UBI - Unsorted Block Images

T. Gleixner

F. Haverkamp

A. Bityutskiy

UBI - Unsorted Block Images

by T. Gleixner, F. Haverkamp, and A. Bityutskiy

Revision History

Revision v1.0.0 2006-06-09 Revised by: haver
Release Version.

Table of Contents

List of Tables

Chapter 1. Document conventions

All sections marked as *Enhancements* are not part of the initial implementation. These sections are informal and not part of the design review process. The sections are provided to point out future extensions and answers to questions which came up during the design, i.e. scalability and speed. They have to be considered in the base implementation in a way that the extensibility is not prohibited or unnecessarily complicated.

Chapter 2. Introduction

UBI is a general purpose flash management layer which has similar functionality as the Logical Volume Manager. The name was created with the picture of Unsorted Block Images in mind and accidentally is also a Latin word, which means: where.

The basic idea is that the UBI layer provides the management for multiple logical volumes, which can contain either static data or dynamic contents e.g file systems.

The management of multiple logical volumes on one or more flash devices of the same type requires the following functionality:

- Bad block management
- Wear levelling across the device(s)
- Logical to physical block mapping
- Volume information storage
- Device information

UBI is not a traditional FLASH Translation Layer for block device emulation. UBI encapsulates the FLASH chip management from file systems and user space interfaces. A traditional FLASH Translation Layer can easily be built on top of an UBI volume.

UBI addresses common FLASH handling mechanisms which have been redundantly implemented in FTLs and FLASH file systems. This allows FLASH file system developers to concentrate on the file system design while participating on the ongoing development and improvement of UBI.

The separation of the FLASH management and the file systems, FTLs and other possible users provides a better layered concept than the intermingled solutions which are available today. Beside the replacement of redundant and single purpose implementations this provides a benefit in testability and error analysis.

UBI Volumes

UBI volumes are container for different data contents. The UBI client can only access the data contents, not the UBI metadata.

UBI handles two volume types:

- static volumes
- dynamic volumes

Static Volumes

Static volumes contain only static data e.g. boot-code, operating-system images, initial-ramdisks, or read-only file-systems. Before writing blocks of a static volume, the data to be written into the blocks is completely known and treated read-only during later operation. Static volumes have:

- Linear usage of logical eraseblocks
- Known number of used eraseblocks
- CRC protection across all used eraseblocks

It is required that static volumes have a linear (consecutive) usage of logical erase blocks to simplifiy consistency checks and allow fast access to the data contents. This applies especially for use cases, where a boot loader has to load e.g. a kernel image from an UBI device. A static volume which does not have all blocks in place is not usable.

Dynamic Volumes

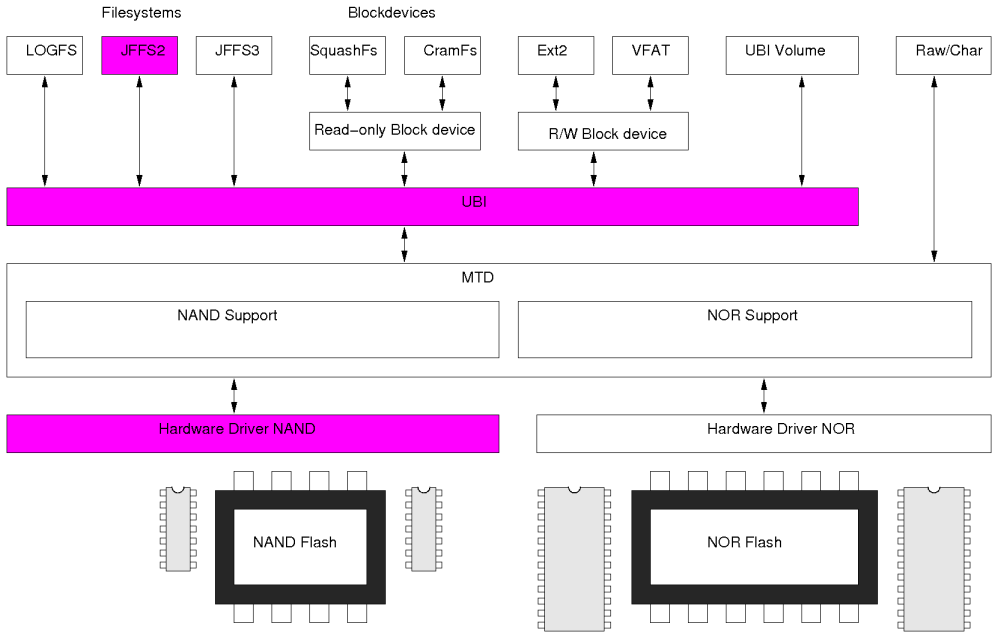
The dynamic volumes contain data which can be changed during operation. Dynamic volumes are a requirement to provide support for writeable filesystems. Dynamic volumes have:

- Nonlinear usage of logical eraseblocks
- Unknown number of used eraseblocks
- CRC protection only for block moving

The users of dynamic volumes have to implement their own mechanisms for data integrity. UBI has no way to provide this because the content of a block is not known when the UBI header is written. The usage pattern of the logical erase blocks is not limited and UBI handles holes in the allocated blocks by treating them as empty, i.e. reading such a block returns an all 0xFF buffer of the requested size without accessing the pyhsical FLASH chip, writing to an unallocated block results in allocation before the data contents are written.

Chapter 3. MTD integration

Figure 3-1. UBI/MTD Integration



This image shows how UBI fits into the MTD software architecture. Note that almost all services like read/write-able and read-only static data blob based file systems are based on the UBI support. The raw access to the MTD devices is available due to the layered structure of MTD. It would require to change the complete base infrastructure of MTD to avoid this.

One UBI instance handles one MTD device. This can be a physical device, a MTD partition or a concatenated MTD device consisting of multiple MTD devices. The concatenation support is available via mtdconcat in the kernel. The MTD NAND FLASH layer is also capable of concatenating multiple equally sized NAND chips.

Note, that the usage of UBI on concatenated devices requires equvivalent boot loader support.

Note, that the usage of UBI on top of a partitioned MTD device is not intended, but is possible due to the existing MTD infrastructure. In some cases, where e.g. a special area of the FLASH has to be excluded from the UBI block management, partitions can be used to do so.

Simple partitioning

Table 3-1. Simple partitioning scheme

Partition	Physical erase blocks

Partition	Physical erase blocks
Bootloader	0 - 3
UBI	4 - max. blocks

This would allow to have wear levelling across the different volumes on the UBI partition and ensures that the bootloader area is not touched.

Another possibility is to mark the boot loader blocks with a "non movable" flag in the UBI header, so UBI does not include these blocks into the wear levelling. This requires an UBI aware boot loader design and might not be possible at all, when the boot code is required to start at offset 0 of a block.

Complex partitioning

Table 3-2. Complex partitioning scheme 1

Partition	Physical erase blocks
Bootloader 0	0 - 3
UBI	4 - 63
Bootloader 1	64 - 67
UBI	68 - max. blocks

Table 3-3. Complex partitioning scheme 2

Partition	Physical erase blocks	Chip
Bootloader	0 - 3	0
UBI	4 - max. blocks	0
Bootloader	0 - 3	1
UBI	4 - max. blocks	1

For such complex paritioning schemes following usage variants are possible:

- Seperate UBI instances on the UBI partitons.
- Concatenation of the UBI partitions to a single MTD device and one UBI instance of top of the concatenated device.
- No paritioning is necessary, when the boot loader or whatever non movable area of the FLASH can be marked with UBI headers and excluded from wear levelling by marking the blocks as unmovable in the header.

Chapter 4. UBI design

The design consists of two parts. The base design and the enhancements. The base design is simple and straight forward, but has some limitations in terms of scalability. The enhancements are built on top of the base design and are marked accordingly. The enhancements do not change the interfaces and are designed with backward compability in mind.

Eraseblock assignement

Each eraseblock of an UBI volume corresponds to one physical flash eraseblock. Although logical eraseblocks appear contiguous to UBI users, the corresponding physical eraseblocks may be spread over the underlying flash chip(s) in a non ordered way.

Basically UBI uses a table, referred to as *the eraseblock assignement table (EAT)*, which maps logical eraseblocks to physical eraseblocks. When a physical eraseblock is assigned to a volume, the corresponding entry in EAT is changed. Also, if a logical eraseblock is used too actively and UBI decides to assign it to another (less worn out) physical eraseblock, the corresponding entry in EAT is changed as well.

In the basic UBI implementation the eraseblock assignement table is kept and maintained in RAM. The EAT is built during the initial scan of the erase blocks. This does not scale very well, but it is simple to implement and is acceptable for the FLASH sizes used in the first projects. Based on first tests 50us per erase block are required during the scan. The test machine was a 300MHz PPC board on a NAND FLASH device. The per block scan time consists of the FLASH read command time (~ 20us), the read out of the header and the analysis of the data. This sums up to following scan times:

Table 4-1. Scan time versus FLASH size

FLASH size	Erase size	Scan time
64 MiB	16 KiB	200 ms
128 MiB	64 KiB	100 ms
256 MiB	64 KiB	200 ms
512 MiB	64 KiB	400 ms
1024 MiB	64 KiB	800 ms
512 MiB	128 KiB	200 ms
1024 MiB	128 KiB	400 ms

Enhancements

Of course it is possible to implement more comprehensive EAT management methods and to maintain EAT on Flash.

The scan based EAT build must be guaranteed as a fall back option, when a FLASH based EAT is inconsistent or unusable.

UBI manages an internal EAT Volume. The EAT volume is a dynamic UBI volume. For wear-leveling

reasons, the physical eraseblocks which are used for the EAT volume must be changed, but should be kept in an area near the beginning or the end of the FLASH device to limit the number of blocks to scan in order to find the EAT volume.

The challenge of FLASH based EAT management is the balance of consistency, access speed and low FLASH update frequency. A possible solution was identified by the following algortihm. UBI writes an EAT table to flash, which consists of real block references and a number of erase block references which have to be scanned during the initialization of the UBI device. The content of these erase blocks are unknown before they have been scanned. The blocks are references to blocks in the free block pool of UBI at the time the table was written to FLASH. These blocks are the ones which UBI gives out for new usage to UBI clients. During the scan UBI either adds or updates the block references in the FLASH table. The table has to be updated on FLASH when:

- the blocks which are required to be scanned are used up and UBI has to add new blocks from the free block pool to this list.
- larger updates to the table happen, e.g by adding, removing, updating volumes.
- wear levelling requires large modifications of the erase block table.

Eraseblock headers

The basic flash management is based on erase blocks. UBI requires equally sized erase blocks. For block management UBI uses headers at the beginning of each erase block which reduces the usable size of the erase block slightly. The headers are used for two purposes:

- erase count tracking;
- volume identification.

Thus, there are two erase block header types in UBI:

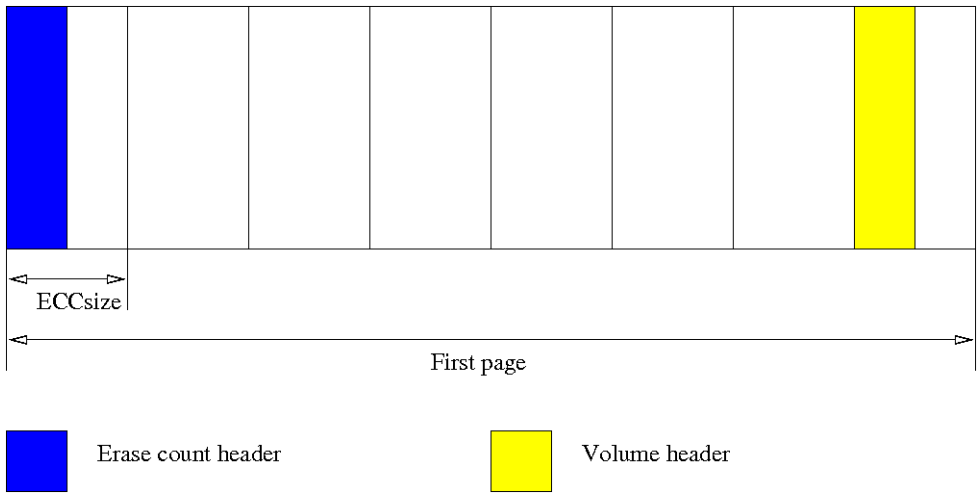
- the erase counter;
- the volume identificator.

Note, the two eraseblock headers are written independently to flash, i.e., UBI requires to perform two write operations to the beginning of an eraseblock. For example, when a physical erase block is not assigned to any volume, it has only an erase count header, but as soon as it is assigned to a volume, the volume identifier must be added.

The eraseblock headers have a fixed position, depending on FLASH type. On NOR the headers are written at the beginning of the erase block on consectutive adresses. On NAND flash devices the erase count header is written to offset 0 of the block without ECC and later the volume header is written to the last subscetion of the first page in the erase block. On small page devices (512 byte / page) this is at

offset 256. On large page devices (2048 Byte/ page) this is at offset (2048 - ECC-Size). The ECC size depends on the hardware / software ECC configuration and is retrieved from the NAND FLASH driver.

Figure 4-1. NAND headers



There are two reasons for this layout on NAND flash.

- Speed up scan time.

  In order to reduce the scan time to identify the erase blocks the proposed layout requires only to read the last ECC-size part of a NAND FLASH page. On large page NAND devices this reduces the number of bytes to read from 2080 to 288 assumed that the ECC size is 256 Byte and the page layout is 8*(256 Byte data) + 8 * (3 Byte ECC).

- Page aligment of the usable FLASH size due to write buffering in file systems.

  NAND flash aware filesystems require page aligned erase block sizes, as writing to NAND FLASH must be page aligned. JFFS2 uses a NAND page sized write buffer to accumulate smaller chunks for writing. Changing this would require non trivial changes which are not considered worth the effort in order to avoid the loss of (NAND page size) - (UBI header size) FLASH space.

General Header format

All UBI headers are written in Big Endian format. This allows removable media to be used on systems with different endianess and all UBI support tools - e.g. for analysis of raw FLASh content, creation of production images - can be written target machine independend

All reserved fields in the header structures are filled with 0 to allow later versions of UBI to handle UBI volumes created by an earlier version.

Erase counter format

```
struct ubi_ec_hdr {
        ubi32_t magic;
        uint8_t version;
        uint8_t padding1[3];
        ubi64_t ec;
        ubi32_t vid_hdr_offset;
        ubi32_t data_offset;
        uint8_t padding2[36];
        ubi32_t hdr_crc;
} __attribute__((packed));
```

magic	The magic number to identify the erase count header. The value of this magic number is 0x55424923, which is ASCII: "UBI#".
version	UBI version number to identify the UBI version which created this header.
ec	The number of erasures for this erase block.
vid_hdr_offset	The position of the volume header in the erase block. This is a constant value depending on FLASH type but it allows to decode raw FLASH dumps without further information.
data_offset	The position of the data in the erase block.
hdr_crc	CRC protection for the erase count header.

Volume identifier format

```
struct ubi_vid_hdr {
        ubi32_t magic;
        uint8_t version;
        uint8_t vol_type;
        uint8_t copy_flag;
        uint8_t compat;
        ubi32_t vol_id;
        ubi32_t lnum;
        ubi32_t leb_ver;
        ubi32_t data_size;
        ubi32_t used_ebs;
        ubi32_t data_pad;
        ubi32_t data_crc;
        uint8_t padding1[12];
        uint8_t ivol_data[UBI_VID_HDR_IVOL_DATA_SIZE];
        ubi32_t hdr_crc;
} __attribute__((packed));
```

magic	The magic number to identify the volume header. The value of this magic number is 0x55424921, which is ASCII: "UBI!".
version	UBI version number to identify the UBI version which created this header.
vol_type	The volume type.
vol_id	The volume id to which this erase block belongs.
lnum	The logical block number of this erase block.
leb_ver	The version number of this erase block.
compat	Flags associated to this erase block.
vol_type	Volume type dependend information.
hdr_crc	CRC protection for the header.

The volume information for static volume contains:

data_size	Data size in this erase block. This field is relevant for the last logical block of a static volume as the last block propably does not contain content up to the usable size of the block. So this field is necessary to prevent access beyond the stored contents boundary.
data_crc	Data CRC for this erase block.
used_ebs	The number of total used blocks in the volume. This information is necessary for consistency checks of a static volume.

The volume information for dynamic volume contains:

data_crc	Data CRC for this erase block

Erase counter update atomicity

When ever an erase block has to be erased the current erase count is kept in RAM and after the erase has completed the incremented erase count is written back to FLASH. When the operation is interupted after the erase block has been erased but before the erase counter has been written completely, the erase counter is lost. When this is discovered later on then the erase counter of the affected block is set to the average erase count of all blocks. Same applies when an erase count header is corrupted.

Note, it might be possible to add a journal to guarantee atomic erase counter updates.

Volumes

UBI volume numbers are in the range of 0 - 65535. The volume number field in the header structures would allow a larger range, but there is no real use case for more volume ids.

Aside of user accesible volumes, UBI maintains internal volumes to store UBI related information e.g. volume information, flash based erase block assignment tables.

The volume numbers 0 - 65279 are available for user volumes. The volume number 65280 - 65535 are reserved for UBI internal volumes. The volume name strings for the internal volumes start with "ubi-".

There are no ordering restrictions for use volumes. The user can choose free volume numbers, e.g. 0,7,8,300. This allows flexible association of volumes for user purposes.

Please note that the currently existing implementation does only support volumes numbers from 0 to 127 for users.

The layout volume

Volume number 65280 (called *layout volume*) contains UBI layout information, i.e. number of volumes, their sizes and properties. The volume name string is "ubi-layout".

The layout volume requires two erase blocks. The data is stored redundantly. The volume information is stored as a consecutive list of volume information structures of all available user volumes.

The volume information ist stored in following structure.

```
struct ubi_vol_tbl_record {
        ubi32_t reserved_pebs;
        ubi32_t alignment;
        ubi32_t data_pad;
        uint8_t vol_type;
        uint8_t padding1;
        ubi16_t name_len;
        uint8_t name[UBI_VOL_NAME_MAX + 1];
        uint8_t padding2[24];
        ubi32_t crc;
} __attribute__ ((packed));
```

The layout volume is updated when a volume is created, deleted or resized. When one of these operations has been performed, a new layout volume is written to FLASH. The version number of the logical erase block (always nr. 0) is increased and when the data has been written sucessfully the original layout volume is erased. The robustness aspects of moving static content erase blocks which are discussed further down apply to this mechanism.

The information about the volumes - reserved, used size - and the free space on the UBI device is accessible via the sysfs interface of the UBI device and the UBI volumes.

Robustness

When the layout volume is defective, then UBI derives the volume sizes from the scan information.

The number of reserved blocks per volume is set to the largest logical erase block number found for each volume.

This allows the system to get into operational state. UBI emits appropriate warnings to the system log and provides status information via the sysfs interface.

FLASH space requirements

Spare erase blocks

UBI needs to keep a number of spare erase blocks to ensure life time operation. These spare blocks reduce the total usable FLASH size.

The minimum requirement of UBI is one spare erase block for wear levelling and update mechanisms.

On some FLASH types e.g. NAND it is required to reserve a number of spare blocks for bad block handling. The minimum number is the number of initial bad blocks. The maximum number is the number of blocks which the system designer wants to reserve in order that UBI keeps fully operational. This is the total number of bad blocks per UBI device. There is no need to take bad blocks into account when calculating the volume sizes. UBI guarantees full functionality up to the configured number of bad blocks.

Bad block parameter

The maximum number of bad blocks is related to the FLASH type and FLASH size so it is a natural choice to extend the MTD info structure with a parameter which can hold the maximum number of bad blocks which are acceptable on a particular device.

This requires also that the mtd concat layer sums up these parameters for concatenated devices.

A command line option to override the device driver value should be provided to make template drivers usable and configurable for a specific hardware.

Robustness

When the number of bad blocks exceeds the specified maximum, then UBI denies write access to all volumes. Manual interaction is necessary. The solution is to increase the maximum number of bad blocks. This is only possible when there is free space on the UBI device. If the complete available space is used, then either volumes have to be deleted or size reduzed before increasing the number of maximum bad blocks.

The actual number of bad blocks is always available through the sysfs interface of the UBI device.

UBI also emits appropriate warning messages to the system log, when the delta of maximum and actual number of bad blocks is less than two.

It’s recommended to monitor the actual number of bad blocks via the sysfs interface of the UBI device with a surveillance application. This allows early information of service personal to ensure device replacement or change of parameters before the maximum number of bad blocks is reached and the device is switched to a restricted operational state.

Space calculation

The available space is calculated via:

$n_{user} = n_{tot} - n_{ubi}$
where
n_{user} = number of erase blocks available to users
n_{tot} = number of physical erase blocks
n_{ubi} = number of UBI reserved erase blocks

The number of UBI reserved blocks is calculated by:

$n_{ubi} = n_{maxbad} + n_{intvol} + 1$
where
n_{maxbad} = number of maximum bad blocks
n_{intvol} = number of blocks used by internal volumes

The number of blocks used by UBI internal volumes is one - used for the layout volume - in the initial implementation. Further enhancements like a flash based EAT will require more reserved blocks for internal purposes. The total number of bytes available for content storage is:

$bytes_{tot} = n_{user} * bytes_{block}$
where
$bytes_{tot}$ = number of bytes available for storage
$bytes_{block}$ = number of bytes available for storage per erase block

The number of bytes available for storage per eraseblock is:

$bytes_{block} = bytes_{phys} - bytes_{ubi}$
where
$bytes_{phys}$ = number of bytes per erase block
$bytes_{ubi}$ = number of bytes reserved by UBI

The number of bytes reserved by UBI is:

For NOR:
$bytes_{ubi}$ = 2 * sizeof(eraseblock header)
For NAND:
$bytes_{ubi}$ = NAND page size

The size of an UBI volume is always a multiple of the number of bytes which are available for content store per erase block.

Volume management

UBIs volume management is based on space accounting. The total number of erase blocks available for user volumes can not be exceeded by the sum of erase blocks which are reserved for the volumes. Even when volumes do not use up the reserved space, UBI does not implement overcommitment mechanisms.

UBI implements three functions related to volume management:

- Volume creation
- Volume deletion
- Volume resizing

Volume creation

A new volume can be created as long as there is enough free space available. The creator requests the size of the new volume and UBI aligns it to a multiple of bytes per erabe block which are available for content storage.

In order to support a block device layer on top of UBI it is necessary to provide an optional parameter which can adjust the minimum unusable space - which is 2 * sizeof (ubi header) to a greater size which ensures that the usable space of an erase block is a multiple of the block device sub block size.

Volume deletion

An existing volume can be deleted, when it is not in use. The erase blocks which were allocated for the volume are erased and put back into the free block pool. The erase operation must be completed before the volume resize can be propagated into the layout volume.

Volume resizing

An existing volume can be resized, when it is not in use.

In case the volume is expanded the required space must be free in the ubi device.

When the volume size is reduced, then the number of used blocks in the volume must be less or equal than the new size. Note, that by default UBI will not reduce the size of dynamic volumes unless the number range of logical erase blocks which will be removed from the volume contains only unused blocks. A specical parameter allows to override this default. UBI then moves the contents of logical blocks from the number range which will be removed from the volume to unused logical erase blocks with lower numbers before resizing the volume. The move operation must be completed before the volume resize can be propagated into the layout volume.

Logical blocks

An UBI volume consists of logical blocks. The maximum number of logical blocks for a volume is specified by the size requirements given at volume creation or resizing.

The logical blocks associated to a volume can hold the logical block numbers 0 to maxblocks-1. The logical block numbers of a volume are unique.

For short periods of time block moving, scrubbing and wear levelling functions can create a second instance of a logical block number. UBI ensures that write access to such blocks is not possible.

Block management

Block allocation

UBI uses late block allocation to ensure optimized wear levelling. A block is allocated and the volume header is written to it when a write access happens to an logical block number which has not been allocated yet. All erase blocks which are not allocated for any volume are kept in a free block pool.

Block erasure

Block erasure is triggered by:

- erase request from an UBI client
- volume deletion
- volume content update
- internal block move operations due to wear levelling
- internal block move operations due to scrubbing

In most cases the block erasure is an asynchronous operation and is deferred to a background operation. Synchronous block erasure requirements are marked in the sections of the functionality which requires them.

Block content movement

Block content movement can be triggered by:

- wear levelling
- bit flip correction (scrubbing)
- volume resizing

In most cases the movement of block contents is an asynchronous operation and is deferred to a background operation. Synchronous requirements are marked in the sections of the functionality which requires them.

When UBI decides to move a logical eraseblock to another physical eraseblock due to wear-leveling or scrubbing considerations, it needs to:

- move the contents of the previous physical eraseblock to the new physical eraseblock;
- update the corresponding entry in the EAT table.
- erase the old erase block

Static volumes

The update mechanism for blocks of a static volume is simple. The volume header of the erase block is copied and the version number of the erase block is incremented. The header CRC is updated, but the data CRC is unchanged. Then the block content is copied to the new block. When the contents have been copied completely, the original erase block is erased.

In case the copying of block contents is interrupted before all data has been transferred, then UBI detects two versions of the same logical block in a volume. UBI tries to use the one with the higher version number first. Due to the interruption of the copying the data CRC is invalid and UBI falls back to the block with the lower version number. The incomplete block is erased and a new move operation can be started.

In case the moving operation is interrupted before the original erase block has been erased, then UBI detects two versions of the same logical block in a volume. UBI tries to use the one with the higher version number first. In this case the data CRC of this block is valid and the block with the lower version number can be erased.

The moving operation is not blocking the read access to the data contents. Until the contents are completely copied the data can be read from the original erase block.

Dynamic volumes

For dynamic volumes a slightly different mechanism is necessary. UBI blocks write access to the logical erase block for the move process. A CRC checksum is generated over the complete data area of the block. This checksum is written to the volume header of the new erase block along with an incremented version number of the logical block and an updated header CRC. Then the block content is copied. Note, that empty content (containing all 0xff) is not copied, but accounted in the data CRC. When the block content has been copied completely then the original erase block is erased and the write access to the logical block is enabled again.

In case the copying of block contents is interrupted before all data has been transferred, then UBI detects two versions of the same logical block in a volume. UBI tries to use the one with the higher version number first. Due to the interruption of the copying the data CRC is invalid and UBI falls back to the block with the lower version number. The incomplete block is erased and a new move operation can be started.

In case the moving operation is interrupted before the original erase block has been erased, then UBI detects two versions of the same logical block in a volume. UBI tries to use the one with the higher version number first. The data CRC is valid, so the block with the lower version number can be erased. Until the original block has been erased write access to the logical block has to be disabled.

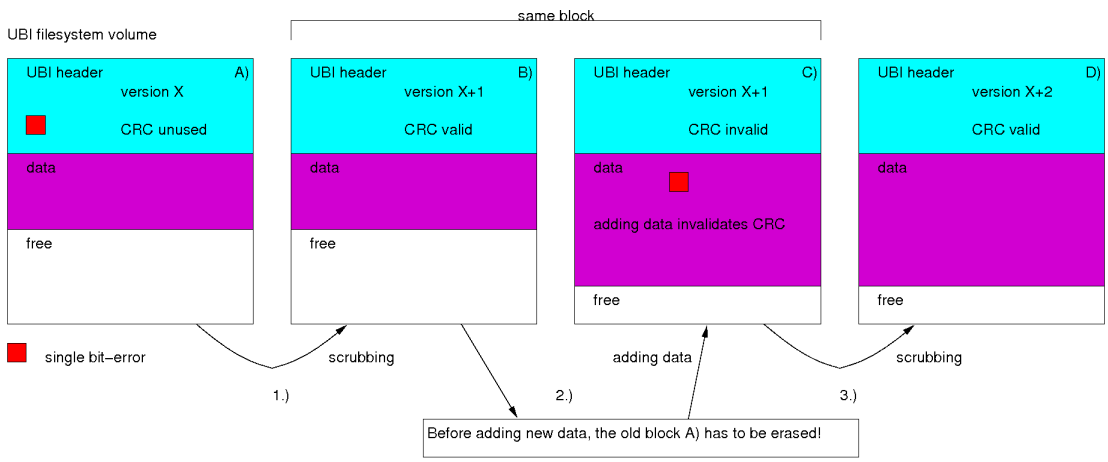
The moving operation is not blocking the read access to the data contents. Until the contents are completely copied the data can be read from the original erase block. Write access to an logical block which contents are moved is not allowed until the moving operation has been finished completely, i.e. the original erase block must have been erased before write access can be granted again. Write access to other erase blocks of the volume is allowed during a moving operation.

Scrubbing

Scrubbing is a safety mechanism to handle bit flips. When a bit flip occures and the ECC algorithm can recover the original data contents, then the contents of the affected block are moved to a free block according to the algorithms described above. The scrubbing operation is scheduled in the background, when a in kernel or user space initiated read operation detects a bit flip.

The following figure shows how an UBI block of a file-system volume is treated, when a bit-error has occurred. UBI block A) needs to be copied into a free block to eliminate the bit-error. The version number is increased and the CRC is calculated and added to the new UBI header. Because the data of this particular kind of UBI block is controlled by the file-system using it, the CRC is usually unused (like A)). But when the block is scrubbed the data CRC is calculated to be able to identify the latest valid version of the block when the copy/erase process should have been interrupted.

Figure 4-2. Scrubbing of an UBI block in a filesystem volume



- Imagine block A) and B) exist at the same time. Here the block with the highest version number should be kept if and only if the data CRC is valid. The existence of both blocks can happen due to an interruption before or while A) is deleted. In the latter case block A) has undefined data but B) is still valid.

- The transformation from block B) into block C) happens when the file-system needs to add more data to the non-empty block. In this case the data CRC becomes invalid. When two blocks exist only the newer blocks data CRC is to be checked. This can be seen in the next step.
- Over the time a bit-error occurred in block C) which needs now to be copied to a new block D). This case is the same like the first one. The data CRC is now calculated, the version number is increased by one and the new block D) is written. If interrupted C) and D) exist at the same time and the CRC of the newer block D) is used to determine its validity. If D)'s CRC is invalid C) is used.

The displayed algorithm ensures that in any cases where the scrubbing process can be interrupted a valid copy of the UBI block exists can be identified by the block's version number and/or the associated data CRC.

UBI Initialization

The first simple UBI implementation performs full flash scan in order to find out positions of layout volume eraseblocks, to build the EAT table. When UBI scans flash media, it reads eraseblock headers and builds all the necessary data structures in main memory.

The scan based mechanism requires seperate scans in the boot loader and the kernel. It would be technically possible to hand the boot loader scan table to the kernel, but this would require a more unified boot loader kernel interface across the different architectures and boards. Such an interface is not available and therefor such an attempt would be a single purpose hack rather than a useful generic solution.

Future enhancements

In order to avoid scanning of the complete FLASH it is necessary to implement a flash based EAT mechanism.

Background operation

UBI uses a background thread or utilizes the available background Linux worker threads for following operations:

- Wear levelling
- Block moving
- Block scrubbing
- Block erasure

Wear leveling

Wear levelling is done across the complete flash device with help of the erase counters.

The default selection criteria is to use the blocks with the lowest erase count.

UBI keeps track of the maximum and minimum erase counts over all erase blocks. To avoid permanent wear levelling activities a treshold of 5000 erase cycles is introduced. Wear levelling is checked when ever a block has been erased. When the newly erased block is the block with the highest erase count and the delta of erase cycles versus the block with the lowest erase count has reached the treshold then UBI selects a long term storage block with a low erase count and copies the block contents to the block with the high erase count using the block moving function. When copying is done the block with the low erase count is erased and enqueued into the free block pool. In case of a very low number of free blocks this may slow down the allocation of erase blocks for UBI clients.

The free - unallocated - erase blocks are kept in a sorted list in erase count order to ensure O(1) selection of a block when allocation is requested.

Enhancements

An UBI client might provide a callback function, which is called by UBI before trying to move block content. The rationale of the function is, that the client might veto the movage of the block contents. One reason for this might be that the client has this block in use for writing or has the block scheduled for garbage collection. In such a case the moving of the block content makes no sense and can be deferred to a later time or the block is made available by garbage collection in the near future.

A function to signal the intended use of a logical erase block can be implemented. An UBI client could signal whether the block is used for long term or short term storage. This would allow to implement more efficient wear levelling algorithms.

Volume update

Update of UBI volumes happen from userspace. The update mechanisms writes raw data contents without meta information. The meta information - i.e. Volume headers - are created by UBI on the fly.

The update tool opens the ubi volume and issues a start_update ioctl command. The argument of start_update is the total number of bytes, which will be written to the volume. This is necessary to provide robustness versus interruption of the update and to calculate the number of total used blocks for static volumes. UBI checks whether the requested data length fits into the volume.

In case of updating a dynamic UBI volume, first a block is written with a volume update header containing the reserved VOLUME_UPDATE volume id and the logical block number 0. The number of the volume which is updated is stored in the data payload of this block. The existence of this block signals the update in progress. UBI erases the eventually existing blocks which are assigned to the volume. The erase of the existing blocks has to be completed before new content can be written.

When a static volume is updated, writing a volume update marker is not necessary. The completeness and correctness of the data is ensured if all used erase blocks are found and their header and data CRC is correct.

Now the data content is written and UBI creates the volume headers for the logical erase blocks and calculates the data CRC in case of a static volume.

When all data of the dynamic volume has been written successfully, then the update volume block is erased.

Note, that a size of 0 bytes given to the start_update command is valid. This creates the update in progress block and erases the existing logical erase block. The update in progress marker is removed after all existing blocks have been erased. This is useful to empty a volume.

When a volume update is interrupted then the block with the reserved update volume id is on the FLASH and invalidates all other eventually existing logical erase blocks for the volume which is protected. A new update has to be started. A possible UBI client which tries to access such a volume is informed by an appropriate error code and the access is denied. See also "Programming Interfaces". In the volume layout the volume still exists, but the number of used blocks is zero. UBI does automatically erase the eventually existing blocks of such an volume, but the block indicating volume update in progress is not erased. User interaction is necessary to resolve this. The update in progress block can be removed by a new update command or by deleting the volume.

The usage of an extra erase block to indicate an update in progress looks like a waste, but for dynamic content volumes there is no other way to ensure the completeness of an update operation before allowing the usage of such a volume. This might change once the enhancements of a journal / super block have been implemented. On the other hand the update in progress marker is perfectly reasonable because such updates happen rather seldom during the lifetime of a device and it allows a simple scan based identification of volume completeness e.g. in a boot loader.

Note, that the extra block is an per UBI device overhead which is reserved anyway for wear levelling and block move purposes. This limits the number of concurrent volume updates to one. Thats not a big problem as volume updates are usually done in a code update situation where time is not the critical factor. Also the limiting factor of an update is the FLASH device itself which allows no erase / program operations in parallel.

Partial updates of volumes are not possible.

Volume readout

The data contents of an UBI volume can be read out from user space. A read out can only be performed on unused volumes or on volumes with read only client.

Chapter 5. Programming Interfaces

Kernel Space

- read
- write
- erase
- get_volume
- put_volume
- Enhanced: move_modify

The kernel space interfaces access the volumes by logical block number and offset inside of the logical block rather than providing a full offset which would have be converted into logical block number and offset inside of the block for following reasons.

A FLASH aware file system has to be aware of erase block boundaries anyway and usually operates on blocks. The currently used MTD interface which requires full offsets is called by adding the in block offset to the block base offset. So the seperate information is available anyway. Of course could UBI deal with the full offset, but due to the fact that UBI reduces the net data size of an erase block this would require a (64bit offset) / (usable block size) division. 64 / 32 bit divisions are perfomance critical on 32 bit CPUs. Otherwise there would be an limitation of 32 bit offsets, which would limit UBI to 4GiB maximum device size.

The implementation of a block device on top of UBI will divide the UBI erase block into smaller sized blocks which are exposed to the block device interface. There is an explicit requirement that the size of the usable UBI block size must be a multiple of the size of these sub blocks. The size of the sub blocks is also influenced by the NAND page size, when the block device is writeable. The minimum sensible sub block size will be 512 byte, which allows the usage of pure 32 bit operations to calculate the logical block number. This increases the device limit to 2 Terrabyte which seems to be a reasonable limitation.

read

The read function takes following arguments:

- volume descriptor
- block number
- offset
- length
- buffer pointer

The return value is the number of bytes read or an error code.

When an UBI client of a dynamic volume wants to read an unallocated logical erase block, then UBI fills the requested number of bytes in the data buffer with 0xFF without accessing the FLASH chip.

The following error codes can be returned:

-EIO	A fatal I/O error occured, e.g. device removal, timeouts.
-EINVAL	An invalid block number, offset or length was requested.

UBI does not provide a read_ecc version because this is not relevant for the UBI client and the ecc related problems are handled inside UBI itself.

write

The write function takes following arguments:

- volume descriptor
- block number
- offset
- length
- buffer pointer

The return value is the number of bytes written or an error code. When the offset is 0 and no physical erase block has been assigned to the logical erase block then UBI assignes automatically a physical erase block and writes the volume header. This function is only available for dynamic content volumes.

The following error codes can be returned:

-EIO	A fatal I/O error occured, e.g. device removal, timeouts.
-EROFS	Attempt to write to a read only volume.
-EINVAL	An invalid block number, offset or length was requested.

The write operation is synchronous. UBI guarantees that data has been written to FLASH when the funtion returns without an error.

Write errors which are detected by the hardware level driver - e.g. write-verify of the NAND driver - are handled by UBI before returning from the write function. In such a case UBI moves the existing block contents to a free erase block according to the block movement mechanisms and writes the data in the data buffer to this new block when the block content move operation has been completed.

UBI does not provide a write_ecc version because this is not relevant for the UBI client and the ecc related problems are handled inside UBI itself.

erase

The erase function takes following arguments:

- volume descriptor
- block number

The erase function schedules the logical erase block for erasure. UBI ensures that the physical erase block has been erased before the next write access to the same logical block number, which allocates a new physical erase block. This function is only available for dynamic content volumes.

The following error codes can be returned:

-EIO	A fatal I/O error occured, e.g. device removal, timeouts.
-EROFS	Attempt to erase a block on a read only volume.
-EINVAL	An invalid block number was requested.

get_volume

The get_volume function takes following arguments:

- ubi device id
- volume id
- mode read/write - readonly

The function must be called before any of the other functions can be used. The function marks the volume as used. Only one user per volume is allowed.

On success the function returns a pointer to the UBI volume description structure. Otherwise an error code is returned.

The following error codes can be returned:

-ENODEV	The requested volume does not exists.
-EROFS	Attempt to get R/W access on a read only volume.
-EBUSY	The volume is already in use.
-EUCLEAN	The volume is corrupted.

put_volume

The put_volume function takes following arguments:

- volume descriptor

The function is called to finish the usage of a volume. The function marks the volume as unused.

Enhanced: move_update

The move_update function takes following arguments:

- volume descriptor
- block number
- offset
- length
- buffer pointer

The function moves the contents of a logical erase block to a free block using the basic block moving algorithm of UBI. The block moving algorithm has to be extended to allow the update of the block content on the fly.

This function allows to implement robust update of block contents, which is a preliminary for r/w block device emulation.

Userspace Interface UBI

The user space interface is implemented in sysfs. Device nodes for I/O operations have to be created by the udev mechanism in userspace.

The sysfs interface provides mostly read only information about the UBI device and the UBI volumes. The operational functions are accessible via character devices, which have to be created by the udev mechanism.

The sysfs interface exposes all status information about the UBI device and the volumes in a human readable way. This has several advantages over ioctl based binary information exchange. The sysfs interface allows simple script based analysis and monitoring and there is no need to keep changes of data structures between kernel and userspace synchronized.

The enumeration of available volume ids is a simple

‘ls /sys/devices/ubi/0/volumes‘

command.

sysfs interface

Deviceinfo (sysfs)

- nr. volumes
- nr. bad blocks

- nr. used blocks
- nr. free blocks
- blockbitmap
- stats - erase count statistics, bitflips
- debug switch

Volumeinfo (sysfs)

- name
- size
- type/flags
- used blocks
- blockbitmap
- status

Examples

/sysfs/devices/ubi/0/device_info/nr_volumes
nr_used
nr_free
bitmap
erase_stats
device - mtd
name
volumes/0/used_size
bitmap
...

UBI device I/O functions

The UBI device provides following ioctl functions:

- create_volume
- delete_volume
- resize_volume

create_volume

The create_volume ioctl takes following arguments:

- volume id
- volume type
- volume size
- volume name
- minimum subblock size

The volume type is either dynamic or static. The minimum subblock size is either 0 to select the system default or a value to adjust the subblock size for UBI clients e.g. a block device layer. See also section "Volume creation" in the chapter "UBI design".

The create_volume ioctl is a synchronous operation.

The volume must not exist and the size of the volume must fit into the free size of the ubi device.

The following error codes can be returned:

-EEXIST	The volume exists already
-EINVAL	An invalid parameter was given.
-ENOSPACE	The requested size is not available on the device.

delete_volume

The delete_volume ioctl takes following arguments:

- volume id

The volume must exist and not be in use.

The delete_volume ioctl is a synchronous operation. After returning without an error the volume and all its data is deleted in flash.

The following error codes can be returned:

-EBUSY	The volume is in use and can not be deleted.
-EINVAL	An invalid parameter was given.

resize_volume

The resize_volume ioctl takes following arguments:

- volume id

- volume size

The volume must exist and not be in use. In case the volume is expanded the required space must be free in the ubi device. When the volume size is reduced, then the number of used blocks in the volume must be less or equal than the new size.

The following error codes can be returned:

-EBUSY	The volume is in use and can not be resized.
-EINVAL	An invalid parameter was given.
-ENOSPACE	The requested size is not available on the device.

UBI volume I/O functions

The UBI volume provides following ioctl functions:

- start_update

The UBI device provides following file I/O functions:

- open
- llseek
- read
- write
- close

start_update ioctl

The start_update ioctl takes following arguments:

- data size

The data size must be less or equal to the reserved size of the volume. When the size is valid then the existing erase blocks of the volume are erased. This is a synchronous operation to ensure that no remains of the previous volume are on FLASH before writing the new image data. This call is necessary to enable write access to the volume through the write file I/O function.

The following error codes can be returned:

-EBUSY	Another operation is active so the volume can not be updated.
-EINVAL	An invalid parameter was given.

-ENOSPACE The requested size is not available on the device.

open

The open interface is the standard open system call. A file descriptor is returned on success.

The following error codes can be returned:

-EBUSY The volume is in use and can not be opened.
-ENODEV The volume is not available.

llseek

The llseek interface is the standard lseek system call.

The following error codes can be returned:

-EINVAL An invalid parameter was given.
-EUCLEAN The volume is corrupted.

read

The read interface is the standard read system call. The volume data content is read in the order of logical erase blocks. The start offset can be set by the llseek function.

The following error codes can be returned:

-EINVAL An invalid parameter was given.
-EUCLEAN The volume is corrupted.

write

The write interface is the standard write system call. The volume data content is written in a consecutive binary stream. The write call which contains the last byte of the stream is a synchronous operation to ensure that the complete data is written to FLASH.

There are two write modes possible

- Volume update write mode
- Dynamic volume write mode

Dynamic volume write mode is an enhancement. Volume update write mode is a basic requirement.

Volume update write mode

Volume update is available for static and dynamic volumes.

The start_update ioctl is called immidiately before the write. The complete volume data has to be written in consecutive order. The write is terminated when the size of data which was announced via the start_update ioctl has been written.

The following error codes can be returned:

-EINVAL An invalid parameter was given.

Enhancement: Dynamic volume write mode

This write mode is only available for dynamic volumes.

The application has to llseek to the offset which should be written to. The application has to be aware of basic FLASH handling restrictions. Writes to non empty locations are not allowed. Writes accross logical erase block boundaries are not allowed.

The following error codes can be returned:

-EINVAL	An invalid parameter was given.
-EUCLEAN	The volume is corrupted.
-PERM	Returned when a write on a static volume without a previous update command was requested.

Chapter 6. MTD modifications

The MTD code needs some small modifications to provide optimal UBI support.

Maximum bad block parameter

As mentionened above the mtd info structure has to be extended to provide a parameter which informs UBI about the maximum number of bad blocks which UBI has to handle. This reduces the usable FLASH space as UBI has to reserve backup blocks for those.

Chapter 7. UBI kernel clients

JFFS2

JFFS2 needs some modifications to be used on top of UBI. The major changes are:

- interface wrapping
- block erasure
- bad block handling
- write failure handling

Interface wrapping

Most of the JFFS2 MTD access functions are wrapped into macros anyway, so only the macros need to be replaced. The few remaining open coded mtd access function have to wrapped into the existing macros.

The interface wrapping substitutes the mtd structure by an UBI structure, so JFFS2 has to be configured for use on top of UBI at compile time. Runtime switching is not necessary and would be hard to implement.

Block erasure

JFFS2 on top of MTD handles erasure of blocks itself. The necessary changes are quite small. Instead of queueing a block to the erase(pending) list, JFFS2 calls the UBI erase function and schedules the block for erasure. The block is immidiately put back to the free block list. UBI guarantees that the logical block number can not be reused until the previous block has been erased. That means a write to a logical erase block scheduled for erasure is blocked until the erasure has taken place. To avoid that this happens when JFFS2 is not under space pressure it is required to enqueue the block at the end of the free block list and pull blocks for usage from the top.

Bad block handling

JFFS2 does not need to keep track of bad blocks anymore. The bad block handling is done in UBI and the bad block related functions of JFFS2 can be deactivated when used on top of UBI.

Write failure handling

The JFFS2 functions related to write failure handling can be kept in place for the time being, but those functions will not be invoked as UBI will handle the write failure recovery.

Enhancements

Further enhancements to JFFS2 are possible. UBI knows due to late allocation which blocks are used by JFFS2 and which blocks are empty. This information is available to JFFS2 and can be used to build the free block list without scanning the blocks.

Enhancement clients

Block device emulation

Block device emulation layers on top of UBI benefit from the generic UBI infrastructure:

- Bad block managment
- Wear levelling
- Erase block assignement infrastructure

Static read only block device

A read only block device emulation on top of a static UBI volume is rather simple to implement. The data contents are readable via the in kernel API and the block emulation has only to provide the basic functionality.

Read/write block device

A read/write block device emulation on top of a dynamic UBI volume is easier to implement than a block device emulation on top of a raw FLASH device. UBI provides a lot of basic functionality to the client already.

Still there has to be the implementation of the cached write with respect to the necessary erase function, but UBI provides the functions for this already and the above discussed enhanced move_modify kernel interface allows an implementation which does not lose a complete logical erase block on interruption.

Chapter 8. Booting with UBI

This chapter describes the the general considerations to be made when booting a UBI-aware system. In particular it describes the impacts of UBI on the boot loader design.

When processors are powered on, they need to be initialized and brought into a state where more complex software, like an operating system, can be executed. The environment used for boot-strapping is mostly very limited and its resources are at a location predetermined by the processor architecture.

In traditional embedded systems usually NOR FLASH was used to store the boot-strap code, which needs to be mapped to contiguous memory range. NAND FLASH does not provide automatically a contiguous memory range for the first couple of instructions needed for the boot-strap process. Instead a special hardware needs to be added as a macro on the chip itself, e.g. the NDFC used in the PowerPC 440EP, an external solution using an FPGA, or exploiting vendor specific solutions like Samsung OneNAND or M-System DiskOnChip.

Important is to note, that even in a UBI-aware system, the FLASH erase block containing this initial boot-strap code, must not be moved, like UBI does with any other flash erase blocks. Particular care needs to be taken when this memory is deleted, or updated, because when this is interrupted, the system might not be able to boot anymore. To overcome this limitation the hardware design must provide watchdog mechanisms used to switch to a different memory location if the predefined memory is accidently invalid.

NOR FLASH system

The erase-blocks containing the boot-strap code need to be fixed, by using a non-UBI partition. The system design can reserve here as many blocks as needed for boot-loaders like U-Boot, Redboot or similar. The FLASH update of those special areas is to be implemented differently than the update of a UBI based volume.

Table 8-1. Partitioning for a NOR system

Partition	Physical erase blocks
Boot-strap code	0 - 4
UBI	5 - max. blocks

The erase blocks excluded from UBI are not due to wear-leveling or scrubbing. If bit-errors occur or the blocks need to be updated, the system must not be turned off. If the system should be able to recover from power-loss or interruption, it must provide system specific hardware mechanisms with a watchdog and a method to use a redundant copy of the boot-code, stored at a different location.

To be able to load the operating system from a static UBI volume the code must contain a UBI scanning procedure as well as a UBI loader.

NAND FLASH system

The FLASH erase block used for storage of the boot-strap code is usually NAND erase block 0. This

block is guaranteed to be a good block by the NAND chip vendors.

A PowerPC 440EP with a build in NDFC provides 4 Kbyte contignous memory for boot-strapping. A device using a Samsung OneNAND chip offers 2 KByte memory and a M-Systems DiskOn Chip comes with 1 KByte. Some ARM chips have on-board flash usable for processor boot-strap. The sizes are varying from 16 to 32 Kbyte, depending on the processor.

Because in NAND based systems contignous memory for the initial code is limited, complex boot-code needs to be split into two parts. Those parts are referred to as IPL (initial program loader) and SPL (secondary program loader). It is beneficial to store the SPL in a static UBI-volume, which can even be redundant, allowing a safe update procedure and gaining the advantages of UBI. The user does not need to make us his mind about wear-leveling, bad block handling and scrubbing for this code.

It is possible to scan a NAND chip for UBI data and do error correction using ECC in about 2 to 3 KByte.

Instead of using an SPL, the system designer may consider using an operating system image directly, if he is able to do the necessary system initialization in the IPL.

In this system NAND block 0 needs to be defined at a fixed location.

Table 8-2. Partitioning for a NAND system

Partition	Physical erase blocks
Initial Program Loader	0
UBI	1 - max. blocks

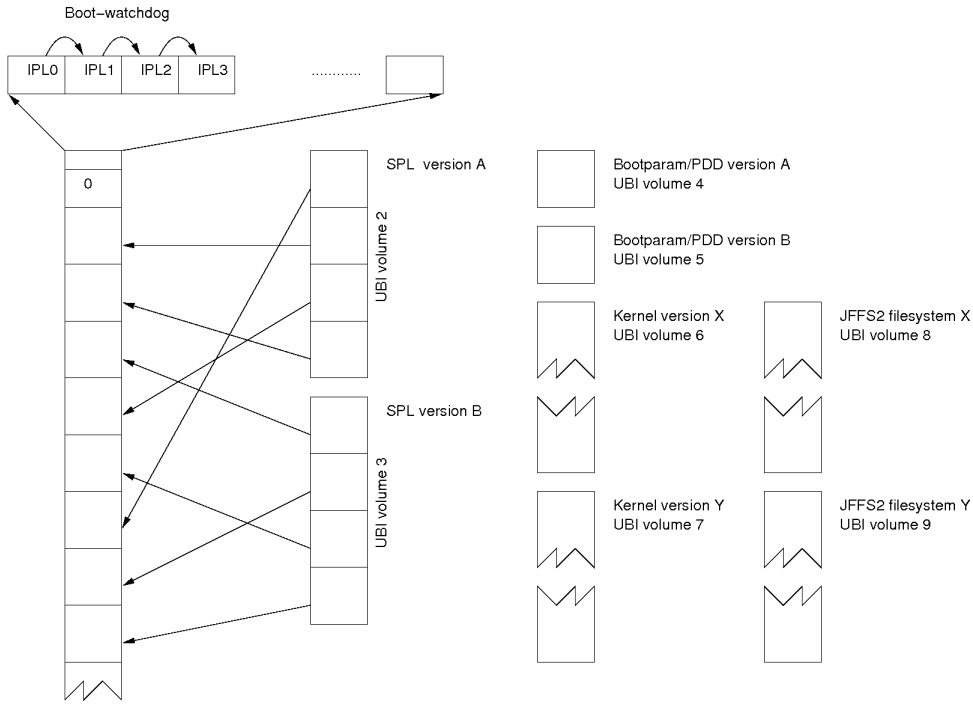
IPL and SPL

We assume that most NAND implementations will use a split of IPL and SPL. Instead of using an SPL we think that some size restricted designs will choose to load the OS image directly. The following examples should illustrate how a systems with redundant SPL and OS images may be setup.

Example volume layout

The example layout shows the fixed erase block 0. Block 0 contains redundant versions of the IPL, selected by a watchdog mechanism. The UBI volume numbering starts here at 2 used for the first redundant copy of the SPL. The second copy of the SPL is stored in volume 3. Volume 4 and 5 are used for SPL configuration data like boot-parameter or platform description data (PDD). Volumes 6 and 7 are used for the operating system binary image. All of the listed UBI volumes are static. Subsequent volumes e.g. 8, 9 and 10 can be used for file systems. Those would be dynamic UBI volumes.

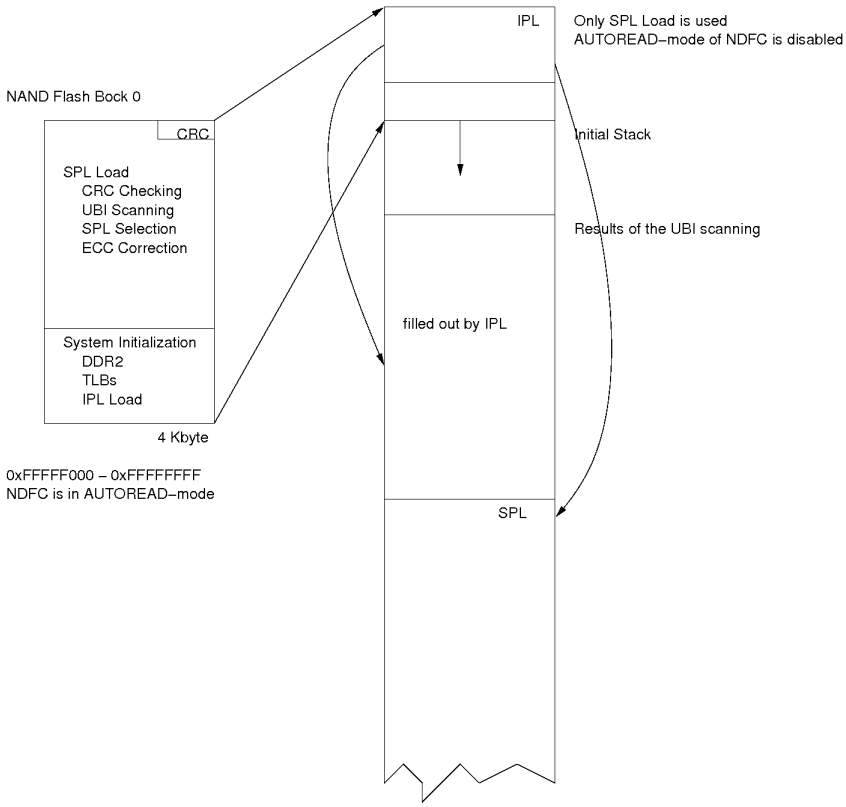
Figure 8-1. Example volume layout



IPL operations - Example: PPC44x using NAND

When using a PowerPC 440EP with an NDFC, the IPL size is restricted to 4 Kbyte. Initially the NDFC is operating in the so called AUTOREAD-mode presenting the 4 Kbyte IPL code at the memory range from 0xfffff000 to 0xffffffff. The IPL needs to boot-strap itself either into iCache or into DRAM because the AUTOREAD-mode needs to be disabled for the UBI-scan and when loading the SPL/OS-image.

Figure 8-2. IPL operations



The first part of the IPL executed via AUTOREAD-mode using the NDFC needs to do the following tasks:

- Basic system initialization
- RAM initialization
- TLB setup
- Load the IPL from NAND into RAM

The second part of the IPL run form RAM. Because the AUTOREAD-mode must be disabled to load the SPL UBI volume, it is required to load the IPL code to a different location. This can be SRAM, DRAM or even the iCache is possible. The code running from RAM is responsible for the following tasks:

- CRC checking of the uploaded IPL. Because the ECC of the IPL is not checked automatically by the hardware, it is important to verify if the uploaded code is good. If this is not the case the system is in an unknown state.
- UBI volume header scanning, and building up the scan table required to find the operating system image.
- SPL selection. It is possible to use more than one copy of the SPL for reliability reasons. In most systems using just one should be fine too.

- Loading SPL with ECC correction. If the ECC needs to be corrected, the UBI layer in the OS must take care that this erase block is copied to a new block and the ECC failure is corrected. Most hardware, like the 440EP implementation can correct one bit failures and detect two bit failures. The system design must ensure that bit flips are corrected as soon as possible to prevent a situation where the code required for booting becomes invalid.

Since the IPL has to scan the flash to find the UBI blocks of the SPL it makes sense to pass the scanning information to the SPL to save some boot-time.

UBI support in boot-loader

The boot-loader must be extended to handle UBI managed FLASH devices. The implementation might not provide full fledged UBI support, but basic functionalities have to be implemented.

- Device scan
- Read volume contents
- Update volume contents (optional)

Device scan

The scan of an UBI device is necessary in the initial boot loader to be able to load the next step boot-code or operating system using an UBI volume. The result of the initial scan result is recommended to be passed to the SPL to speed up the boot-process.

The scan of an UBI device is either performed by a full FLASH scan or when available by analyzing a FLASH based erase block translation table. Since the FLASH based erase block table does not need to exist or be valid, a full scan is always be possible. This was one of the UBI design requirements.

A full scan reads the headers of all FLASH erase blocks and creates a list of the available UBI volumes. The scan can be performed in 3 steps. First each header is read into an RAM array and checked for the availability of the UBI magic number and consistency of the header CRC. The number of erase blocks per volume is accounted. This information is used to create a table which contains the volume offsets in the final load order table. This table is created in the last step of the scan. For each found volume the erase block numbers of the image parts is stored in load order.

To optimize the scanning time reading out the OOB area, for a system using NAND, can be omitted. The information if the data is valid is concluded from the correctness of the data CRC.

Handling of FLASH based erase block translation tables is not described yet as the design of this mechanism in UBI is not done at the moment.

The result of the scan in the SPL or Initial boot loader can be stored in a RAM based structure which lists all found volumes, the number of used blocks per volume and the references to the logical blocks of each volume in load order.

Scan result table

In an IPL/SPL implementation the information is generated by the IPL, because it needs it to load the SPL. The SPL can reuse the table to load e.g. different versions of the operating system image.

read volumes

Boot-code support for reading UBI volumes can be restricted to static volumes to keep it as simple as possible. Reading the raw contents of dynamic volumes is possible of course, but usage of the content requires detailed knowledge of its data structures.

The read support for static volumes is used to load binary images e.g. kernel, initrd or the boot-parameter data blob from FLASH into RAM.

The read support should be aware of interrupted updates and block moving operations. In case of an interrupted update the volume content can not be retrieved. In case of an interrupted block moving operation the robustness mechanism described in the UBI design document apply.

write volumes

Write support for UBI volumes requires to implement following functionality:

- Bad block awareness
- Analyzing the layout volume
- Erase count preserving block erasure
- UBI aware writing of logical erase blocks
- UBI aware updating of static volumes
- Awareness about the reserved blocks