offset of the first byte to read
buffer	Pointer to user buffer to read into

FlOtpOutput	
status	flOK – succeeded flDataError – EDC/ECC error flBadLength – size exceeds OTP area size

OTP_WRITE_LOCK

Write to OTP area, add EDC/ECC and lock the customer OTP.

FlOtpInput Parameters	
type	OTP_WRITE_LOCK
length	The length to write in bytes
buffer	Pointer to user buffer to write from
FlOtpOutput	
status	flOK – succeeded flDataError – EDC/ECC error flHWProtection – OTP was already locked flBadLength – size exceeds OTP area size

3.10 Unique ID

Each DiskOnChip Millennium Plus device has a unique 16 bytes ID number. The number is randomly generated and is guaranteed to be unique to this DiskOnChip device alone (i.e. no two DiskOnChip Millennium Plus units in the world are the same!). When ordering large quantities of DiskOnChip units, a 4 bytes customer ID signature can be burned into them (in the FAB). Both ID's are hardware protected against write and erase operation.

3.10.1 FL_IOCTL_CUSTOMER_ID

Returns the H/W embedded customer ID. Not all M-Systems devices support this feature. To find out call the function FL_IOCTL_INQUIRE_CAPABILITIES with option SUPPORT_CUSTOMER_ID (see inquire capabilities extended functionality).

Returns 4 bytes customer ID information.

As of the writing of these lines the only device with the Customer ID feature is the DiskOnChip Millennium Plus.

Input record:

DO NOT CARE

Output record:

{

unsigned char id[4] ;

FLStatus status;

} flCustomerIdOutput;

3.10.2 FL_IOCTL_UNIQUE_ID

Returns the H/W embedded unique device ID. Not all M-Systems devices support this feature. To find out, call the function FL_IOCTL_INQUIRE_CAPABILITIES with option SUPPORT_UNIQUE_ID (see inquire capabilities extended functionality).

Returns 16 bytes chip ID information.

As of the writing of these lines the only device with the Device ID features is the DiskOnChip Millennium Plus.

Input record:

DO NOT CARE

Output record:

{

unsigned char id[16] ;

FLStatus status;

} flJUniqueIdOutput;

3.11 FL_IOCTL_NUMBER_OF_PARTITIONS

Returns the number of BDTL Partitions in a specific device.

Input record:

DO NOT CARE

Output record:

typedef struct {

unsigned char noOfPartitions;

FLStatus status;

} flCountPartitionsOutput;

3.12 FL_IOCTL_INQUIRE_CAPABILITIES

Returns if the current hardware and software support a specific feature.

Input record:

typedef struct {

flCapability capability; / See flags below */*

} flCapabilityInput;

Output record:

typedef struct {

flCapability capability; / See flags bellow */*

FLStatus status;

} flOutputStatusRecord;

CAPABILITY_NOT_SUPPORTED	*- 0*
CAPABILITY_SUPPORTED	*- 1*
SUPPORT_UNERASABLE_BBT	*- 2*
SUPPORT_MULTIPLE_BDTL_PARTITIONS	*- 3*
SUPPORT_MULTIPLE_BINARY_PARTITIONS	*- 4*
SUPPORT_HW_PROTECTION	*- 5*
SUPPORT_HW_LOCK_KEY	*- 6*
SUPPORT_CUSTOMER_ID	*- 7*
SUPPORT_UNIQUE_ID	*- 8*
SUPPORT_DEEP_POWER_DOWN_MODE	*- 9*
SUPPORT_OTP_AREA	*- 10*
SUPPORT_WRITE_IPL_ROUTINE	*- 11*

3.13 FL_IOCTL_SET_ENVIRONMENT_VARIABLES

TrueFFS based driver support several runtime configuration variables.

For each of these options there is a global variable inside the driver that dictates the driver behavior. These global variables are called environment variables. It is required to set the values of the environment variables before the first mount and then not to change them again.

3.13.1 Using Translation Layer Cache

Turning on this option improves performance, but requires additional RAM resources.

The drivers Flash Translation Layer uses a small part of each flash unit for control information. This control information allows accessing the data stored on the DiskOnChip as a Virtual Block Device.

If the variable is set to 1, then the TL keeps in RAM an identical table of the necessary control information. Whenever it is necessary to read a table's entry, the TL can read it from the RAM, saving time by not having to read that sector from the DiskOnChip.

0 – cache disabled.

1 – cache enabled.

3.13.2 Using the Function flUseisRAM

During the DiskOnChip mount process, a test is made to see if the memory address where the DiskOnChip resides behaves like RAM. Since flash media does not behave like RAM, this function can detect if the DiskOnChip is located at the address where it is supposed to be, or not.

The testing for RAM is simple. It reads and stores the value that is written in the supposed DiskOnChip address, writes a new value to the address, and reads again. If the new value is there, the memory address behaves like RAM and the mount process stops (restoring the previous values). If the old value is still there, the media does not behave like RAM and the mount process continues. Usually this test is harmless, but in some cases the direct memory access can cause problems. If this is the case the test should be skipped.

0 – Skip test.

1 – Perform test.

3.13.3 Using 8-bit Access to the DiskOnChip

This option defines the type of access to the DiskOnChip. When set to 1 the access is 8 bit, and when set to 0 the access is 32 bit. This is not to say that the DiskOnChip actually supplies the full 32 bit at once. But usually a request for 32bit is much faster then a 'for' loop of single bytes.

0 – 32 bit access.

1 – 8 bit access.

3.13.4 Combining all Drives into a Multi-DOC

TrueFFS SDK 5.0 introduces the Multi-DOC feature. The Multi-DOC feature can combine several devices into a single large media. This feature formats multiple TrueFFS supported devices to form a single large media.

The low-level format process of the combined devices can be done separate, but it is recommended to perform the format on the already combined target devices, allowing the proper BPB (Boot Partition Block) to be written. High-level format, such as FAT file system format, must be done on the combined Multi-DOC device in order to allocate the proper media size.

Physical calls like "Get Customer and Unique ID", "Read/Write and size of OTP area", "Deep Power Down Mode" and "Physical Media Information" will ignore the Multi-DOC feature and access one of the devices.

Binary routines will ignore the Multi-DOC feature, except for the format routine, which will restrict the Binary Partition(s) to the first device.

H/W protection routines will work as if the device is a single device. Protecting BDTL partition requires that all of the devices support H/W protection, while Binary Partitions require only the first device to support H/W protection since the Binary Partitions are restricted to the first device.

#define FL_MULTI_DOC_NOT_ACTIVE 0

#define FL_MULTI_DOC_ACTIVE 1

3.13.5 Set Driver Policy

This option defines the internal TL algorithms.

#define FL_DEFAULT_POLICY 0 - for the default optimum configuration.

#define FL_COMPLETE_ASAP 1 - TrueFFS SDK will try to complete the current command as fast as possible, ignoring all future considerations.

Input record:

typedef struct {

FLEnvVars varName; / Enum describing the variable */*

int varValue; / New variable value */*

} flEnvVarsInput;

The varValue is an enumerator type defined as:

typedef enum { / Variable types */*

FL_IS_RAM_CHECK_ENABLED = 1,

FL_NFTL_CACHE_ENABLED = 2,

FL_DOC_8BIT_ACCESS = 3,

FL_MULTI_DOC_ENABLED = 4,

FL_SET_POLICY = 5,

} FLEnvVars;

Output record:

typedef struct {

int prevValue; / The previous value of the variable */*

FLStatus status;

} flEnvVarsOutput;

3.14 FL_IOCTL_PLACE_EXB_BY_BUFFER

Place an EXB file (firmware file) on to the media using small buffers.

Note: The first buffer must be at least 512 bytes long.

Note: Binary operations to Binary Partition 0 (except for remove \ insert key) will reset the process.

Note Only M-Systems EXB files are supported by this routine.

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

Input record:

typedef struct {

byteFAR1 buf; /* Buffer of EXB file */*

unsigned long bufLen; / Buffer length */*

word exbWindow / Explicitly set device window. 0 will automatically set window */*

word exbFlags / A combination of EXB flags see bellow: */*

} flPlaceExbInput;

exbFlags	*INSTALL_FIRST*	*1*	Make the device the first hard drive
	QUIET	*4*	Do not show titles while BIOS expansion is found
	INT15_DISABLE	*8*	Disable INT15 hooking
	FLOPPY	*16*	Make device assume drive A: (This will not make it bootable)
	SIS5598	*32*	Support Windows NT platforms with SIS5598 VGA Chipset
	EBDA_SUPPORT	*64*	Support BIOS with EBDA.
	NO_PNP_HEADER	*128*	Do not place the PNP header
	LEAVE_EMPTY	*256*	Leave firmware area empty.

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

3.15 FL_IOCTL_WRITE_IPL

Write data to the IPL area (see glossary for details) of the Millennium Plus and DiskOnChip 2000 TSOP devices. Other DiskOnChip have other means for this purpose.

- DiskOnChip 2000 does not have a writable IPL area.
- Millennium 8MB uses the first 512 bytes of flash as the IPL data. This area can be written to using the Binary partition extended functionalities.

To find out if your device support the write IPL extended functionality call the function FL_IOCTL_INQUIRE_CAPABILITIES with option SUPPORT_WRITE_IPL_ROUTINE (see inquire capabilities extended functionality).

Input record:

unsigned char FAR1 buf; /* IPL data buffer */*
word bufLen; / IPL data buffer length */*
} flIplInput;

Output record:

typedef struct {

FLStatus status;

} flOutputStatusRecord;

3.16 FL_IOCTL_DEEP_POWER_DOWN_MODE

Changes the power consumption mode of a DiskOnChip Millennium Plus device.

To verify that your device support this feature call the function FL_IOCTL_INQUIRE_CAPABILITIES with option SUPPORT_DEEP_POWER_DOWN_MODE

(see inquire capabilities extended functionality).

Note: Once in power down the DiskOnChip device boot detection mechanism is disabled. This means that if you platform uses M-System BIOS expansion driver and your system initiated a reset command without asserting the DiskOnChip reset pin the driver will not be loaded.

Input record:

typedef struct {

unsigned char state; / DEEP_POWER_DOWN– low power consumption*

otherwise - regular power consumption/*

} flPowerDownInput;

#define DEEP_POWER_DOWN 1

Output record:

typedef struct {

FLStatus status;
} flOutputStatusRecord;

3.17 FL_IOCTL_BDK_OPERATION

The functions described in this section perform standard operations on a Binary Partition (read/write/erase/create/get size of binary partitions). Such partition cannot be accessed by the file system, and is reserved for customer use. The most common use for this partition is to store system boot code or OS image file. Operations on Binary Partition blocks do not affect the file system activity and will not slow down performance. For a full description, see BOOT SDK Developer's Guide.

The BDK extended functions are subdivided into sub-functions.

Input record:

```
typedef struct {
unsigned char type;          /* One of the operation types mentioned bellow: */
BDKStruct bdkStruct;         /* parameters for Binary operations see bellow */
} flBDKOperationInput;

#define BDK_INIT_READ        0
#define BDK_READ             1
#define BDK_INIT_WRITE       2
#define BDK_WRITE            3
#define BDK_ERASE            4
#define BDK_CREATE           5
#define BDK_GET_INFO         6
```

The BDKStruct is common to all of the sub-functions and is defined as follows:

```
typedef struct {
unsigned char oldsign[4];    /* Signature of the Binary partition to work on      */
unsigned char newsign[4];    /* Signature of the new Binary partition to create   */
                             /* (relevant only for bdkCreate)                     */
unsigned char signoffset;    /* Offset of the signature                           */
unsigned long startingBlock; /* First block in the partition to operate on        */
unsigned long length;        /* Number of bytes to read/write or number of blocks to erase */
unsigned char flags;         /* Option flags: activate EDC/ECC mechanism,         */
                             /*               write full or partial partition     */
unsigned char FAR2 bdkBuffer; /* Read/write buffer                                */
} BDKStruct;
```

Output record:

```
typedef struct {
FLStatus status;
} flOutputStatusRecord;
```

Note: Some binary operations return values in the *bdkStruct* input record.

3.17.1 BDK_INIT_READ

Performs an initialization procedure on the Binary Partition before BDK_READ is called.

This function checks that the read operation about to be performed on the Binary Partition is within the sub partition's "already written" boundary. This function must be called before any BDK_READ operation, and it is followed by a sequence of BDK_READ calls. If the flag EDC is on, the error detection and correction (EDC/ECC) mechanism is activated.

Note: Read operations beyond the FFFF mark fail in the initialization stage, with status *flNoSpaceInVolume*. For further explanation of *BDK_COMPLETE_IMAGE_UPDATE*, refer to the Boot SDK developers guide.

BDKStruct Parameter	
StartingBlock	Unit number from which to start the read operation (counting from zero)
Length	Number of bytes to read
OldSign	Signature of sub partition
flags	EDC
signOffset	Offset of the sub partition signature (0 or 8)
Returns	
FLStatus	0 on success, non-zero on failure

3.17.2 BDK_READ

Reads from a sub partition of a Binary Partition.

BDK_INIT_READ must be called immediately before this operation. The *length* parameter in the *BDKStruct* must not cause a read operation from 2 different erasable blocks. To avoid such complications, it is recommended to keep the *length* parameter at a flash page size (512 bytes for interleave-1 and 1024 bytes for interleave-2) and if possible use full erasable blocks.

BDKStruct Parameters	
length	Number of bytes to read
bdkBuffer	Pointer to a buffer that receives the binary data
Returns	
FLStatus	0 on success, non-zero on failure

3.17.3 BDK_INIT_WRITE

Performs an initialization procedure before `BDK_WRITE` is called.

`BDK_INIT_WRITE` checks that the write operation about to be performed on the Binary Partition is within the sub partition boundary. It must be called before calling `BDK_WRITE`. If the flag EDC is on, the error detection and correction (EDC/ECC) mechanism is activated.

BDKStruct Parameters	
startingBlock	Unit number from which to start the write operation
length	Number of bytes to write
oldSign	Signature of sub partition
flags	EDC
	BDK_COMPLETE_IMAGE_UPDATE
signOffset	Offset of the partition signature (0 or 8)
Returns	
FLStatus	0 on success, non-zero on failure

3.17.4 BDK_WRITE

Writes to a sub partition of a Binary Partition.

`BDK_INIT_WRITE` must be called immediately before this operation. The *length* parameter in the *BDKStruct* must not cause a write operation to 2 different erasable blocks. To avoid these complications, it is recommended to keep the *length* parameter at a flash page size (512 bytes for interleave-1 and 1024 bytes for interleave-2).

BDKStruct Parameters	
Length	Number of bytes to write
oldSign	Signature of the sub partition to write into
bdkBuffer	Pointer to a buffer containing the binary data to write.
flags	ERASE_BEFORE_WRITE – block will be erased before it is written
Returns	

FLStatus	0 on success, non-zero on failure

3.17.5 BDK_ERASE

Erase sequential blocks in a sub partition of a Binary Partition. The signature indicating the sub partition remains and only the data is erased.

BDKStruct Parameters	
startingBlock	Block number of the sub partition where the erase operation starts
length	Number of blocks to erase
oldSign	Signature of the sub partition to erase
signOffset	Offset of the sub partition's signature (0 or 8)
Returns	
FLStatus	0 on success, non-zero on failure

3.17.6 BDK_CREATE

Create a new, empty sub partition in a Binary partition by overwriting an existing sub partition. The new partition is always created at the beginning of the old one. If the new sub partition is larger than the old one, an error status is returned.

For example, if you start with a 4MB sub partition having signature "AAAA" and create on top of it a 1MB sub partition having signature "BBBB", the result is a 1MB sub partition having signature "BBBB" followed by a 3MB sub partition having signature "AAAA".

Note: If a partition with the same signature already exists, it might be hard to tell which is the first of the two. To avoid complications, save the data of the old partition and rewrite it after the creation of the new "enlarged" partition.

Note: The *bdkCreate* function (like any other Binary partition function) cannot cross the Binary partition boundaries.

BDKStruct Parameters	
length	Length, in erasable blocks, of the sub partition to create
oldSign	Signature of existing sub partition
newSign	Signature of sub partition to create
signOffset	The offset of the sub partition's signature (0 or 8)

Returns	
FLStatus	0 on success, non-zero on failure

3.17.7 BDK_GET_INFO

Returns the number of Binary partitions in the physical drive, the total size of a specific sub partition and its used size.

BDKStruct Parameters	
oldSign	Signature of sub partition to find its length
signOffset	Offset of the sub partition's signature (0 or 8)
startingBlock	Block of the sub-partition to start the search
Returns	
FLStatus	0 on success, non-zero on failure
Ioreq Parameters	
irLength	Binary Partition physical length in bytes
BDKStruct Parameters	
length	Used virtual size of the sub partition in bytes
startingBlock	Virtual size of the sub-partition in bytes
flags	Number of Binary Partitions

Additional Information and Tools

Additional information about DiskOnChip, including application notes, data sheets, and utilities can be found at http://www.m-sys.com.

Additional tools and documents are listed in the following table:

Document/Tool	Description
DiskOnChip Millennium Plus Data Sheet	Data Sheet
DiskOnChip 2000 TSOP Data Sheet	Data Sheet
DiskOnChip Millennium Data Sheet	DiskOnChip Millennium Data Sheet
DiskOnChip 2000 Data Sheet	DiskOnChip 2000 Data Sheet
AP-DOC-017	Application Note - Designing with the DiskOnChip in Windows CE.
IM-DOC-020	BOOT SDK Developer's Guide
Driver Installation Manuals	Manuals for specific OSs are enclosed with the driver package in PDF format.
AP-DOC-031	Application Note - Designing with the DiskOnChip Millennium in a PC Environment
AP-DOC-030	Application Note - Designing with the DiskOnChip Millennium in a RISC Environment
AP-DOC-039	Application Note – On board Programming of the DiskOnChip TSOP
DiskOnChip Utilities	DiskOnChip Utilities User Manual
DiskOnChip-GANG	DiskOnChip GANG Programmer User Manual
DiskOnChip EVB	DiskOnChip Evaluation Board

How to Contact Us

Internet: http://www.m-sys.com

E-mail: info@m-sys.com

USA Office:
M-Systems Inc.
8371 Central Ave, Suite A
Newark CA 94560
Phone: +1-510-494-2090
Fax: +1-510-494-5545

Taiwan Office:
Room B, 13 F, No. 133 Sec. 3
Min Sheng East Road
Taipei, Taiwan
R.O.C.
Tel: +886-2-8770-6226
Fax: +886-2-8770-6295

Korea Office:
18th FL, Kyoung Am Bldg.
157-27, Samsung-Dong
Kangnam-Ku, 135-090
Seoul, Korea
Phone: +82-2-565-5355
Fax: +82-2-555-3612

Japan Office:
M-Systems Japan Inc.
Arakyu Bldg., 5F
2-19-2 Nishi-Gotanda Shinagawa-ku
Tokyo 141-0031
Phone: +81-3-5437-5739
Fax: +81-3-5437-5759

China Office:
25A International Business Commercial Bldg.
Nanhu Rd., Lou Hu District
Shenzhen, China 518001
Phone: +86-755-519-4732
Fax: +86-755-519-4729

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