



Home

FAQ

Mailing Lists / IRC

Source

Documentation

Misc

Archive

Fellows

Memory Technology Devices

General

NAND

OneNAND

JFFS2

UBI

UBIFS

UBIFS FAQ and HOWTO

Table of contents

1. [How do I enable UBIFS?](#)
2. [How do I mount UBIFS?](#)
3. [How do I create an UBIFS image?](#)
4. [May an empty UBI volume be mounted?](#)
5. [What is the purpose of -c \(--max-leb-cnt\) mkfs.ubifs option?](#)
6. [Why do I have to use ubiformat?](#)
7. [What is the the purpose of the -F \(--space-fixup\) mkfs.ubifs option?](#)
8. [How do I compile mkfs.ubifs?](#)
9. [What is "favor LZO" compression?](#)
10. [Can UBIFS mount loop-back devices?](#)
11. [How do I change a file atomically?](#)
12. [Does UBIFS support atime?](#)
13. [Does UBIFS support NFS?](#)
14. [Does UBIFS become slower when it is full?](#)
15. [Why df reports too few free space?](#)
16. [Why does my UBIFS volume have significantly lower capacity than my equivalent iffs2 volume?](#)
17. [How do I disable compression?](#)
18. [How do I use UBIFS with nandsim?](#)
19. [How do I extract files from an UBI/UBIFS image?](#)
20. [I need more space - should I make UBIFS journal smaller?](#)
21. [Why is my file empty after an unclean reboot?](#)
22. [Why does my file have zeroes at the end after an unclean reboot?](#)
23. [What does the "ubifs_bgt0_0" thread do?](#)
24. [UBIFS suddenly became read-only - what is this?](#)
25. [How do I detect if UBIFS became read-only?](#)
26. [I get: "validate_sb: LEB size mismatch: 129024 in superblock, 126976 real"](#)
27. [I get: "INFO: task pdf flush:110 blocked for more than 120 seconds"](#)
28. [I get: "init_constants_early: too few LEBs \(12\), min. is 17"](#)
29. [I want to study UBIFS - any recommendations?](#)
30. [How do I debug UBIFS?](#)
31. [How do I send an UBIFS bug report?](#)

How do I enable UBIFS?

Since UBIFS works on top of UBI, you have to enable UBI first (see [here](#)).

Then in the Linux configuration menu, go to "**File systems**" -> "**Miscellaneous filesystems**", and mark the "**UBIFS file system support**" check-box. UBIFS may be either compiled into the kernel or be built as a kernel module.

How do I mount UBIFS?

The modern way of mounting UBIFS is mounting UBI volume character device nodes, e.g.:

```
$ mount -t ubifs /dev/ubi0_0 /mnt/ubifs
```

will mount UBIFS to UBI volume 0 on UBI device 0. This is the easiest way to mount UBIFS, but it is supported only in kernels starting from version 2.6.32. The UBIFS back-port trees (see [here](#)) also support this mounting method.

The old method is to use device-less mount, just like `procfs` or `sysfs`. The volume to mount is specified using `ubiX_Y` or `ubiX:NAME` syntax, where

- X - UBI device number;
- Y - UBI volume number;
- NAME - UBI volume name.

For example,

```
$ mount -t ubifs ubi1_0 /mnt/ubifs
```

mounts volume 0 on UBI device 1 to /mnt/ubifs, and

```
$ mount -t ubifs ubi0:rootfs /mnt/ubifs
```

mounts "rootfs" volume of UBI device 0 to /mnt/ubifs ("rootfs" is volume name). This method of specifying UBI volume is more preferable because it does not depend on volume number.

Note, if *x* is not specified, UBIFS assumes 0, i.e., "ubi0:rootfs" and "ubi:rootfs" are equivalent.

Some environments like busybox are confused by the ":" delimiter (e.g., ubi:rootfs) and "!" may be used instead (e.g., ubi!rootfs).

In order to mount UBIFS as the root file system, you have to compile UBIFS into the kernel (instead of compiling it as a kernel module) and specify proper kernel boot arguments and make the kernel mount UBIFS on boot. You have to provide the boot arguments to attach the UBI device (using the `ubi.mtd=` argument, see [here](#)). Then you should tell the kernel the file system type by providing the `rootfstype=` argument. And finally, you should specify which UBI volume has to be mounted on boot using the `root=` argument. The volume is specified the same way as described above (`ubiX_Y` or `ubiX:NAME`).

The following is an example of the kernel boot arguments to attach `mtd0` to UBI and mount UBI volume "rootfs":

```
ubi.mtd=0 root=ubi0:rootfs rootfstype=ubifs
```

Please, see [this](#) section for information about how to create UBI devices and [this](#) section for information about how to create UBI volumes.

How do I create an UBIFS image?

Creating UBIFS images might be a little trickier than creating JFFS2 images. First of all, you have to understand that UBIFS works on top of UBI which works on top of MTD which basically represents your raw flash. This means, that if you need to create an image which should be flashed to the raw flash, you should first create an UBIFS image, then UBI image. In other words, the process has 2 steps.

However, as described [here](#), UBI has a volume update facility and there is an [ubiupdatevol](#) utility for this. So you may update UBI volumes on your running system as well. In this case you only need an UBIFS image, and you do not have to make the UBI image, i.e., the process has only 1 step in this case.

Moreover, you may even build the image on your target system and write it directly to your UBI volume - just specify the volume character device as the output file to `mkfs.ubifs`.

So, there are 2 utilities:

- [mkfs.ubifs](#) which creates UBIFS images;
- [ubinize](#) which creates UBI images out of UBIFS images.

And depending on the needs you use either `mkfs.ubifs` or `mkfs.ubifs` plus `ubinize`. Choose the former if you are going to upload the update UBIFS image on your target and then update the UBI volume using `ubiupdatevol`. Choose the latter if you are going to flash the image to raw flash, e.g., at the factory.

The UBI and UBIFS images depend on parameters of the flash they are going to be used on. Namely, you have to know the following characteristics of the flash before creating images:

- MTD partition size;
- flash physical eraseblock size;
- [minimum flash input/output unit](#) size;
- for NAND flashes - [sub-page](#) size;
- logical eraseblock size.

If you run Linux kernel version 2.6.30 or higher, or you have the MTD sysfs support back-ported, then you may find all these parameters by running the [mtdinfo](#) tool with `-u` parameter. Of course, the tool has to be run on the target system.

Please, refer [this](#) for more information about how to find these parameters.

And optionally, you should decide which compression algorithm you want to use for the file-system. UBIFS supports zlib and LZO (default) at the moment (see [here](#)). There is also [favor LZO](#) `mkfs.ubifs` compression method. Generally, zlib compresses better, but it is slower on both compression and decompression. So this is a trade-off between space savings and speed. The best idea is to try both and choose the one which is more appropriate for you. [Here](#) you may find compression test results for ARM platform. Alternatively, the compression may be switched off. See `-x` option of the `mkfs.ubifs` utility.

There are other advanced file-system and UBI characteristics which may be altered with various options of the tools. Use them only if you understand what they do.

The below example demonstrates how to create an UBI/UBIFS image for a 256MiB SLC OneNAND flash chip with 128KiB physical eraseblocks, 2048-byte NAND pages, and 512-byte sub-pages (this means that it allows to do 4x512 bytes writes to the same NAND page, which is quite typical for SLC flashes). The resulting image will have only one UBI volume storing UBIFS file-system.

```
$ mkfs.ubifs -q -r root-fs -m 2048 -e 129024 -c 2047 -o ubifs.img
$ ubinize -o ubi.img -m 2048 -p 128KiB -s 512 ubinize.cfg
```

where ubinize.cfg contains:

```
$ cat ubinize.cfg
[ubifs]
mode=ubi
image=ubifs.img
vol_id=0
vol_size=200MiB
vol_type=dynamic
vol_name=rootfs
vol_flags=autoresize
```

Some comments about what the options mean:

- `-r root-fs`: tells `mkfs.ubifs` to create an UBIFS image which would have identical contents as the local `root-fs` directory;
- `-m 2048`: tells `mkfs.ubifs` that the minimum input/output unit size of the flash this UBIFS image is created for is 2048 bytes (NAND page in this case);
- `-e 129024`: logical eraseblock size of the UBI volume this image is created for;
- `-c 2047`: specifies maximum file-system size in logical eraseblocks; this means that it will be possible to use the resulting file-system on volumes up to this size (less or equivalent); so in this particular case, the resulting FS may be put on volumes up to about 251MiB (129024 multiplied by 2047); See [this](#) section for more details.
- `-p 128KiB`: tells `ubinize` that physical eraseblock size of the flash chip the UBI image is created for is 128KiB (128 * 1024 bytes);
- `-s 512`: tells `ubinize` that the flash supports sub-pages and sub-page size is 512 bytes; `ubinize` will take this into account and put the VID header to the same NAND page as the EC header.

The `ubinize` utility requires volumes description file. Please, refer [this](#) section for more `ubinize` usage information.

In the example, the `ubinize.cfg` file tells `ubinize` to create an UBI image which has a single 200MiB dynamic volume with ID 0, and name "rootfs". The configuration file also sets the "autoresize" volume flag, which means that the volume will be automatically enlarged by UBI to have the maximum possible size when it runs for the first time. See [here](#) for more information about what the auto-resize feature is. And because we specified "`-c 2047`" `mkfs.ubifs` option, UBIFS will also automatically re-size on the first mount. So the end result will be that you have one single volume of maximum possible size, and UBIFS spans whole volume.

Please, run `ubinize -h` and `mkfs.ubifs -h` for more information and for more possibilities to tweak the generated images.

Here is one more example for a 32MiB NOR flash with 128KiB physical eraseblock size.

```
$ mkfs.ubifs -q -r root-fs -m 1 -e 130944 -c 255 -o ubifs.img
$ ubinize -o ubi.img -m 1 -p 128KiB ubinize.cfg
```

where ubinize.cfg contains:

```
$ cat ubinize.cfg
[ubifs]
mode=ubi
image=ubifs.img
vol_id=0
vol_size=30MiB
vol_type=dynamic
vol_name=rootfs
vol_alignment=1
vol_flags=autoresize
```

And one more example for a 512MiB MLC NAND flash with 128KiB physical eraseblock size, 2048 bytes NAND page size and no sub-page write support.

```
$ mkfs.ubifs -q -r root-fs -m 2048 -e 126976 -c 4095 -o ubifs.img
$ ubinize -o ubi.img -m 2048 -p 128KiB ubinize.cfg
```

where `ubinize.cfg` contains:

```
$ cat ubinize.cfg
[ubifs]
mode=ubi
image=ubifs.img
vol_id=0
vol_size=450MiB
vol_type=dynamic
vol_name=rootfs
vol_alignment=1
vol_flags=autoresize
```

To flash UBI images, please use the `ubiformat` utility (see [here](#)) or use/implement a **proper** custom flasher program. ([here](#) you may find some hints). Please, read [this](#) section for more information why you should use `ubiformat`.

Is it OK to mount empty UBI volumes?

Yes, it is OK to mount empty UBI volumes, i.e. the volumes which contain only `0xFF` bytes. In this case UBIFS formats the media automatically with default parameters (journal size, compression, etc). But generally, this feature should have limited use (developing, debugging), and a proper UBIFS image should preferably be created and flashed (see [this](#) section).

Note, UBI has similar property and it automatically formats the flash media if it is empty (see [here](#)). So if there is an `mtd0` MTD device, the following will work:

```
# Wipe the MTD device out. Note, we could use flash_eraseall, but we do not
# want to lose erase counters
ubiformat /dev/mtd0

# Load UBI module
modprobe ubi

# Attach mtd0 to UBI - UBI will detect that the MTD device is
# empty and automatically format it. This command will also create
# UBI device 0 and udev should create /dev/ubi0 node
ubiattach /dev/ubi_ctrl -m 0

# Create an UBI volume - the created volume will be empty
ubimkvol /dev/ubi0 -N test_volume -s 10MiB

# Mount UBIFS - it will automatically format the empty volume
mount -t ubifs ubi0:test_volume /mnt/ubifs
```

It is also possible to wipe out an existing UBIFS volume represented by `/dev/ubi0_0` using the following command:

```
ubiupdatevol /dev/ubi0_0 -t
```

What is the purpose of `-c (--max-leb-cnt)` `mkfs.ubifs` option?

It is a form of specifying file-system size. But instead of specifying the exact file-system size, this option defines the *maximum* file-system size (more strictly, maximum UBI volume size). For example, if you use `--max-leb-cnt=200 mkfs.ubifs` option, then it will be possible to put the resulting image to smaller UBI volume and mount it. But if the image is put to a larger UBI volume, the file-system will anyway take only first 200 LEBs, and the rest of the volume will be wasted.

Note, the `--max-leb-cnt` option does not affect the size of the resulting image file, which depends only on the amount of data in the file-system. `mkfs.ubifs` just writes the `--max-leb-cnt` value to the file-system superblocks.

This feature is quite handy on NAND flashes, because they have random amount of initial bad eraseblocks (marked as bad in production). This means, that different devices may have slightly different volume sizes (especially if the UBI [auto-resize](#) feature is used). So you may specify the maximum possible volume size and this will guarantee that the image will work on all devices, irrespectively on the amount of initial bad eraseblocks.

Fundamentally, `mkfs.ubifs` has to know file-system size because UBIFS maintains and stores per-LEB information (like amount of dirty and free space in each LEB) in so-called LPT area on the media. So obviously, the size of this area depends on the total amount of LEBs, i.e. on the volume size. Note, various characteristics of the LPT B-tree depend on the LPT area size, e.g., we use less bits in LPT tree keys of smaller LPT area. So do not use unnecessarily large `--max-leb-cnt` value to achieve better performance.

Can UBIFS mount loop-back devices?

Unfortunately not, because loop-back devices are block devices (backed by regular files), while UBIFS works on top of UBI devices (see [here](#)).

However, there is an unusual way to make UBIFS work with a file-backed image using NAND simulator (see [here](#)). If your image is not very big, then you can create a RAM-backed `nandsim` MTD device, then copy your image to that emulated MTD device. If the image is large and you do not have that much RAM, you can create a file-backed `nandsim` MTD device using the `cache_file` `nandsim` module option. Below is an example:

```
# Create a 1GiB emulated MTD device backed by regular file "my_image"
$ modprobe nandsim cache_file=my_image first_id_byte=0xec second_id_byte=0xd3 \
  third_id_byte=0x51 fourth_id_byte=0x95
```

See [here](#) for more instructions about using UBIFS with `nandsim`.

What is "favor LZO" compression?

Starting from version 1.1, the `mkfs.ubifs` utility supports so-called "favor LZO" compression. This is not a new compression algorithm, it is just a method of combining LZO and zlib compressors to balance speed and compression ratio. A similar method exists in the `mkfs.jffs2` utility, and we borrowed this idea from there.

As [this](#) documentation section highlights, zlib compressor provides better compression ratio comparing to the LZO compressor, but it is considerably slower, especially on embedded platforms which do not usually have powerful CPUs. The favor LZO compression method makes it possible to use zlib compressor for data chunks which zlib may compress much better than LZO, and to use LZO compressor for data chunks which zlib compresses only slightly better than LZO. See [this](#) section for some favor LZO experiment results on ARM platform.

Here is the favor LZO algorithm. Each 4KiB data chunk is compressed by both zlib and LZO compressors. If zlib compresses at least 20% better than LZO, then zlib is used for this data chunk. Otherwise, LZO is used. Very simple. The 20% threshold is configurable via the `-X mkfs.ubifs` option, so you may make it 10% or anything else. Example:

```
$ mkfs.ubifs -q -r rootfs -m 2048 -e 129024 -c 2047 -x favor_lzo -X 5 -o ubifs.img
```

This command creates an UBIFS image containing the `rootfs` directory, using favor LZO compression with threshold 5%. This means that if a data chunk compresses at least 5% better with zlib, then `mkfs.ubifs` uses zlib, otherwise it uses LZO.

So UBIFS images created with favor LZO compression will contain both LZO and zlib-compressed data nodes. However, the default file-system compressor will be LZO. This means, the kernel will use LZO compressor for all data. This is because the favor LZO method is not implemented in-kernel, just because UBIFS authors did not need it there. Of course you can mount the UBIFS images created with the favor LZO compression method, and they will work fine. But if you write a file to the UBIFS file-system, it will be compressed only by the LZO compressor. So favor LZO is actually useful only for read-only files which are not going to be over-written. However, it should be easy to implement favor LZO in the kernel.

Why do I have to use ubiformat?

The first obvious reason is that `ubiformat` preserves erase counters, so you do not lose your wear-leveling information when flashing new images.

The other reason is more subtle, and specific to NAND flashes which have ECC calculation algorithm which produces ECC code not equivalent to all `0xFF` bytes if the NAND page contains only `0xFF` bytes. Consider an example.

- We erase whole flash, so everything is `0xFF`'ed now.
- We write an UBI/UBIFS image to flash using `nandwrite`.
- Some eraseblocks in the UBIFS image may contain several empty NAND pages at the end, and UBIFS will write to them when it is run.
- The `nandwrite` utility writes whole image, and it explicitly writes `0xFF` bytes to those NAND pages.
- The ECC checksums are calculated for these `0xFF`'ed NAND pages and are stored in the OOB area. The ECC codes are not `0xFF`'ed. This is often the case for HW ECC calculation engines, and it is difficult to fix this. Normally, ECC codes should be `0xFF`'ed for such pages.
- When later UBIFS runs, it writes data to these NAND pages, which means that a new ECC code is calculated, and written on top of the existing one (unsuccessfully, of course). This may trigger an error straight away, but usually at this point no error is triggered.
- At some point UBIFS is trying to read from these pages, and gets an ECC error (`-EBADMSG = -74`).

In fewer words, `ubiformat` makes sure that every NAND page is written once and only once after the erasure. If you use `nandwrite`, some pages are written twice - once by `nandwrite`, and once by UBIFS.

If you can not use `ubiformat`, an alternative is to set the "free space fixup" flag when generating the UBIFS image (see [here](#)).

What is the the purpose of the -F (--space-fixup) mkfs.ubifs option?

Because of subtle ECC errors that can arise when programming NAND flash (see [here](#)), `ubiformat` is the recommended way of flashing a UBI image which contains a UBIFS file system. However, this is not always possible - for example, some embedded devices are manufactured using an industrial NAND flash programmer which has no knowledge of UBI or UBIFS.

The `-F` option causes `mkfs.ubifs` to set a special flag in the superblock, which triggers a "free space fixup" procedure in the kernel the very first time the filesystem is mounted. This fixup procedure involves finding all empty pages in the UBIFS file system and re-erasing them. This ensures that NAND pages which contain all `0xFF` data get fully erased, which removes any problematic non-`0xFF` data from their OOB areas.

Of course it is not possible to re-erase individual NAND pages, and entire PEBs are erased. UBIFS performs this procedure by reading the useful (non `0xFF`'ed) contents of LEBs and then invoking the [atomic LEB change](#) UBI operation. Obviously, this means that UBIFS has to read and write a lot of LEBs which takes time. But this happens only once, and the "free space fixup" procedure then unsets the "fixup" UBIFS superblock flag.

This option is supported if you are running a kernel version 3.0 or higher, or if you have pulled the changes from a UBIFS [back-port tree](#). Note that `ubiformat` is still the preferred flashing method if the image is not being flashed for the first time, since it preserves existing erase counters (while using `nandwrite` or its equivalent does not).

How do I compile mkfs.ubifs?

The `mkfs.ubifs` utility requires `zlib`, `lzo` and `uuid` libraries. The former two are used for compressing the data, and the latter one is used for generating *universally unique ID number* for the file-system. In Fedora install `zlib-devel`, `lzo-devel`, and `libuuid-devel`. Old Fedora distributions (Fedora 11 and earlier) had the `uuid` library in the `e2fsprogs-devel` package. In Debian install `zlib1g-dev`, `liblzo2-dev` and `uuid-dev` packages.

Note, [this](#) section provides information about other dependencies in the `mt-utils` tree.

How do I change a file atomically?

Changing a file atomically means changing its contents in a way that unclean reboots could not lead to any corruption or inconsistency in the file. The only reliable way to do this in UBIFS (and in most of other file-systems, e.g. JFFS2 or ext3) is the following:

- make a copy of the file;
- change the copy;
- synchronize the copy (see [here](#));
- re-name the copy to the file (using the `rename()` *libc* function or the `mv` utility).

Note, if a power-cut happens during the re-naming, the original file will be intact because the re-name operation is atomic. This is a `POSIX` requirement and UBIFS satisfies it.

Often applications do not do the third step - synchronizing the copy. Although this is generally an application bug, the ext4 file-system has a hack which makes sure the data of the copy hits the disk before the re-name meta-data, which "fixes" buggy applications. However, UBIFS does not have this feature, although we plan to implement it. Please, refer [this](#) section.

Does UBIFS support atime?

No, it does not support atime. The authors think it is not very useful in embedded world and did not implement this. Indeed most of the users do not provably want the file-system doing inode updates every time they are read.

Does UBIFS support NFS?

Not, it does not, which means you cannot export UBIFS file-system via NFS. We did make an attempt to support NFS, but the support was not exactly correct so it was dropped, and we have never found time to come back to that. Please, refer to [this thread](#) for some more details. The original patch can also be found there.

Does UBIFS become slower when it is full?

Yes, UBIFS writes (but not reads) become slower when it is full or close to be full. There are 2 main reasons for this:

- Garbage Collection becomes slower, because the small pieces of dirty space are distributed all over the media, and UBIFS Garbage Collector has to move many eraseblocks to produce one free eraseblock; this is a fundamental reason and it exists in JFFS2 as well;
- The file-system becomes more synchronous; UBIFS buffers dirty data, but due to compression and some other factors like wasting small pieces of space at the end of eraseblocks, UBIFS does not exactly know how much space the buffered dirty data would take on the flash media, so it uses pessimistic calculations and assumes that the data are uncompressible. In many cases this is not true, but UBIFS has to assume worst-case scenario. So when all free space on the file-system is reserved for the buffered dirty data, but users want to write more, UBIFS forces write-back to actually write the buffered dirty data and see how much space will be available after that. Then UBIFS may allow new data to be written. Thus, it is obvious that the less free flash space is available, the less dirty data may be buffered, and the more synchronous the file-system becomes.
- Note, also slows down when it is close to being full, but UBIFS should be better than JFFS2 in this respect (slow down less). This is because UBIFS always chooses optimal LEBs to garbage-collect, while JFFS2 may choose random eraseblocks.

Why df reports too few free space?

UBIFS flash space accounting is quite challenging and it is not always possible to report accurate amount of free space. The `df` utility usually reports *less* free space than users may actually write to the file-system, but it *never* reports more space.

UBIFS cannot precisely predict how much data the user will be able to write to the file-system. There are several reasons for this - compression, write-back, space wastage at the end of logical eraseblocks, garbage-collection, etc. Please, refer [this](#) section for details.

Note, JFFS2 also has problems with free space predictions, but in average, it reports much more accurate amount of free space. However, JFFS2 may lie and report more free space than it actually has. For example, we experienced situations when JFFS2 reported 8MiB free space, while we were able to write only 2 MiB of data. This makes some user-space applications very unhappy.

UBIFS also lies, but it always report *less* space that user may actually write. For example, it may report 2MiB of free space, but if you start writing to it, may be able to write 4MiB there (even if you have compression disabled).

Thus, the only way to find out *precise* amount of free space is to fill up the file-system and see how much has been written. Try something like this:

```
$ touch /mnt/ubifs/file
$ chatter -c /mnt/ubifs/file # Disable compression for this file
$ dd if=/dev/zero of=/mnt/ubifs/file bs=4096
```

or

```
# Presumably random data does not compress
dd if=/dev/urandom of=/mnt/ubifs/file bs=4096
```

and see the size of the file. And do not forget that some space is reserved for the super-user (see [here](#)), so it is better to be the root.

UBIFS users should know that the more dirty cached FS data there are, the less precise is the `df` report. Try to create a big file, and look at the `df` report. Then synchronize the file-system (using the `sync` command) and look at the `df` report again. You should notice that `df` reports more free space after the synchronization. Here is an example:

```
# Create a 64MiB uncompressible file
$ dd if=/dev/urandom of=/mnt/ubifs/file bs=8192 count=8192
$ df
Filesystem            1K-blocks      Used Available Use% Mounted on
ubi0:ubifs             117676      68880     43736   62% /mnt/ubifs

$ sync
$ df
Filesystem            1K-blocks      Used Available Use% Mounted on
ubi0:ubifs             117676      64304     48308   58% /mnt/ubifs
```

Notice that the amount of free space increased by 4%. However, that was an uncompressible file. Here is a similar example, but which uses a file which compresses well:

```
# Create a 64MiB file containing zeroes
$ dd if=/dev/zero of=/mnt/ubifs/file bs=8192 count=8192
$ df
```

```
Filesystem            1K-blocks      Used Available Use% Mounted on
ubi0:ubifs            117676        69312      43304   62% /mnt/ubifs
```

```
$ sync
$ df
```

```
Filesystem            1K-blocks      Used Available Use% Mounted on
ubi0:ubifs            117676        7052      105564    7% /mnt/ubifs
```

If instead of synchronizing the file-system you just watch how the `df` report is changing, you will notice that the amount of free space continuously grows until reaches its final value. This happens because the kernel starts writing the dirty data back by time-out (5 sec by default) and the amount of dirty data goes down, making the `df` report more precise.

If you want to have as precise free space prediction as possible - synchronize the file-system. This not only flushes dirty data to the media, this also commits UBIFS journal, which improves free space prediction even more. The other possibility is to mount UBIFS in synchronous mode using `-o sync` mount option. However, it may perform not as well as in asynchronous (default) mode.

It is also worth noting that the closer is UBIFS to being full, the less accurate is free space reporting.

To conclude:

- do not trust `df` free space reports;
- `df` reports *less* free space in most cases;
- if `df` reports `x` bytes of free space, you are guaranteed to be able to write at least `x` bytes of data to a file, but usually more than `x` bytes;
- run `sync` to get the most accurate `df` report; alternatively, you may mount UBIFS in synchronous mode using `-o sync` mount option;
- read [this](#) section if you are curious why UBIFS has issues with `df`.

Why does my UBIFS volume have significantly lower capacity than my equivalent JFFS2 volume?

When migrating a filesystem image from JFFS2 to UBIFS you may notice some or all of the following:

- The size of the image file to be flashed is significantly larger than before.
- `df` reports significantly less available free space than before (while there may be *some* truth in this, please be aware of the other considerations that must be applied to `df` values described elsewhere in this FAQ).
- When measuring the capacity of the disk by seeing how much data can be written, JFFS2 is able to store significantly more data than UBIFS.

There are several reasons for this.

Firstly, did you create your volumes with default arguments to `mkfs.jffs2` and `mkfs.ubifs`? This alone introduces a significant difference: `mkfs.jffs2` effectively defaults to `zlib` compression and disables `lzo`, but `mkfs.ubifs` defaults to `LZO` compression. `zlib` compresses significantly better, but is noticeably slower to compress/decompress at runtime. Using `zlib` compression with `mkfs.ubifs` will probably reduce the size of your NAND image file by 10-15%, put it on-par with an equivalent JFFS2 image, and increase the available capacity, at the price of performance.

Secondly, did you choose a good value for `vol_size` or enable `autoresize` in your `ubinize` configuration? It is possible that your UBI volume is not utilising the full space of the available flash.

Beyond that, it is unfortunately true that UBI/UBIFS has higher space overhead than its predecessors. OLPC has [measured](#) this overhead to be approximately 50MiB per 1GiB of storage (using JFFS2 as a baseline). While UBIFS cannot be made as good as JFFS2 in this respect, work could be undertaken to improve space efficiency for current or future UBI/UBIFS versions. Reasons for overhead and opportunities for improvement include:

- For MTD controllers (such as CaFe) that do not support sub-page writes, 2 KiB is taken from each eraseblock. If such controller drivers were adapted (or hacked) to use software ECC for the first NAND page of every eraseblock, around 8 MiB could be recovered per 1GB of storage.
- UBIFS on-flash data structures could be reduced in size. These headers currently include many unused fields that are reserved for future use.
- UBIFS could be improved to reserve less space for internal needs, but touching this part of the code is risky.

How do I disable compression?

UBIFS compression may be disabled for whole file system during the image creation time using the `"-x none"` `mkfs.ubifs` option. However, if UBIFS compression is enabled, it may be disabled for individual files by cleaning the inode *compression flag*:

```
$ chattr -c /mnt/ubifs/file
```


in shell, or

```
/* Get inode flags */
ioctl(fd, FS_IOC_GETFLAGS, &flags);
/* Set "compression" flag */
flags &= ~FS_COMPR_FL;
/* Change inode flags */
ioctl(fd, FS_IOC_SETFLAGS, &flags);
```

in C programs. Similarly, if compression is disabled by default, you may enable it for individual inodes by setting the compression flag. Note, the code which uses the compression flag works fine on other Linux file-systems, because the flag is just ignored in this case.

It might be a good idea to disable compression for say, mp3 or jpeg files which would anyway not compress and UBIFS would just waste CPU time trying to compress them. The compression may also be disabled if one wants faster file I/O, because UBIFS would not need to compress or decompress the data on reads and write. However, I/O speed may actually become slower if compression is disabled. Indeed, in case of a very fast CPU and very slow flash compressed writes are faster, but this is usually not true for embedded systems.

How do I use UBIFS with nandsim?

The same way as with any MTD device. Here is an example of how to load nandsim, create an UBI volume and mount it.

```
# Create an 256MiB emulated NAND flash with 2KiB NAND page size
# (you should see the new MTD device in /proc/mtd)
modprobe nandsim first_id_byte=0x20 second_id_byte=0xaa \
    third_id_byte=0x00 fourth_id_byte=0x15

# MTD is not LDM-enabled and udev does not create device
# MTD device nodes automatically, so create /dev/mtd0
# (we assume that you do not have other MTD devices)
mknod /dev/mtd0 c 90 0

# Load UBI module and attach mtd0
modprobe ubi mtd=0

# Create a 200MiB UBI volume with name "ubifs-vol"
ubimkvol /dev/ubi0 -N ubifs-vol -s 200MiB

# Mount UBIFS
mount -t ubifs /dev/ubi0_0 /mnt/ubifs
```

For more information about nandsim see [here](#).

How do I extract files from an UBI/UBIFS image?

Unfortunately, at the moment there are no user-space tools which can unwrap UBI and UBIFS images. UBIFS cannot be loop-back mounted as well, because it does not work with block devices.

However, there is a hacky way to unwrap UBI/UBIFS images. But you have to make sure you have UBIFS support in your host machine. UBIFS is a relatively new file system and is not supported by all Linux distributions. But at least Fedora 11 does include it.

Let's consider a simple example. Suppose you have an `ubi.img` file. This is an UBI image, which contains a single UBI volume, which in turn contains UBIFS file-system. In other words, this is an image which was created using the `mkfs.ubifs` and `ubinize` tools, just like it is described in [this](#) section (the image is created for a 256MiB NAND flash with 2KiB NAND page size and which supports sub-pages). Here is what you can do:

```
# Create an 256MiB emulated NAND flash with 2KiB NAND page size
# (you should see the new MTD device in /proc/mtd)
modprobe nandsim first_id_byte=0x20 second_id_byte=0xaa \
    third_id_byte=0x00 fourth_id_byte=0x15

# MTD is not LDM-enabled and udev does not create device
# MTD device nodes automatically, so create /dev/mtd0
# (we assume that you do not have other MTD devices)
mknod /dev/mtd0 c 90 0

# Copy the contents of your image to the emulated MTD device
dd if=ubi.img of=/dev/mtd0 bs=2048

# Load UBI module and attach mtd0
modprobe ubi mtd=0
```

```
# Mount UBIFS
mount -t ubifs /dev/ubi0_0 /mnt/ubifs
```

Now you have the file-system in /mnt/ubifs. Use the following to get rid of it:

```
umount /mnt/ubifs
rmmmod ubifs ubi nandsim
```

I need more space - should I make UBIFS journal smaller?

UBIFS journal is very different to ext3 journal. In case of ext3, the journal has fixed position on the block device. The data are first written to the journal, and then copied to the file-system. This copying is done during the commit. After the commit, new data may be written to the journal, and so on. So in case of ext3 changing journal size would change file-system capacity.

The situation is different in UBIFS. UBIFS journal is "wandering". It does not have fixed position in the UBI volume. When the journal is full, UBIFS is committing it which means it simply amends the FS index to refer the data which is stored in the journal. Then different LEBs are picked for the new journal, and so on. So the journal constantly migrates. The journal contains the same information as other data LEBs, but this information is just not referred from the index, it is not indexed. But there is flash space reserved for the indexing information, and this space is used during commit.

In other words, if your file-system is full, the journal will also be full and contain data. And the situation does not really change if you vary journal size.

To put it simple, the amount of available space on UBIFS does not really depend on the journal size. There is very weak dependency, though, because for a bigger journal we need a bigger log, but it really does not make a noticeable difference.

Why is my file empty after an unclean reboot?

Zero-length files are a special case of corruption which happens when an application first truncates a file, then updates it. The truncation is synchronous in UBIFS, so it is written to the media straight away. But when the data are written, they go to the page cache, not to the flash media. So when an unclean reboot happens, the file becomes empty (truncated) because the data are lost.

Zero-length files also appear when an application creates a new file, then writes to the file, and a power cut happens. The reason is similar - file creation is a synchronous operation, data writing is not.

Well, the description is a bit simplified. Actually, when a file is created or truncated, the creation/truncation UBIFS information is written to the [write-buffer](#), not straight to the media. So if a power cut happens before the write-buffer is synchronized, the file will disappear (creation case) or stay intact (truncation case). But since the write-buffer is small and all UBIFS writes go there, it is usually synchronized very soon. After this point the file is created/truncated for real.

There are several ways to affect the situation.

1. Synchronize files using something like `fsync()` - [this](#) section contains all information about synchronization. But this does not guarantee that unclean reboots will not corrupt the file. Indeed, if an unclean reboot happens when only half of the data are written, the file will be corrupted/inconsistent. But this lessens the probability of corruptions.
2. If you gave up fixing all your applications, you may mount UBIFS in synchronous mode - use `"-o sync"` mount option. But this makes UBIFS perform worse. And unfortunately, this also does not save you from all possible corruptions - you may still end up with holes (zeroes) at the end of files. See [this](#) section for more information.
3. You may use the well-known atomic file update technique - see [this](#) section.
4. You may use the Linux write-back knobs to lessen the dirty write-back time-out - see [this](#) section.

The ext4 file-system helps buggy applications to lessen the probability of getting zero-length files by implementing a special hack. Please, refer [this](#) section for more information. UBIFS does not provide a similar hack, although we are planning to implement it.

Why does my file have zeroes at the end after an unclean reboot?

Power cuts often lead to holes at the end of files. Holes are areas of the file which contain no data. For example, if you truncate a file to a larger size and synchronize it - you end up with a hole. Holes are read as zeroes. Often files with holes are referred to as sparse files. People sometimes deliberately create sparse files in order to save space - this is sometimes better than filling files with lots of zeroes.

Please, read more information about how unclean reboots result in holes in [this](#) section.

What does the "ubifs_bgt0_0" thread do?

The UBIFS background thread is created for every mounted file-system and has `"ubifs_bgtX_Y"` name, where `"X"` is UBI device number and `"Y"` is UBI volume ID. For example, `"ubifs_bgt1_2"` is UBIFS background thread corresponding to the mounted volume 2 on UBI device 1.

The background thread exists for optimization. One of its functions is background journal commit. It starts committing the journal in background when it is about 80% full. The idea is to make sure the journal is committed or almost committed by the time it becomes full, so writers would not have to wait for commit and keep writing data. The UBIFS presentation slides and the UBIFS white-paper contain more information about the journal and committing, see [this](#) section.

The other function of the background thread is flushing write-buffers. Each write-buffer has a timer, and when the timer expires, the background thread is woken up and the write-buffer is flushed. Please, refer [this](#) section for more information about UBIFS write-buffers.

Another thing which the background thread could do is background garbage collection, just like in JFFS2. However, this is not implemented and UBIFS does not garbage-collect in background at the moment.

UBIFS suddenly became read-only - what is this?

Read-write UBIFS file-system may suddenly become read-only because of an error. This is how UBIFS reacts on unexpected errors which it cannot properly handle - it immediately switches to read-only mode in order to protect the data from any possible further corruption.

If this happened, you should look at UBIFS-related `dmesg` messages. UBIFS usually prints error messages before switching to read-only mode. The messages may shed some light on what happened. Feel free to ask for help from the [MTD mailing list](#). If you think this is an UBIFS bug, please, send a [bug report](#).

How do I detect if UBIFS became read-only?

If you use up-to-date UBIFS which includes commit `2fde99cb55fb9d9b88180512a5e8a5d939d27fec` (UBIFS: mark VFS SB RO too), then you should be able to find this out from `/proc/mounts`. You should also be able to use something like `inotify` to catch events when UBIFS becomes R/O (e.g., due to some errors).

I see this UBIFS error: "validate_sb: LEB size mismatch: 129024 in superblock, 126976 real"

When you create an UBIFS image using the `mkfs.ubifs` utility, you specify LEB size using the `-e` option. This is a very important parameter and you should specify it correctly in order to have working UBIFS image. Indeed, LEB size is the major UBIFS storage unit, e.g., UBIFS nodes never cross LEB boundaries, garbage collection is performed on individual LEBs, etc. See [this](#) section for more information.

The error message means that LEB size information which is stored in the UBIFS superblock does not match the real LEB size, which UBIFS takes from UBI. The superblock was created by the `mkfs.ubifs` utility, therefore you failed to pass the correct LEB size to the utility. Fix this by passing correct LEB size via the `-e` option.

I see this error: "INFO: task pdflush:110 blocked for more than 120 seconds"

If this happens with a NOR flash, then this is a known issue and is about the UBI background thread doing a lot of erasures. When you attach to empty flash to UBI, it will format the flash in background, in the context of the UBI background thread (see [here](#)). Formatting means that for each eraseblock it does the following:

- erases (very slow on NOR!)
- writes the UBI headers

Depending on your MTD/CFI chip driver the "MTD/CFI chip lock" may be held for the time it needs to erase an eraseblock. User-space applications which manipulate files on the UBIFS file-system may then also be blocked on the same "MTD/CFI chip lock". This causes `pdflush` to block as well since it tries to acquire the UBIFS journal mutex, which is already locked by the process which is waiting on the "MTD/CFI chip lock". See [this](#) discussion for some more details. The ways to solve this:

1. Use `ubiiformat` and format the NOR partition before attaching it to UBI. But this will not help in situations when you delete may files, and starts erasing many eraseblocks, so the "MTD/CFI chip lock" becomes very contended.
2. Use `erase-suspend` for writing (if your chip supports this).

I get: "init_constants_early: too few LEBs (12), min. is 17"

This error means that you are trying to mount too small UBI volume. Probably because your flash is too small? Try to use JFFS2, then, because it suits small flashes better since it has much lower space overhead. Indeed, UBIFS stores much more indexing information on the flash media than JFFS2, so it has much higher overhead. Also, UBI has some overhead (see [here](#)). Thus, if you have a small flash device (e.g., about 64MiB), it makes sense to consider using JFFS2.

I want to study UBIFS - any recommendations?

Follow [these](#) instructions.

How do I debug UBIFS?

Use fake MTD device

It is often much easier to debug on a PC with a flash emulator, rather than debugging on a real system. E.g., most of the UBIFS development was done on a standard PC with the `nandsim` NAND simulator. Please, refer [here](#) for more information about the available fake MTD devices.

Enable UBIFS debugging support

Enable UBIFS debugging support in the configuration menu (the "**UBIFS debugging support**" checkbox). This will make sure that assertions, debugging messages, self-checks and test modes are compiled-in. The assertions will be enabled, but messages, self-checks and test modes will be disabled by default. This option will also make many error messages to be more verbose (e.g., include flash dumps). This option should not slow down UBIFS, so it is recommended to always have it switched on, unless you are very concerned about UBIFS code size.

Debugging messages

Sometimes it is necessary to make UBIFS print about what it is doing. You may enable various UBIFS debugging messages using the `debug_msgs` UBIFS module parameter, or using the `ubifs.debug_msgs` kernel boot parameter if you have UBIFS compiled in. Alternatively, the same can be achieved by changing the `/sys/module/ubifs/parameters/debug_tsts` file.

The `debug_msgs` option is a bit-mask which controls which message type has to be printed. You can combine the message types arbitrarily. The following message types are supported:

Message Type	Value
General messages	1
Journal messages	2
Mount messages	4
Commit messages	8
LEB search messages	16
Budgeting messages	32
Garbage collection messages	64
Tree Node Cache (TNC) messages	128
LEB properties (lprops) messages	256
Input/output messages	512
Log messages	1024
Scan messages	2048
Recovery messages	4096

E.g., `echo 4097 > /sys/module/ubifs/parameters/debug_tsts` enables general and recovery messages.

UBIFS may print huge amount of debugging messages and slow down your system considerably. You might also end up losing them if your ring buffer is not large enough. [This](#) section explains how to make the ring buffer larger.

Extra self-checks

UBIFS contains various internal self-check functions which are often very useful for debugging or testing. However, the self-checks are very expensive and slow down UBIFS a lot. We recommend to use them only while hunting bugs or testing UBIFS changes.

Similarly to debugging messages, the self-checks can be switched on using the `debug_chks` UBIFS module parameter or the `/sys/module/ubifs/parameters/debug_chks` file. The `debug_chks` option is a bit-mask which selects the check types to be enabled.

Check type	Value
General checks	1

Check Tree Node Cache (TNC)	2
Check indexing tree size	4
Check orphan area	8
Check old indexing tree	16
Check LEB properties (lprops)	32
Check leaf nodes and inodes	64

E.g., "echo 3 > /sys/module/ubifs/parameters/debug_chks" enables general checks and TNC checks. But for testing, it is better to just enable all checks: "echo 127 > /sys/module/ubifs/parameters/debug_chks".

Test modes

UBIFS currently supports 2 test modes:

- "Force in-the-gaps" forces the "in-the-gaps" garbage collection method to be used as much as possible. Normally, UBIFS uses this method extremely rarely, only as the last resort way to do garbage collection, which makes it difficult to test. This test mode changes this behavior.
- "Failure mode for recovery" test mode makes UBIFS emulate power cuts. When a power-cut is emulated, UBIFS switches to read-only mode, and then it is supposed to be unmounted and mounted again, which causes recovery. The main idea of this mode is to emulate power cuts in "interesting" places, e.g., when changing the log, or the orphans area. Indeed, real power-cuts testing mostly interrupts UBIFS when it is writing data to the journal. The testing mode makes it more probable to interrupt UBIFS in other places. And of course, the testing mode should be considered as supplementary to the real power cut testing.

How do I send an UBIFS bug report?

Before sending a bug report, please, try to do the following:

1. run the MTD tests to validate your flash (see [here](#));
2. enable the UBIFS extra self checks and try to reproduce the problem. See [this](#) chapter for more information about how to enable self-checks.

When sending a bug report, please:

1. make sure you use up-to-date UBIFS; pull the latest UBIFS patches if needed, see [here](#); no one is interested in digging already fixed problems;
2. make sure you have compiled the kernel symbols in (CONFIG_KALLSYMS_ALL=y in .config);
3. mark the **Enable debugging support** check-box in the UBIFS kernel configuration menu (CONFIG_UBIFS_FS_DEBUG=y in .config); this option will make UBIFS print more informative error messages; **note**, you should enable UBIFS debugging, not UBI debugging - distinguish between UBI and UBIFS please, they are different things;
4. include all the messages UBIFS prints, not only those you see at the console, but also those you see when running dmesg; or before running your UBIFS test, which reproduces the error, just invoke dmesg -n8 command to make all kernel messages to go to the console; another possibility is to boot the kernel with ignore_loglevel option, in which makes the kernel print all messages to the console unconditionally; [this](#) section contains some more information about how the messages can be collected.
5. explicitly tell about whether you did any checking as described in the previous list of "actions before sending a bug-report" (running MTD tests, enabling UBIFS extra self-checks; and tell the results; if you saw any errors/warnings, describe them and include all corresponding prints from tests / UBIFS / etc;
6. provide UBI and UBIFS configuration from your .config file, or just attach whole file;
7. describe your flash device, attach the "mtdinfo -a" output (or the less useful "cat /proc/mtd" output);
8. specify which kernel version you are using; if your kernel is not the latest one, please, explicitly tell whether you updated UBIFS by pulling one of the back-port trees or not; if you did not, you will probably be asked to do this, depending on the problem;
9. describe how the problem can be reproduced;

The bugreport should be sent to the [MTD mailing list](#). Please, **do not** send private e-mails to UBIFS authors, always CC the mailing list!

Note, sometimes UBIFS bugs may appear to be UBI bugs. If you suspect there are UBI problems, please, also enable UBI debugging. Please, refer to the [UBI debugging](#) section for more information.