

XFS

Filesystem

Structure

2nd Edition
Revision 1

[This document is incomplete and currently undergoing revision]

© Copyright 2006 Silicon Graphics Inc. All rights reserved.

Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-Share Alike, Version 3.0 or any later version published by the Creative Commons Corp. A copy of the license is available at <http://creativecommons.org/licenses/by-sa/3.0/us/>.

Table of Contents

Introduction	4
Common XFS Types	5
Allocation Groups	6
Superblocks	7
AG Free Space Management	13
AG Free Space Block	13
AG Free Space B+trees	14
AG Free List	16
AG Inode Management	18
Inode Numbers	18
Inode Information	18
Inode B+trees	19
Real-time Devices	22
On-disk Inode	23
Inode Core	24
Unlinked Pointer	28
Data Fork	29
Regular Files (S_IFREG)	29
Directories (S_IFDIR)	29
Symbolic Links (S_IFLNK)	29
Other File Types	29
Attribute Fork	30
Extended Attribute Versions	30
Data Extents	31
Extent List	32
B+tree Extent List	34
Directories	38
Shortform Directories	39
Block Directories	42
Leaf Directories	47
Node Directories	53
B+tree Directories	58
Symbolic Links	61
Shortform Symbolic Links	61
Extent Symbolic Links	62
Extended Attributes	63
Shortform Attributes	64
Leaf Attributes	68
Node Attributes	72
B+tree Attributes	75
Internal Inodes	76
Quota Inodes	76
Real-time Inodes	79

Journaling Log

80

Introduction

This document describes the layout of an XFS filesystem.

It shows how to manually inspect it by showing examples using the `xfs_db` user-space tool supplied with the XFS filesystem driver.

TODO:

Common XFS Types

All the following XFS types can be found in `xfs_types.h`. NULL values are always -1 on disk (ie. all bits for the value set to one).

xfs_ino_t

Unsigned 64 bit absolute [inode number](#).

xfs_off_t

Signed 64 bit file offset.

xfs_daddr_t

Signed 64 bit disk address.

xfs_agnumber_t

Unsigned 32 bit [AG](#) number.

xfs_agblock_t

Unsigned 32 bit AG relative block number.

xfs_extlen_t

Unsigned 32 bit [extent](#) length in blocks.

xfs_extnum_t

Signed 32 bit number of extents in a file.

xfs_dablk_t

Unsigned 32 bit block number for [directories](#) and [extended attributes](#).

xfs_dahash_t

Unsigned 32 bit hash of a directory file name or extended attribute name.

xfs_dfsbno_t

Unsigned 64 bit filesystem block number combining [AG](#) number and block offset into the AG.

xfs_drfsbno_t

Unsigned 64 bit raw filesystem block number.

xfs_drtbno_t

Unsigned 64 bit extent number in the [real-time](#) sub-volume.

xfs_dfiloff_t

Unsigned 64 bit block offset into a file.

xfs_dfilblks_t

Unsigned 64 bit block count for a file.

Allocation Groups

XFS filesystems are divided into a number of equally sized chunks called Allocation Groups. Each AG can almost be thought of as an individual filesystem that maintains it's own space usage. Each AG can be up to one terabyte in size ($512 \text{ bytes} * 2^{31}$), regardless of the underlying device's sector size.

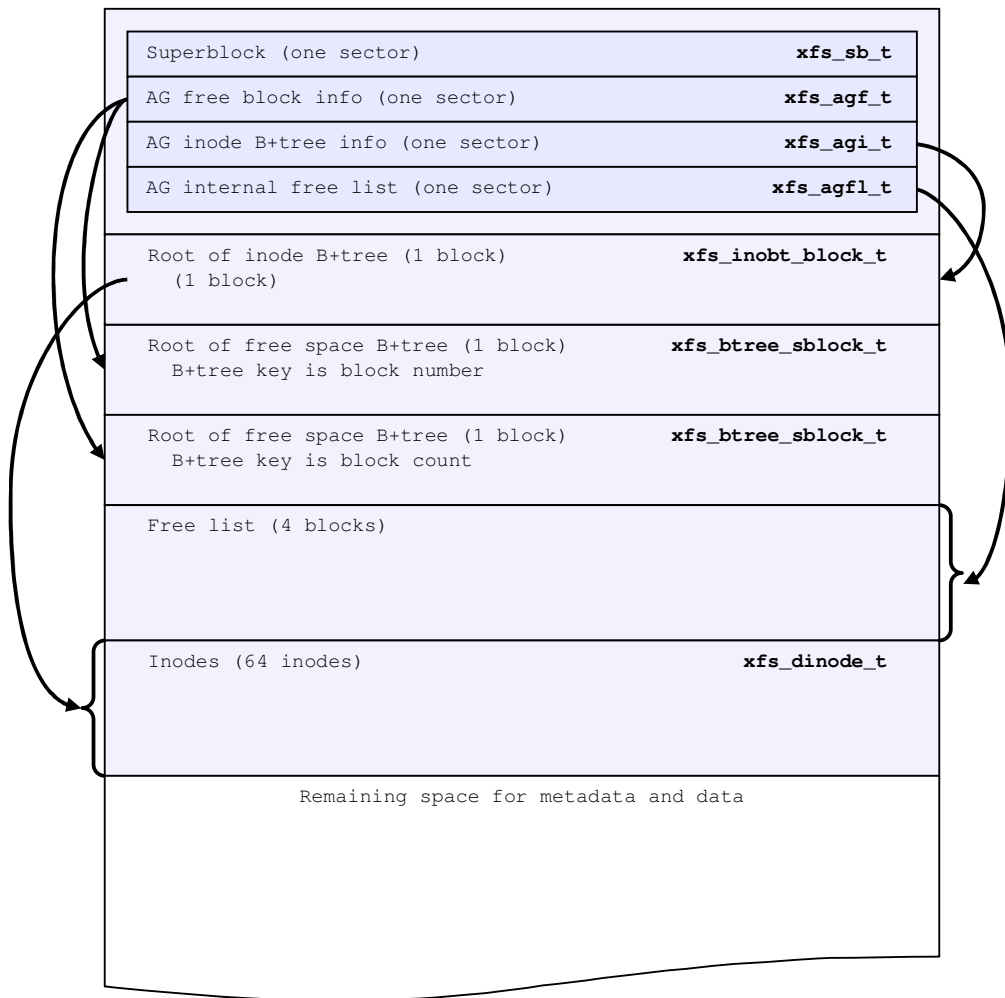
Each AG has the following characteristics:

- A super block describing overall filesystem info
- Free space management
- Inode allocation and tracking

Having multiple AGs allows XFS to handle most operations in parallel without degrading performance as the number of concurrent accessing increases.

The only global information maintained by the first AG (primary) is free space across the filesystem and total inode counts. If the `XFS_SB_VERSION2_LAZYSB_COUNTBIT` flag is set in the superblock, these are only updated on-disk when the filesystem is cleanly unmounted (umount or shutdown).

Immediately after a `mkfs.xfs`, the primary AG has the following disk layout the subsequent AGs do not have any inodes allocated:



Each of these structures are expanded upon in the following sections.

Superblocks

Each AG starts with a superblock. The first one is the primary superblock that stores aggregate AG information. Secondary superblocks are only used by `xfs_repair` when the primary superblock has been corrupted.

The superblock is defined by the following structure. The description of each field follows.

```
typedef struct xfs_sb
{
    __uint32_t        sb_magicnum;
    __uint32_t        sb_blocksize;
    xfs_drfsbno_t     sb_dblocks;
    xfs_drfsbno_t     sb_rblocks;
    xfs_drtbno_t      sb_rextents;
    uuid_t            sb_uuid;
    xfs_dfsbno_t      sb_logstart;
    xfs_ino_t         sb_rootino;
    xfs_ino_t         sb_rbmino;
    xfs_ino_t         sb_rsumino;
    xfs_agblock_t     sb_rextsize;
    xfs_agblock_t     sb_agblocks;
    xfs_agnumber_t    sb_agcount;
    xfs_extlen_t      sb_rbmblocks;
    xfs_extlen_t      sb_logblocks;
    __uint16_t        sb_versionnum;
    __uint16_t        sb_sectsize;
    __uint16_t        sb_inodesize;
    __uint16_t        sb_inopblock;
    char              sb_fname[12];
    __uint8_t         sb_blocklog;
    __uint8_t         sb_sectlog;
    __uint8_t         sb_inodelog;
    __uint8_t         sb_inopblog;
    __uint8_t         sb_agblklog;
    __uint8_t         sb_rextslog;
    __uint8_t         sb_inprogress;
    __uint8_t         sb_imax_pct;
    __uint64_t        sb_icount;
    __uint64_t        sb_ifree;
    __uint64_t        sb_fdblocks;
    __uint64_t        sb_frextents;
    xfs_ino_t         sb_uquotino;
    xfs_ino_t         sb_gquotino;
    __uint16_t        sb_qflags;
    __uint8_t         sb_flags;
    __uint8_t         sb_shared_vn;
    xfs_extlen_t      sb_inoalignmt;
    __uint32_t        sb_unit;
    __uint32_t        sb_width;
    __uint8_t         sb_dirblklog;
    __uint8_t         sb_logsectlog;
    __uint16_t        sb_logsectsize;
    __uint32_t        sb_logsunit;
    __uint32_t        sb_features2;
} xfs_sb_t;
```

sb_magicnum

Identifies the filesystem. It's value is `XFS_SB_MAGIC = 0x58465342 "XFSB"`.

sb_blocksize

The size of a basic unit of space allocation in bytes. Typically, this is 4096 (4KB) but can range from 512 to 65536 bytes.

sb_dblocks

Total number of blocks available for data and metadata on the filesystem.

sb_rblocks

Number blocks in the real-time disk device. Refer to [Real-time Devices](#) for more information.

sb_rextents

Number of extents on the real-time device.

sb_uuid

UUID (Universally Unique ID) for the filesystem. Filesystems can be mounted by the UUID instead of device name.

sb_logstart

First block number for the journaling log if the log is internal (ie. not on a separate disk device). For an external log device, this will be zero (the log will also start on the first block on the log device).

sb_rootino

Root inode number for the filesystem. Typically, this is 128 when using a 4KB block size.

sb_rbmino

Bitmap inode for real-time extents.

sb_rsumino

Summary inode for real-time bitmap.

sb_rextsize

Realtime extent size in blocks.

sb_agblocks

Size of each AG in blocks. For the actual size of the last AG, refer to the [AG Freespace Block's](#) `agf_length` value.

sb_agcount

Number of AGs in the filesystem.

sb_rbmblocks

Number of real-time bitmap blocks.

sb_logblocks

Number of blocks for the journaling log.

sb_versionnum

Filesystem version number. This is a bitmask specifying the features enabled when creating the filesystem. Any disk checking tools or drivers that do not recognize any set bits must not operate upon the filesystem. Most of the flags indicate features introduced over time. The value must be 4 including the following flags:

Flag	Description
XFS_SB_VERSION_ATTRBIT	Set if any inode have extended attributes.

XFS_SB_VERSION_NLINKBIT	Set if any inodes use 32-bit <code>di_nlink</code> values.
XFS_SB_VERSION_QUOTABIT	Quotas are enabled on the filesystem. This also brings in the various quota fields in the superblock.
XFS_SB_VERSION_ALIGNBIT	Set if <code>sb_inoalignmt</code> is used.
XFS_SB_VERSION_DALIGNBIT	Set if <code>sb_unit</code> and <code>sb_width</code> are used.
XFS_SB_VERSION_SHAREDBIT	Set if <code>sb_shared_vn</code> is used.
XFS_SB_VERSION_LOGV2BIT	Version 2 journaling logs are used.
XFS_SB_VERSION_SECTORBIT	Set if <code>sb_sectsize</code> is not 512.
XFS_SB_VERSION_EXTFLGBIT	Unwritten extents are used. This is always set.
XFS_SB_VERSION_DIRV2BIT	Version 2 directories are used. This is always set.
XFS_SB_VERSION_MOREBITSBIT	Set if the <code>sb_features2</code> field in the superblock contains more flags.

sb_sectsize

Specifies the underlying disk sector size in bytes. Majority of the time, this is 512 bytes. This determines the minimum I/O alignment including Direct I/O.

sb_inodesize

Size of the inode in bytes. The default is 256 (2 inodes per standard sector) but can be made as large as 2048 bytes when creating the filesystem.

sb_inopblock

Number of inodes per block. This is equivalent to `sb_blocksize / sb_inodesize`.

sb_fname[12]

Name for the filesystem. This value can be used in the mount command.

sb_blocklog

\log^2 value of `sb_blocksize`. In other terms, `sb_blocksize = 2sb_blocklog`.

sb_sectlog

\log^2 value of `sb_sectsize`.

sb_inodelog

\log^2 value of `sb_inodesize`.

sb_inopblog

\log^2 value of `sb_inopblock`.

sb_agblklog

\log^2 value of `sb_agblocks` (rounded up). This value is used to generate inode numbers and absolute block numbers defined in extent maps.

sb_rextslog

\log^2 value of `sb_rextents`.

sb_inprogress

Flag specifying that the filesystem is being created.

sb_imax_pct

Maximum percentage of filesystem space that can be used for inodes. The default value is 25%.

sb_icount

Global count for number inodes allocated on the filesystem. This is only maintained in the first superblock.

sb_ifree

Global count of free inodes on the filesystem. This is only maintained in the first superblock.

sb_fdblocks

Global count of free data blocks on the filesystem. This is only maintained in the first superblock.

sb_frextents

Global count of free real-time extents on the filesystem. This is only maintained in the first superblock.

sb_uquotino

Inode for user quotas. This and the following two quota fields only apply if `XFS_SB_VERSION_QUOTABIT` flag is set in `sb_versionnum`. Refer to [Quota Inodes](#) for more information.

sb_gquotino

Inode for group or project quotas. Group and Project quotas cannot be used at the same time.

sb_qflags

Quota flags. It can be a combination of the following flags:

Flag	Description
<code>XFS_UQUOTA_ACCT</code>	User quota accounting is enabled.
<code>XFS_UQUOTA_ENFD</code>	User quotas are enforced.
<code>XFS_UQUOTA_CHKD</code>	User quotas have been checked and updated on disk.
<code>XFS_PQUOTA_ACCT</code>	Project quota accounting is enabled.
<code>XFS_OQUOTA_ENFD</code>	Other (group/project) quotas are enforced.
<code>XFS_OQUOTA_CHKD</code>	Other (group/project) quotas have been checked.
<code>XFS_GQUOTA_ACCT</code>	Group quota accounting is enabled.

sb_flags

Miscellaneous flags.

sb_shared_vn

Reserved and must be zero ("vn" stands for version number).

sb_inoalignmt

Inode chunk alignment in fsblocks.

sb_unit

Underlying stripe or raid unit in blocks.

sb_width

Underlying stripe or raid width in blocks.

sb_dirblklog

\log^2 value multiplier that determines the granularity of directory block allocations in fsblocks.

sb_logsectlog

\log^2 value of the log subvolume's sector size. This is only used if the journaling log is on a separate disk device (i.e. not internal).

sb_logsectsize

The log's sector size in bytes if the filesystem uses an external log device.

sb_logsunit

The log device's stripe or raid unit size. This only applies to version 2 logs (XFS_SB_VERSION_LOGV2BIT is set in sb_versionnum).

sb_features2

Additional version flags if XFS_SB_VERSION_MOREBITSBIT is set in sb_versionnum. The currently defined additional features include:

- XFS_SB_VERSION2_LAZYSB_COUNTBIT (0x02): Lazy global counters. Making a filesystem with this bit set can improve performance. The global free space and inode counts are only updated in the primary superblock when the filesystem is cleanly unmounted.
- XFS_SB_VERSION2_ATTR2BIT (0x08): Extended attributes version 2. Making a filesystem with this optimises the inode layout of extended attributes.
- XFS_SB_VERSION2_PARENTBIT (0x10): Parent pointers. All inodes must have an extended attribute that points back to its parent inode. The primary purpose for this information is in backup systems.

xfstools Example:

A filesystem is made on a single SATA disk with the following command:

```
# mkfs.xfs -i attr=2 -n size=16384 -f /dev/sda7
meta-data=/dev/sda7          isize=256    agcount=16, agsize=3923122 blks
=                               sectsz=512   attr=2
data      =                   bsize=4096  blocks=62769952, imaxpct=25
=                               sunit=0     swidth=0 blks, unwritten=1
naming    =version 2         bsize=16384
log       =internal log     bsize=4096  blocks=30649, version=1
=                               sectsz=512   sunit=0 blks
realtime  =none              extsz=65536 blocks=0, rtextents=0
```

And in `xfs_db`, inspecting the superblock:

```
xfs_db> sb
xfs_db> p
magicnum = 0x58465342
blocksize = 4096
dblocks = 62769952
rblocks = 0
rextents = 0
uuid = 32b24036-6931-45b4-b68c-cd5e7d9a1ca5
logstart = 33554436
rootino = 128
rbmino = 129
rsumino = 130
rextsize = 16
agblocks = 3923122
agcount = 16
rbmblocks = 0
logblocks = 30649
versionnum = 0xb084
sectsize = 512
inodesize = 256
inopblock = 16
fname = "\000\000\000\000\000\000\000\000\000\000\000\000"
blocklog = 12
sectlog = 9
inodelog = 8
inopblog = 4
agblklog = 22
rextslog = 0
inprogress = 0
imax_pct = 25
icount = 64
ifree = 61
fdblocks = 62739235
fextents = 0
uquotino = 0
gquotino = 0
qflags = 0
flags = 0
shared_vn = 0
inoalignmt = 2
unit = 0
width = 0
dirblklog = 2
logsectlog = 0
logsectsize = 0
logsunit = 0
features2 = 8
```

AG Free Space Management

The XFS filesystem tracks free space in an allocation group using two B+trees. One B+tree tracks space by block number, the second by the size of the free space block. This scheme allows XFS to quickly find free space near a given block or of a given size.

All block numbers, indexes and counts are AG relative.

AG Free Space Block

The second sector in an AG contains the information about the two free space B+trees and associated free space information for the AG. The "AG Free Space Block", also known as the AGF, uses the following structure:

```
typedef struct xfs_agf {
    __be32          agf_magicnum;
    __be32          agf_versionnum;
    __be32          agf_seqno;
    __be32          agf_length;
    __be32          agf_roots[XFS_BTNUM_AGF];
    __be32          agf_spare0;
    __be32          agf_levels[XFS_BTNUM_AGF];
    __be32          agf_spare1;
    __be32          agf_flfirst;
    __be32          agf_fllast;
    __be32          agf_flcount;
    __be32          agf_freeblks;
    __be32          agf_longest;
    __be32          agf_btreeblks;
} xfs_agf_t;
```

The rest of the bytes in the sector are zeroed. `XFS_BTNUM_AGF` is set to 2, index 0 for the count B+tree and index 1 for the size B+tree.

agf_magicnum

Specifies the magic number for the AGF sector: "XAGF" (0x58414746).

agf_versionnum

Set to `XFS_AGF_VERSION` which is currently 1.

agf_seqno

Specifies the AG number for the sector.

agf_length

Specifies the size of the AG in filesystem blocks. For all AGs except the last, this must be equal to the superblock's `sb_agblocks` value. For the last AG, this could be less than the `sb_agblocks` value. It is this value that should be used to determine the size of the AG.

agf_roots

Specifies the block number for the root of the two free space B+trees.

agf_levels

Specifies the level or depth of the two free space B+trees. For a fresh AG, this will be one, and the "roots" will point to a single leaf of level 0.

agf_ffirst

Specifies the index of the first "free list" block. Free lists are covered in more detail later on.

agf_fllast

Specifies the index of the last "free list" block.

agf_flicount

Specifies the number of blocks in the "free list".

agf_freblks

Specifies the current number of free blocks in the AG.

agf_longest

Specifies the number of blocks of longest contiguous free space in the AG.

agf_btreblks

Specifies the number of blocks used for the free space B+trees. This is only used if the `XFS_SB_VERSION2_LAZYSB_COUNTBIT` bit is set in `sb_features2`.

AG Free Space B+trees

The two Free Space B+trees store a sorted array of block offset and block counts in the leaves of the B+tree. The first B+tree is sorted by the offset, the second by the count or size.

The trees use the following header:

```
typedef struct xfs_btree_sblock {
    __be32          bb_magic;
    __be16          bb_level;
    __be16          bb_numrecs;
    __be32          bb_leftsib;
    __be32          bb_rightsib;
} xfs_btree_sblock_t;
```

Leaves contain a sorted array of offset/count pairs which are also used for node keys:

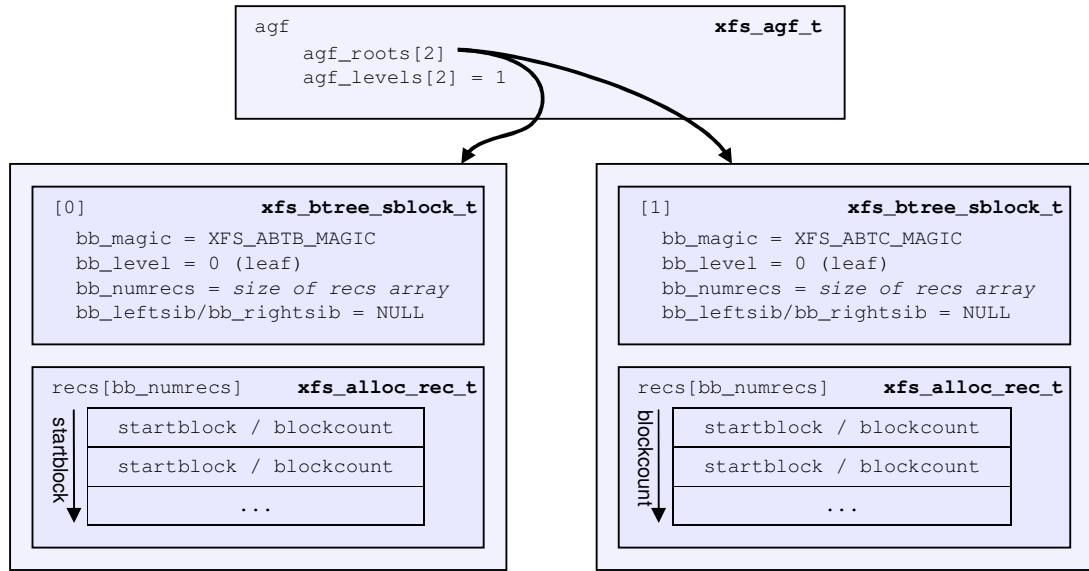
```
typedef struct xfs_alloc_rec {
    __be32          ar_startblock;
    __be32          ar_blockcount;
} xfs_alloc_rec_t, xfs_alloc_key_t;
```

Node pointers are an AG relative block pointer:

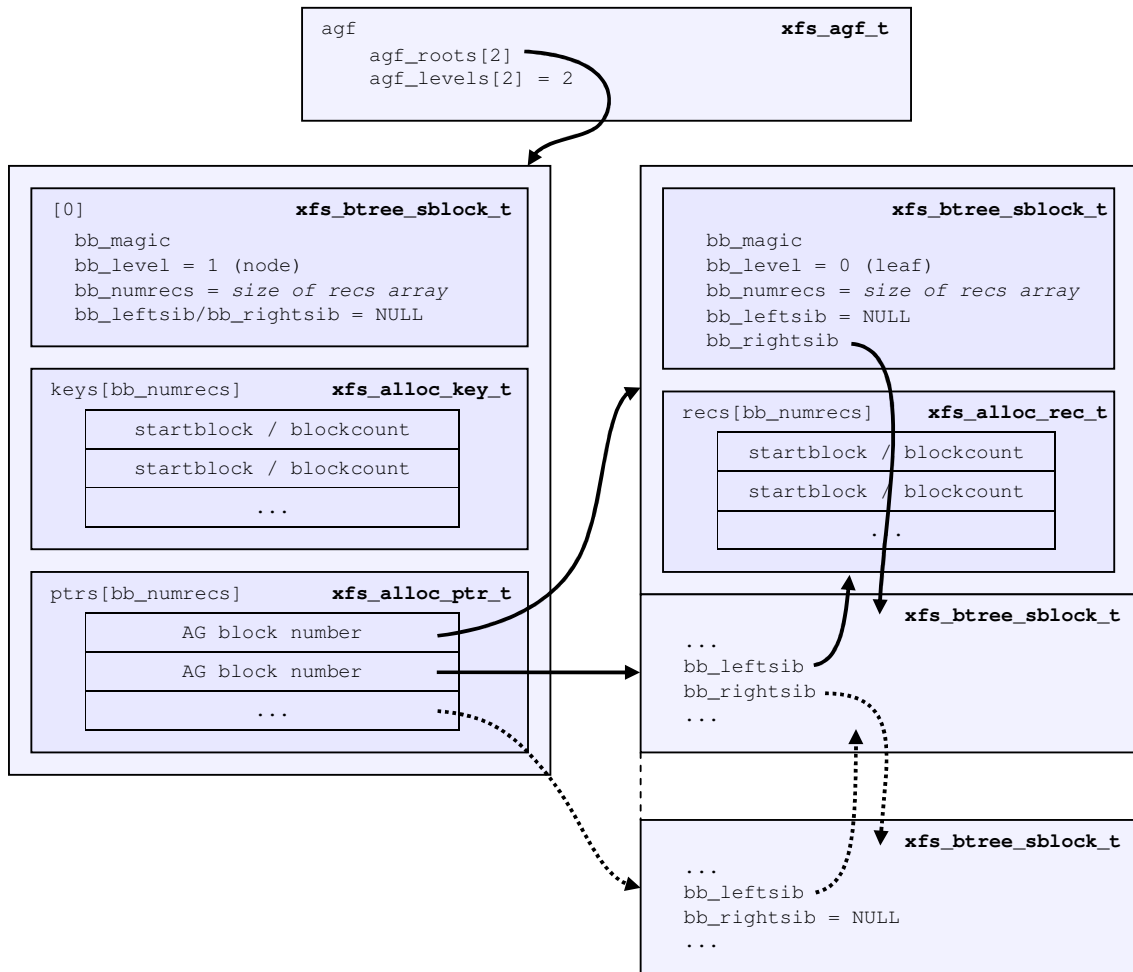
```
typedef __be32 xfs_alloc_ptr_t;
```

- As the free space tracking is AG relative, all the block numbers are only 32-bits.
- The `bb_magic` value depends on the B+tree: "ABTB" (0x41425442) for the block offset B+tree, "ABTC" (0x41425443) for the block count B+tree.
- The `xfs_btree_sblock_t` header is used for intermediate B+tree node as well as the leaves.
- For a typical 4KB filesystem block size, the offset for the `xfs_alloc_ptr_t` array would be 0xab0 (2736 decimal).
- There are a series of macros in `xfs_btree.h` for deriving the offsets, counts, maximums, etc for the B+trees used in XFS.

The following diagram shows a single level B+tree which consists of one leaf:



With the intermediate nodes, the associated leaf pointers are stored in a separate array about two thirds into the block. The following diagram illustrates a 2-level B+tree for a free space B+tree:



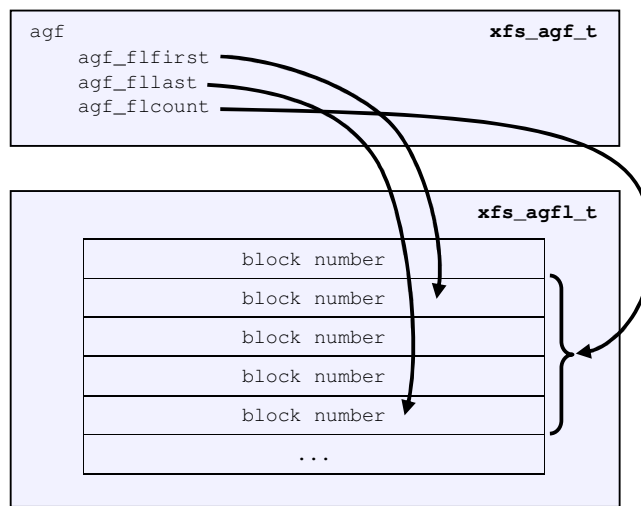
AG Free List

The AG Free List is located in the 4th sector of each AG and is known as the AGFL. It is an array of AG relative block pointers for reserved space for growing the free space B+trees. This space cannot be used for general user data including inodes, data, directories and extended attributes.

With a freshly made filesystem, 4 blocks are reserved immediately after the free space B+tree root blocks (blocks 4 to 7). As they are used up as the free space fragments, additional blocks will be reserved from the AG and added to the free list array.

As the free list array is located within a single sector, a typical device will have space for 128 elements in the array (512 bytes per sector, 4 bytes per AG relative block pointer). The actual size can be determined by using the `XFS_AGFL_SIZE` macro.

Active elements in the array are specified by the [AGF's](#) `agf_flfirst`, `agf_fllast` and `agf_flcount` values. The array is managed as a circular list.



The presence of these reserved block guarantees that the free space B+trees can be updated if any blocks are freed by extent changes in a full AG.

xfs_db Examples:

These examples are derived from an AG that has been deliberately fragmented.

The AGF:

```
xfs_db> agf <ag#>
xfs_db> p
magicnum = 0x58414746
versionnum = 1
seqno = 0
length = 3923122
bnoroot = 7
cntroot = 83343
bnolevel = 2
cntlevel = 2
flfirst = 22
fllast = 27
flcount = 6
freeblks = 3654234
longest = 3384327
btreeblks = 0
```


In the AGFL, the active elements are from 22 to 27 inclusive which are obtained from the `flfirst` and `fllast` values from the `agf` in the previous example:

```
xfs_db> agfl 0
xfs_db> p
bno[0-127] = 0:4 1:5 2:6 3:7 4:83342 5:83343 6:83344 7:83345 8:83346 9:83347
            10:4 11:5 12:80205 13:80780 14:81496 15:81766 16:83346 17:4 18:5
            19:80205 20:82449 21:81496 22:81766 23:82455 24:80780 25:5
            26:80205 27:83344
```

The free space B+tree sorted by block offset, the root block is from the AGF's `bnoroot` value:

```
xfs_db> fsblock 7
xfs_db> type bnobt
xfs_db> p
magic = 0x41425442
level = 1
numrecs = 4
leftsib = null
rightsib = null
keys[1-4] = [startblock,blockcount]
            1:[12,16] 2:[184586,3] 3:[225579,1] 4:[511629,1]
ptrs[1-4] = 1:2 2:83347 3:6 4:4
```

Blocks 2, 83347, 6 and 4 contain the leaves for the free space B+tree by starting block. Block 2 would contain offsets 16 up to but not including 184586 while block 4 would have all offsets from 511629 to the end of the AG.

The free space B+tree sorted by block count, the root block is from the AGF's `cntroot` value:

```
xfs_db> fsblock 83343
xfs_db> type cntbt
xfs_db> p
magic = 0x41425443
level = 1
numrecs = 4
leftsib = null
rightsib = null
keys[1-4] = [blockcount,startblock]
            1:[1,81496] 2:[1,511729] 3:[3,191875] 4:[6,184595]
ptrs[1-4] = 1:3 2:83345 3:83342 4:83346
```

The leaf in block 3, in this example, would only contain single block counts. The offsets are sorted in ascending order if the block count is the same.

Inspecting the leaf in block 83346, we can see the largest block at the end:

```
xfs_db> fsblock 83346
xfs_db> type cntbt
xfs_db> p
magic = 0x41425443
level = 0
numrecs = 344
leftsib = 83342
rightsib = null
recs[1-344] = [startblock,blockcount]
              1:[184595,6] 2:[187573,6] 3:[187776,6]
              ...
              342:[513712,755] 343:[230317,258229] 344:[538795,3384327]
```

The longest block count must be the same as the AGF's longest value.

AG Inode Management

Inode Numbers

Inode numbers in XFS come in two forms: AG relative and absolute.

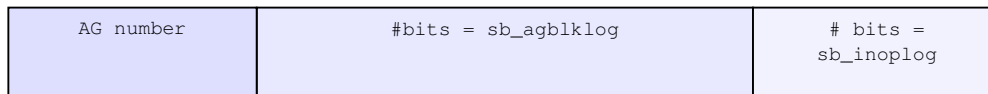
AG relative inode numbers always fit within 32 bits. The number of bits actually used is determined by the sum of the [superblock's](#) `sb_inoplog` and `sb_agblklog` values. Relative inode numbers are found within the AG's inode structures.

Absolute inode numbers include the AG number in the high bits, above the bits used for the AG relative inode number. Absolute inode numbers are found in [directory](#) entries.

Relative Inode number format



Absolute Inode number format



MSB

LSB

Inode Information

Each AG manages its own inodes. The third sector in the AG contains information about the AG's inodes and is known as the AGI.

The AGI uses the following structure:

```
typedef struct xfs_agi {
    __be32          agi_magicnum;
    __be32          agi_versionnum;
    __be32          agi_seqno;
    __be32          agi_length;
    __be32          agi_count;
    __be32          agi_root;
    __be32          agi_level;
    __be32          agi_freecount;
    __be32          agi_newino;
    __be32          agi_dirino;
    __be32          agi_unlinked[64];
} xfs_agi_t;
```

agi_magicnum

Specifies the magic number for the AGI sector: "XAGI" (0x58414749).

agi_versionnum

Set to `XFS_AGI_VERSION` which is currently 1.

agi_seqno

Specifies the AG number for the sector.

agi_length

Specifies the size of the AG in filesystem blocks.

agi_count

Specifies the number of inodes allocated for the AG.

agi_root

Specifies the block number in the AG containing the root of the inode B+tree.

agi_level

Specifies the number of levels in the inode B+tree.

agi_freecount

Specifies the number of free inodes in the AG.

agi_newino

Specifies AG relative inode number most recently allocated.

agi_dirino

Deprecated and not used, it's always set to NULL (-1).

agi_unlinked[64]

Hash table of unlinked (deleted) inodes that are still being referenced. Refer to [Inode Unlinked Pointer](#) for more information.

Inode B+trees

Inodes are allocated in chunks of 64, and a B+tree is used to track these chunks of inodes as they are allocated and freed. The block containing root of the B+tree is defined by the AGI's `agi_root` value.

The B+tree header for the nodes and leaves use the `xfst_btree_sblock` structure which is the same as the header used in the [AGF B+trees](#):

```
typedef struct xfst_btree_sblock xfst_inobt_block_t;
```

Leaves contain an array of the following structure:

```
typedef struct xfst_inobt_rec {
    __be32          ir_startino;
    __be32          ir_freecount;
    __be64          ir_free;
} xfst_inobt_rec_t;
```

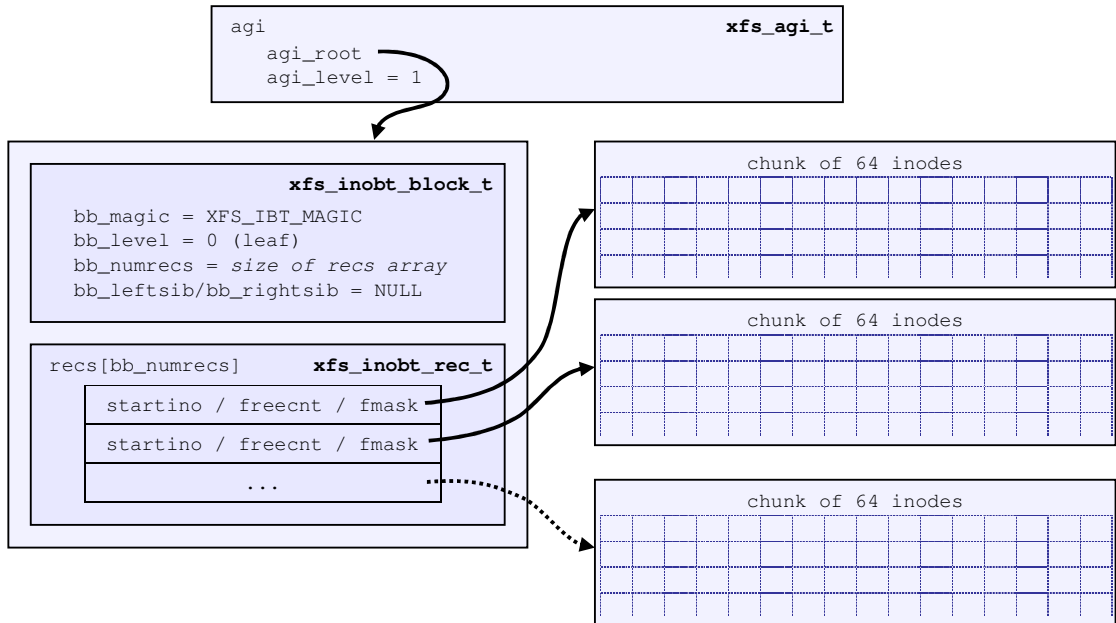
Nodes contain key/pointer pairs using the following types:

```
typedef struct xfst_inobt_key {
    __be32          ir_startino;
} xfst_inobt_key_t;
```

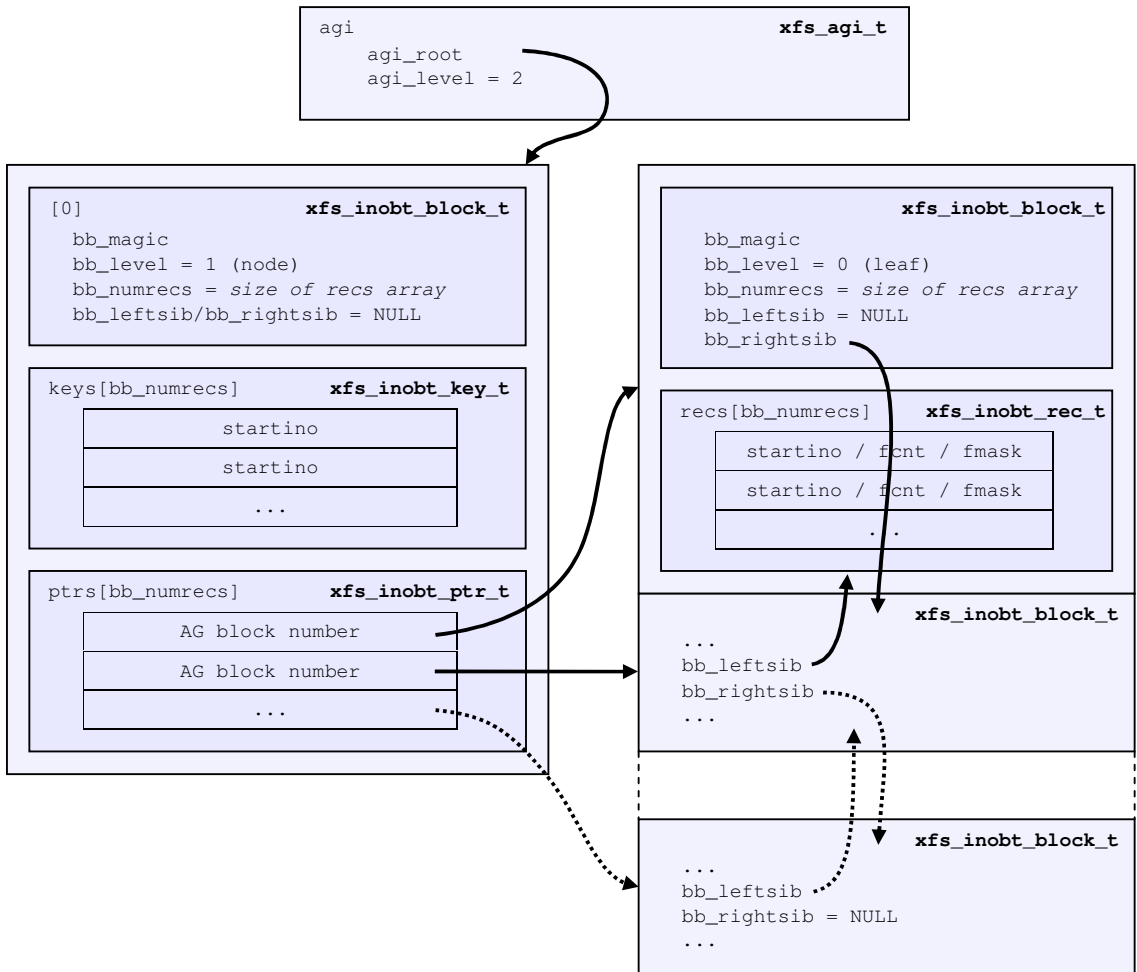
```
typedef __be32 xfst_inobt_ptr_t;
```

For the leaf entries, `ir_startino` specifies the starting inode number for the chunk, `ir_freecount` specifies the number of free entries in the chunk, and the `ir_free` is a 64 element bit array specifying which entries are free in the chunk.

The following diagram illustrates a single level inode B+tree:



And a 2-level inode B+tree:



xf_s_db Examples:

TODO:

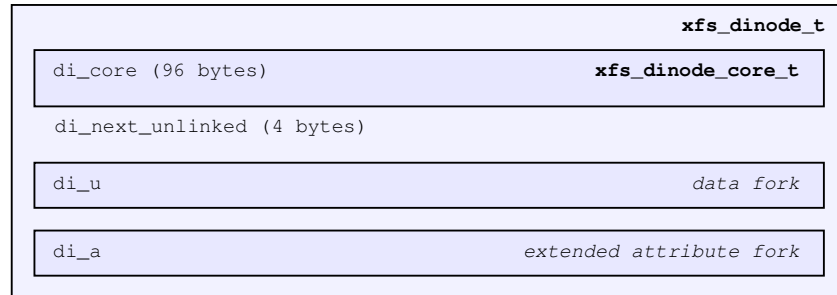
Real-time Devices

TODO:

On-disk Inode

All files, directories and links are stored on disk with inodes and descend from the root inode with its number defined in the [superblock](#). The previous section on [AG Inode Management](#) describes the allocation and management of inodes on disk. This section describes the contents of inodes themselves.

An inode is divided into 3 parts:



- The core contains what the inode represents, stat data and information describing the data and attribute forks.
- The `di_u` "data fork" contains normal data related to the inode. Its contents depends on the file type specified by `di_core.di_mode` (eg. regular file, directory, link, etc) and how much information is contained in the file which determined by `di_core.di_format`. The following union to represent this data is declared as follows:

```

union {
    xfs_bmdr_block_t    di_bmbt;
    xfs_bmbt_rec_t     di_bmx[1];
    xfs_dir2_sf_t      di_dir2sf;
    char                di_c[1];
    xfs_dev_t          di_dev;
    uuid_t              di_muuid;
    char                di_symlink[1];
} di_u;

```

- The `di_a` "attribute fork" contains extended attributes. Its layout is determined by the `di_core.di_aformat` value. Its representation is declared as follows:

```

union {
    xfs_bmdr_block_t    di_abmbt;
    xfs_bmbt_rec_t     di_abmx[1];
    xfs_attr_shortform_t di_attrsf;
} di_a;

```

Note: The above two unions are rarely used in the XFS code, but the structures within the union are directly cast depending on the `di_mode/di_format` and `di_aformat` values. They are referenced in this document to make it easier to explain the various structures in use within the inode.

The remaining space in the inode after `di_next_unlinked` where the two forks are located is called the inode's "literal area". This starts at offset 100 (0x64) in the inode.

The space for each of the two forks in the literal area is determined by the inode size, and `di_core.di_forkoff`. The data fork is located between the start of the literal area and `di_forkoff`. The attribute fork is located between `di_forkoff` and the end of the inode.

Inode Core

The inode's core is 96 bytes in size and contains information about the file itself including most stat data information about data and attribute forks after the core within the inode. It uses the following structure:

```
typedef struct xfs_dinode_core {
    __uint16_t      di_magic;
    __uint16_t      di_mode;
    __int8_t        di_version;
    __int8_t        di_format;
    __uint16_t      di_onlink;
    __uint32_t      di_uid;
    __uint32_t      di_gid;
    __uint32_t      di_nlink;
    __uint16_t      di_projid;
    __uint8_t       di_pad[8];
    __uint16_t      di_flushiter;
    xfs_timestamp_t di_atime;
    xfs_timestamp_t di_mtime;
    xfs_timestamp_t di_ctime;
    xfs_fsize_t     di_size;
    xfs_drfsbno_t   di_nblocks;
    xfs_extlen_t    di_extsize;
    xfs_extnum_t    di_nextents;
    xfs_aextnum_t   di_anextents;
    __uint8_t       di_forkoff;
    __int8_t        di_aformat;
    __uint32_t      di_dmevmask;
    __uint16_t      di_dmstate;
    __uint16_t      di_flags;
    __uint32_t      di_gen;
} xfs_dinode_core_t;
```

di_magic

The inode signature where these two bytes are 0x494e, or "IN" in ASCII.

di_mode

Specifies the mode access bits and type of file using the standard S_lxxx values defined in stat.h.

di_version

Specifies the inode version which currently can only be 1 or 2. The inode version specifies the usage of the `di_onlink`, `di_nlink` and `di_projid` values in the inode core. Initially, inodes are created as v1 but can be converted on the fly to v2 when required.

di_format

Specifies the format of the data fork in conjunction with the `di_mode` type. This can be one of several values. For directories and links, it can be "local" where all metadata associated with the file is within the inode, "extents" where the inode contains an array of extents to other filesystem blocks which contain the associated metadata or data or "btree" where the inode contains a B+tree root node which points to filesystem blocks containing the metadata or data. Migration between the formats depends on the amount of metadata associated with the inode. "dev" is used for character and block devices while "uuid" is currently not used.

```
typedef enum xfs_dinode_fmt {
    XFS_DINODE_FMT_DEV,
    XFS_DINODE_FMT_LOCAL,
    XFS_DINODE_FMT_EXTENTS,
    XFS_DINODE_FMT_BTREE,
}
```



```

        XFS_DINODE_FMT_UUID
    } xfs_dinode_fmt_t;

```

di_onlink

In v1 inodes, this specifies the number of links to the inode from directories. When the number exceeds 65535, the inode is converted to v2 and the link count is stored in `di_nlink`.

di_uid

Specifies the owner's UID of the inode.

di_gid

Specifies the owner's GID of the inode.

di_nlink

Specifies the number of links to the inode from directories. This is maintained for both inode versions for current versions of XFS. Old versions of XFS did not support v2 inodes, and therefore this value was never updated and was classed as reserved space (part of `di_pad`).

di_projid

Specifies the owner's project ID in v2 inodes. An inode is converted to v2 if the project ID is set. This value must be zero for v1 inodes.

di_pad[8]

Reserved, must be zero.

di_flushiter

Incremented on flush.

di_atime

Specifies the last access time of the files using UNIX time conventions the following structure. This value maybe undefined if the filesystem is mounted with the "noatime" option.

```

typedef struct xfs_timestamp {
    __int32_t          t_sec;
    __int32_t          t_nsec;
} xfs_timestamp_t;

```

di_mtime

Specifies the last time the file was modified.

di_ctime

Specifies when the inode's status was last changed.

di_size

Specifies the EOF of the inode in bytes. This can be larger or smaller than the extent space (therefore actual disk space) used for the inode. For regular files, this is the filesize in bytes, directories, the space taken by directory entries and for links, the length of the symlink.

di_nblocks

Specifies the number of filesystem blocks used to store the inode's data including relevant metadata like B+trees. This does not include blocks used for extended attributes.

di_extsize

Specifies the extent size for filesystems with real-time devices and an extent size hint for standard filesystems. For normal filesystems, and with directories, the `XFS_DIFLAG_EXTSZINHERIT` flag must be set in `di_flags` if this field is used. Inodes created in these directories will inherit the `di_extsize` value and have `XFS_DIFLAG_EXTSIZE` set in their `di_flags`. When a file is written to beyond allocated space, XFS will attempt to allocate additional disk space based on this value.

di_nextents

Specifies the number of data extents associated with this inode.

di_anextents

Specifies the number of extended attribute extents associated with this inode.

di_forkoff

Specifies the offset into the inode's literal area where the extended attribute fork starts. This is an 8-bit value that is multiplied by 8 to determine the actual offset in bytes (ie. attribute data is 64-bit aligned). This also limits the maximum size of the inode to 2048 bytes. This value is initially zero until an extended attribute is created. When an attribute is added, the nature of `di_forkoff` depends on the `XFS_SB_VERSION2_ATTR2BIT` flag in the superblock. Refer to the [Extended Attribute Versions](#) section for more details.

di_aformat

Specifies the format of the attribute fork. This uses the same values as `di_format`, but restricted to "local", "extents" and "btree" formats for extended attribute data.

di_dmevmask

DMAPI event mask.

di_dmstate

DMAPI state.

di_flags

Specifies flags associated with the inode. This can be a combination of the following values:

Flag	Description
<code>XFS_DIFLAG_REALTIME</code>	The inode's data is located on the real-time device.
<code>XFS_DIFLAG_PREALLOC</code>	The inode's extents have been preallocated.
<code>XFS_DIFLAG_NEWRTBM</code>	Specifies the <code>sb_rbmno</code> uses the new real-time bitmap format
<code>XFS_DIFLAG_IMMUTABLE</code>	Specifies the inode cannot be modified.
<code>XFS_DIFLAG_APPEND</code>	The inode is in append only mode.
<code>XFS_DIFLAG_SYNC</code>	The inode is written synchronously.
<code>XFS_DIFLAG_NOATIME</code>	The inode's <code>di_atime</code> is not updated.
<code>XFS_DIFLAG_NODUMP</code>	Specifies the inode is to be ignored by <code>xfsdump</code> .
<code>XFS_DIFLAG_RTINHERIT</code>	For directory inodes, new inodes inherit the <code>XFS_DIFLAG_REALTIME</code> bit.

XFS_DIFLAG_PROJINHERIT	For directory inodes, new inodes inherit the <code>di_projid</code> value.
XFS_DIFLAG_NOSYMLINKS	For directory inodes, symlinks cannot be created.
XFS_DIFLAG_EXTSIZE	Specifies the extent size for real-time files or a and extent size hint for regular files.
XFS_DIFLAG_EXTSZINHERIT	For directory inodes, new inodes inherit the <code>di_extsize</code> value.
XFS_DIFLAG_NODEFRAG	Specifies the inode is to be ignored when defragmenting the filesystem.

di_gen

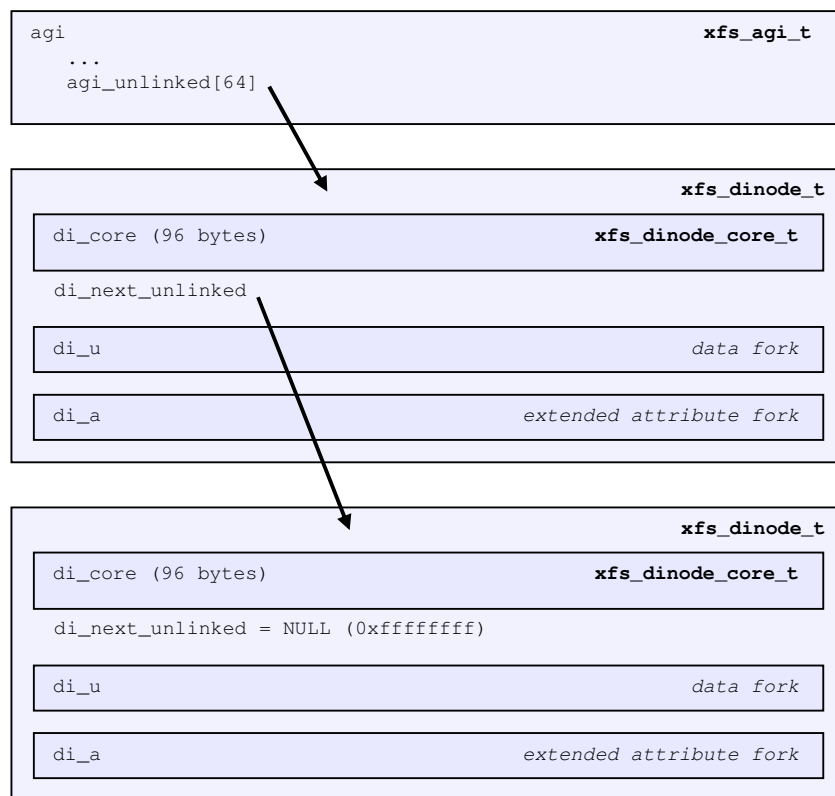
A generation number used for inode identification. This is used by tools that do inode scanning such as backup tools and `xfsdump`. An inode's generation number can change by unlinking and creating a new file that reuses the inode.

Unlinked Pointer

The `di_next_unlinked` value in the inode is used to track inodes that have been unlinked (deleted) but which are still referenced. When an inode is unlinked and there is still an outstanding reference, the inode is added to one of the [AGI's](#) `agi_unlinked` hash buckets. The AGI unlinked bucket points to an inode and the `di_next_unlinked` value points to the next inode in the chain. The last inode in the chain has `di_next_unlinked` set to NULL (-1).

Once the last reference is released, the inode is removed from the unlinked hash chain, and `di_next_unlinked` is set to NULL. In the case of a system crash, XFS recovery will complete the unlink process for any inodes found in these lists.

The only time the unlinked fields can be seen to be used on disk is either on an active filesystem or a crashed system. A cleanly unmounted or recovered filesystem will not have any inodes in these unlink hash chains.



Data Fork

The structure of the inode's data fork based is on the inode's type and `di_format`. It always starts at offset 100 (0x64) in the inode's space which is the start of the inode's "literal area". The size of the data fork is determined by the type and format. The maximum size is determined by the inode size and `di_forkoff`. In code, use the `XFS_DFORK_PTR` macro specifying `XFS_DATA_FORK` for the "which" parameter. Alternatively, the `XFS_DFORK_DPTR` macro can be used.

Each of the following sub-sections summarises the contents of the data fork based on the inode type.

Regular Files (S_IFREG)

The data fork specifies the file's data extents. The extents specify where the file's actual data is located within the filesystem. Extents can have 2 formats which is defined by the `di_format` value:

- `XFS_DINODE_FMT_EXTENTS`: The extent data is fully contained within the inode which contains an array of extents to the filesystem blocks for the file's data. To access the extents, cast the return value from `XFS_DFORK_DPTR` to `xfst_bmbt_rec_t*`.
- `XFS_DINODE_FMT_BTREE`: The extent data is contained in the leaves of a B+tree. The inode contains the root node of the tree and is accessed by casting the return value from `XFS_DFORK_DPTR` to `xfst_bmdr_block_t*`.

Details for each of these data extent formats are covered in the [Data Extents](#) section later on.

Directories (S_IFDIR)

The data fork contains the directory's entries and associated data. The format of the entries is also determined by the `di_format` value and can be one of 3 formats:

- `XFS_DINODE_FMT_LOCAL`: The directory entries are fully contained within the inode. This is accessed by casting the value from `XFS_DFORK_DPTR` to `xfst_dir2_sf_t*`.
- `XFS_DINODE_FMT_EXTENTS`: The actual directory entries are located in another filesystem block, the inode contains an array of extents to these filesystem blocks (`xfst_bmbt_rec_t*`).
- `XFS_DINODE_FMT_BTREE`: The directory entries are contained in the leaves of a B+tree. The inode contains the root node (`xfst_bmdr_block_t*`).

Details for each of these directory formats are covered in the [Directories](#) section later on.

Symbolic Links (S_IFLNK)

The data fork contains the contents of the symbolic link. The format of the link is determined by the `di_format` value and can be one of 2 formats:

- `XFS_DINODE_FMT_LOCAL`: The symbolic link is fully contained within the inode. This is accessed by casting the return value from `XFS_DFORK_DPTR` to `char*`.
- `XFS_DINODE_FMT_EXTENTS`: The actual symlink is located in another filesystem block, the inode contains the extents to these filesystem blocks (`xfst_bmbt_rec_t*`).

Details for symbolic links is covered in the [Symbolic Links](#) section later on.

Other File Types

For character and block devices (`S_IFCHR` and `S_IFBLK`), cast the value from `XFS_DFORK_DPTR` to `xfst_dev_t*`.

Attribute Fork

The attribute fork in the inode always contains the location of the extended attributes associated with the inode.

The location of the attribute fork in the inode's literal area (offset 100 to the end of the inode) is specified by the `di_forkoff` value in the inode's core. If this value is zero, the inode does not contain any extended attributes. Non-zero, the byte offset into the literal area = `di_forkoff * 8`, which also determines the 2048 byte maximum size for an inode. Attributes must be allocated on a 64-bit boundary on the disk. To access the extended attributes in code, use the `XFS_DFORK_PTR` macro specifying `XFS_ATTR_FORK` for the "which" parameter. Alternatively, the `XFS_DFORK_APTR` macro can be used.

Which structure in the attribute fork is used depends on the `di_aformat` value in the inode. It can be one of the following values:

- `XFS_DINODE_FMT_LOCAL`: The extended attributes are contained entirely within the inode. This is accessed by casting the value from `XFS_DFORK_APTR` to `xfs_attr_shortform_t*`.
- `XFS_DINODE_FMT_EXTENTS`: The attributes are located in another filesystem block, the inode contains an array of pointers to these filesystem blocks. They are accessed by casting the value from `XFS_DFORK_APTR` to `xfs_bmbt_rec_t*`.
- `XFS_DINODE_FMT_BTREE`: The extents for the attributes are contained in the leaves of a B+tree. The inode contains the root node of the tree and is accessed by casting the value from `XFS_DFORK_APTR` to `xfs_bmdr_block_t*`.

Detailed information on the layouts of extended attributes are covered in the [Extended Attributes](#) section later on in this document.

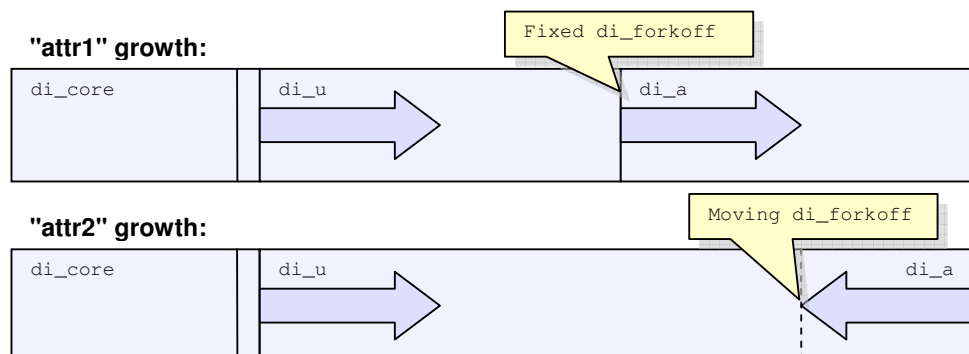
Extended Attribute Versions

Extended attributes come in two versions: "attr1" or "attr2". The attribute version is specified by the `XFS_SB_VERSION2_ATTR2BIT` flag in the `sb_features2` field in the superblock. It determines how the inode's extra space is split between `di_u` and `di_a` forks which also determines how the `di_forkoff` value is maintained in the inode's core.

With "attr1" attributes, the `di_forkoff` is set to somewhere in the middle of the space between the core and end of the inode and never changes (which has the effect of artificially limiting the space for data information). As the data fork grows, when it gets to `di_forkoff`, it will move the data to the level format level (ie. local > extent > btree). If very little space is used for either attributes or data, then a good portion of the available inode space is wasted with this version.

"Attr2" was introduced to maximum the utilisation of the inode's literal area. The `di_forkoff` starts at the end of the inode and works its way to the data fork as attributes are added. Attr2 is highly recommended if extended attributes are used.

The following diagram compares the two versions:

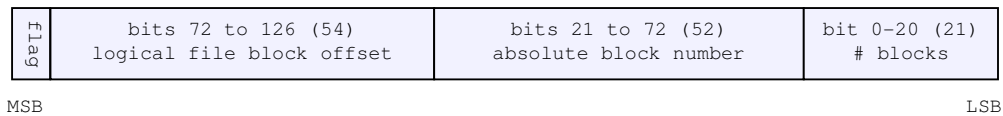


Data Extents

XFS allocates space for a file using extents: starting location and length. XFS extents also specify the file's logical starting offset for a file. This allows a file's extent map to automatically support sparse files (i.e. "holes" in the file). A flag is also used to specify if the extent has been preallocated and not yet been written to (unwritten extent).

A file can have more than one extent if one chunk of contiguous disk space is not available for the file. As a file grows, the XFS space allocator will attempt to keep space contiguous and merge extents. If more than one file is being allocated space in the same AG at the same time, multiple extents for the files will occur as the extents get interleaved. The effect of this can vary depending on the extent allocator used in the XFS driver.

An extent is 128 bits in size and uses the following packed layout:



The extent is represented by the `xfs_bmbt_rec_t` structure which uses a big endian format on-disk. In-core management of extents use the `xfs_bmbt_irec_t` structure which is the unpacked version of `xfs_bmbt_rec_t`:

```
typedef struct xfs_bmbt_irec {
    xfs_fileoff_t    br_startoff;
    xfs_fsbblock_t  br_startblock;
    xfs_filblks_t   br_blockcount;
    xfs_exntst_t    br_state;
} xfs_bmbt_irec_t;
```

The extent `br_state` field uses the following enum declaration:

```
typedef enum {
    XFS_EXT_NORM,
    XFS_EXT_UNWRITTEN,
    XFS_EXT_INVALID
} xfs_exntst_t;
```

Some other points about extents:

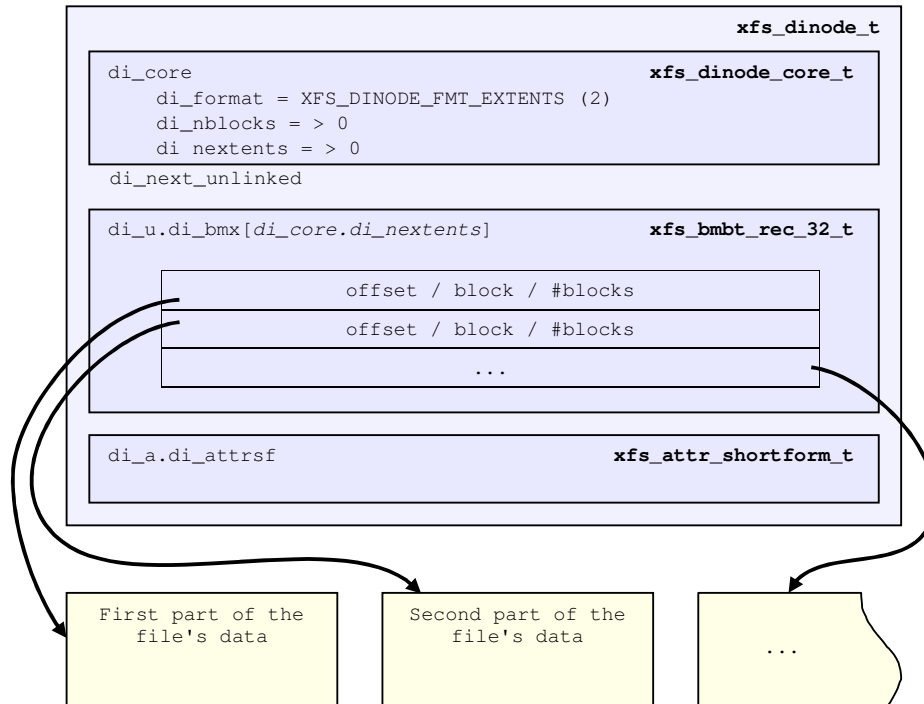
- The `xfs_bmbt_rec_32_t` and `xfs_bmbt_rec_64_t` structures are effectively the same as `xfs_bmbt_rec_t`, just different representations of the same 128 bits in on-disk big endian format.
- When a file is created and written to, XFS will endeavour to keep the extents within the same AG as the inode. It may use a different AG if the AG is busy or there is no space left in it.
- If a file is zero bytes long, it will have no extents, `di_nblocks` and `di_nexents` will be zero. Any file with data will have at least one extent, and each extent can use from 1 to over 2 million blocks (2^{21}) on the filesystem. For a default 4KB block size filesystem, a single extent can be up to 8GB in length.

The following two subsections cover the two methods of storing extent information for a file. The first is the fastest and simplest where the inode completely contains an extent array to the file's data. The second is slower and more complex B+tree which can handle thousands to millions of extents efficiently.

Extent List

Local extents are where the entire extent array is stored within the inode's data fork itself. This is the most optimal in terms of speed and resource consumption. The trade-off is the file can only have a few extents before the inode runs out of space.

The "data fork" of the inode contains an array of extents, the size of the array determined by the inode's `di_nextents` value.



The number of extents that can fit in the inode depends on the inode size and `di_forkoff`. For a default 256 byte inode with no extended attributes, a file can up to 19 extents with this format. Beyond this, extents have to use the B+tree format.

xfs_db Example:

An 8MB file with one extent:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0100644
core.version = 1
core.format = 2 (extents)
...
core.size = 8294400
core.nblocks = 2025
core.extsize = 0
core.nextents = 1
core.naextents = 0
core.forkoff = 0
...
u.bmx[0] = [startoff, startblock, blockcount, extentflag]
          0: [0, 25356, 2025, 0]
```


B+tree Extent List

Beyond the simple extent array, to efficiently manage large extent maps, XFS uses B+trees. The root node of the B+tree is stored in the inode's data fork. All block pointers for extent B+trees are 64-bit absolute block numbers.

For a single level B+tree, the root node points to the B+tree's leaves. Each leaf occupies one filesystem block and contains a header and an array of extents sorted by the file's offset. Each leaf has left and right (or backward and forward) block pointers to adjacent leaves. For a standard 4KB filesystem block, a leaf can contain up to 254 extents before a B+tree rebalance is triggered.

For a multi-level B+tree, the root node points to other B+tree nodes which eventually point to the extent leaves. B+tree keys are based on the file's offset. The nodes at each level in the B+tree point to the adjacent nodes.

The base B+tree node is used for extents, directories and extended attributes. The structures used for inode's B+tree root are:

```
typedef struct xfs_bmdr_block {
    __be16          bb_level;
    __be16          bb_numrecs;
} xfs_bmdr_block_t;

typedef struct xfs_bmbt_key {
    xfs_dfiloff_t   br_startoff;
} xfs_bmbt_key_t, xfs_bmdr_key_t;

typedef xfs_dfsbno_t xfs_bmbt_ptr_t, xfs_bmdr_ptr_t;
```

- On disk, the B+tree node starts with the `xfs_bmbt_block_t` header followed by an array of `xfs_bmbt_key_t` values and then an array of `xfs_bmbt_ptr_t` values. The size of both arrays is specified by the header's `bb_numrecs` value.
- The root node in the inode can only contain up to 19 key/pointer pairs for a standard 256 byte inode before a new level of nodes is added between the root and the leaves. This will be less if `di_forkoff` is not zero (i.e. attributes are in use on the inode).

The subsequent nodes and leaves of the B+tree use the `xfs_bmbt_block_t` declaration:

```
typedef struct xfs_btree_lblock xfs_bmbt_block_t;

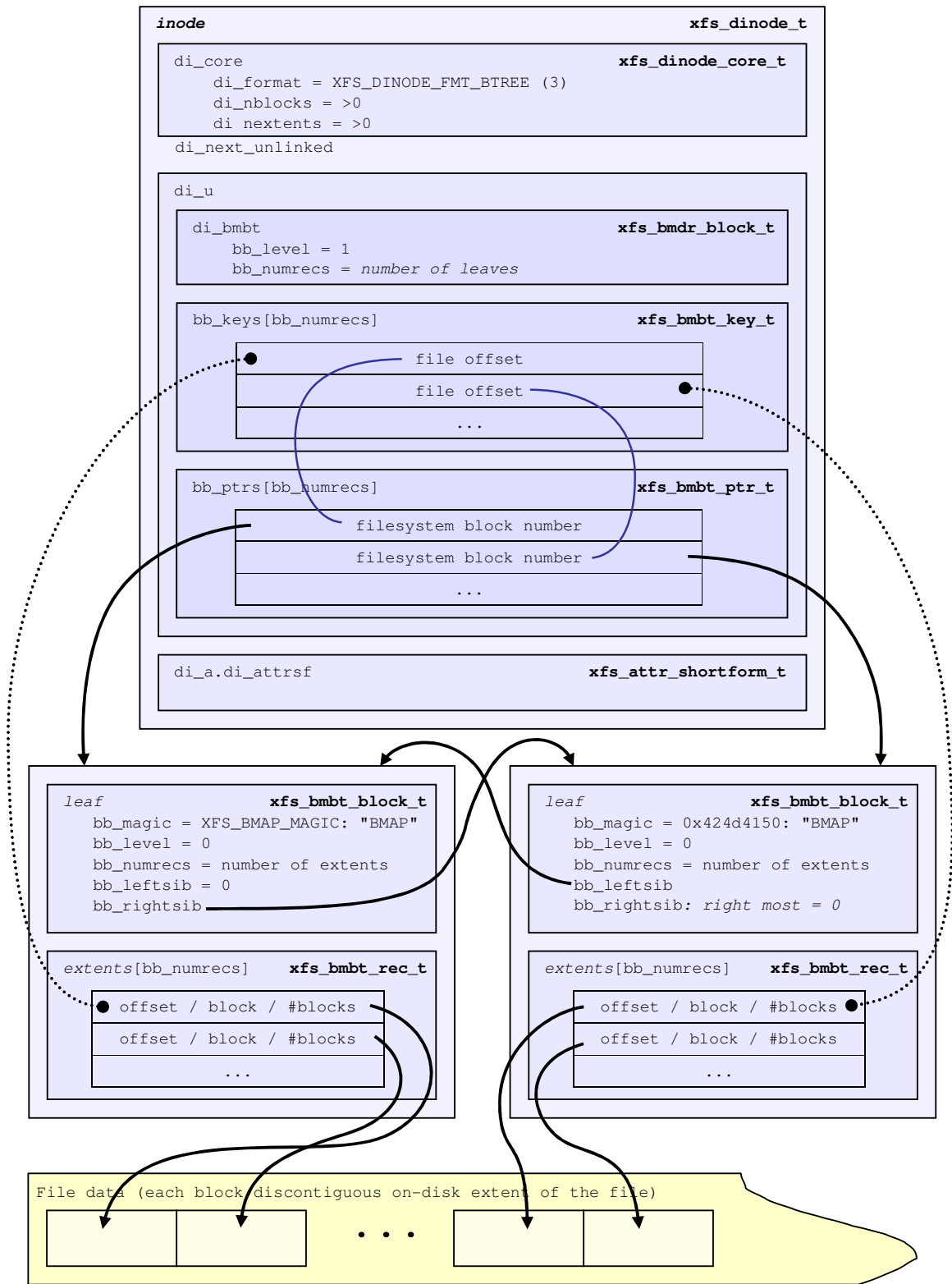
typedef struct xfs_btree_lblock {
    __be32          bb_magic;
    __be16          bb_level;
    __be16          bb_numrecs;
    __be64          bb_leftsib;
    __be64          bb_rightsib;
} xfs_btree_lblock_t;
```

- For intermediate nodes, the data following `xfs_bmbt_block_t` is the same as the root node: array of `xfs_bmbt_key_t` value followed by an array of `xfs_bmbt_ptr_t` values that starts halfway through the block (offset 0x808 for a 4096 byte filesystem block).
- For leaves, an array of `xfs_bmbt_rec_t` extents follow the `xfs_bmbt_block_t` header.
- Nodes and leaves use the same value for `bb_magic`:

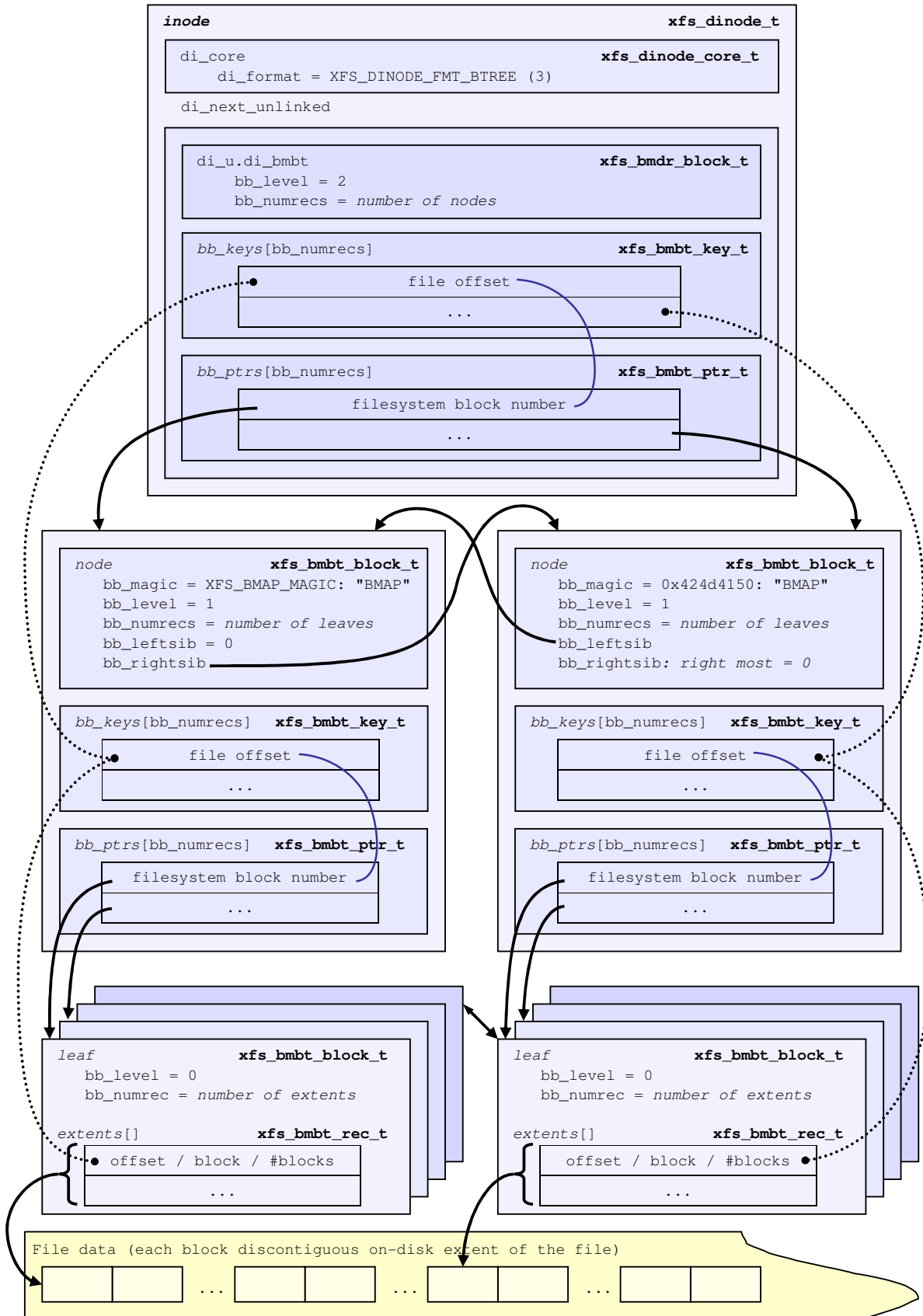
```
#define XFS_BMAP_MAGIC          0x424d4150 /* 'BMAP' */
```

- The `bb_level` value determines if the node is an intermediate node or a leaf. Leaves have a `bb_level` of zero, nodes are one or greater.
- Intermediate nodes, like leaves, can contain up to 254 pointers to leaf blocks for a standard 4KB filesystem block size as both the keys and pointers are 64 bits in size.

The following diagram illustrates a single level extent B+tree:



The following diagram illustrates a two level extent B+tree:



xfi_db Example:

TODO:

Directories

- Only v2 directories covered here. v1 directories are obsolete.
- The size of a "directory block" is defined by the [superblock's](#) `sb_dirblklog` value. The size in bytes = `sb_blocksize * 2sb_dirblklog`. For example, if `sb_blocksize = 4096`, `sb_dirblklog = 2`, the directory block size is 16384 bytes. Directory blocks are always allocated in multiples based on `sb_dirblklog`. Directory blocks cannot be more than 65536 bytes in size.

Note: the term "block" in this section will refer to directory blocks, not filesystem blocks unless otherwise specified.

- All directory entries contain the following "data":
 - Entry's name (counted string consisting of a single byte `namelen` followed by `name` consisting of an array of 8-bit chars without a NULL terminator).
 - Entry's [absolute inode number](#), which are always 64 bits (8 bytes) in size except a special case for shortform directories.
 - An `offset` or `tag` used for iterative `readdir` calls.
- All non-shortform directories also contain two additional structures: "leaves" and "freespace indexes".
 - Leaves contain the sorted hashed name value (`xfs_da_hashname()` in `xfs_da_btree.c`) and associated "address" which points to the effective offset into the directory's data structures. Leaves are used to optimise lookup operations.
 - Freespace indexes contain free space/empty entry tracking for quickly finding an appropriately sized location for new entries. They maintain the largest free space for each "data" block.
- A few common types are used for the directory structures:

```
typedef __uint16_t      xfs_dir2_data_off_t;
typedef __uint32_t      xfs_dir2_dataptr_t;
```

Shortform Directories

- Directory entries are stored within the inode.
- Only data stored is the name, inode # and offset, no "leaf" or "freespace index" information is required as an inode can only store a few entries.
- "." is not stored (as it's in the inode itself), and ".." is a dedicated `parent` field in the header.
- The number of directories that can be stored in an inode depends on the [inode](#) size, the number of entries, the length of the entry names and extended attribute data.
- Once the number of entries exceed the space available in the inode, the format is converted to a [Block Directory](#)".
- Shortform directory data is packed as tightly as possible on the disk with the remaining space zeroed:

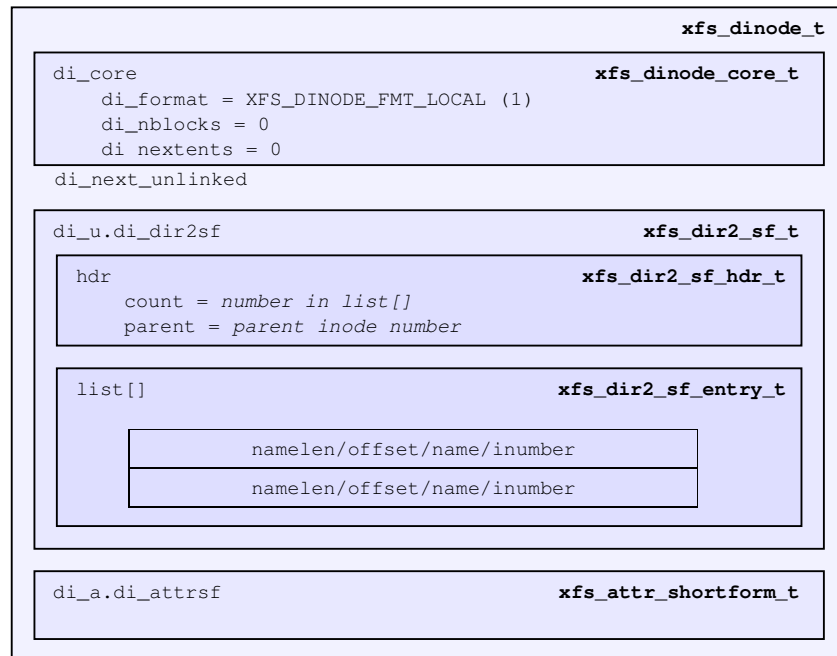
```

typedef struct xfs_dir2_sf {
    xfs_dir2_sf_hdr_t      hdr;
    xfs_dir2_sf_entry_t    list[1];
} xfs_dir2_sf_t;

typedef struct xfs_dir2_sf_hdr {
    __uint8_t              count;
    __uint8_t              i8count;
    xfs_dir2_inou_t        parent;
} xfs_dir2_sf_hdr_t;

typedef struct xfs_dir2_sf_entry {
    __uint8_t              namelen;
    xfs_dir2_sf_off_t      offset;
    __uint8_t              name[1];
    xfs_dir2_inou_t        inumber;
} xfs_dir2_sf_entry_t;

```



- Inode numbers are stored using 4 or 8 bytes depending on whether all the inode numbers for the directory fit in 4 bytes (32 bits) or not. If all inode numbers fit in 4 bytes, the header's `count` value specifies the number of entries in the directory and `i8count` will be zero. If any inode number exceeds 4 bytes, all inode numbers will be 8 bytes in size and the header's `i8count` value specifies the number of entries and `count` will be zero. The following union covers the shortform inode number structure:

```
typedef struct { __uint8_t i[8]; } xfs_dir2_ino8_t;
typedef struct { __uint8_t i[4]; } xfs_dir2_ino4_t;

typedef union {
    xfs_dir2_ino8_t    i8;
    xfs_dir2_ino4_t    i4;
} xfs_dir2_inou_t;
```

xfs_db Example:

A directory is created with 4 files, all inode numbers fitting within 4 bytes:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 1 (local)
core.nlinkv1 = 2
...
core.size = 94
core.nblocks = 0
core.extsize = 0
core.nextents = 0
...
u.sfdir2.hdr.count = 4
u.sfdir2.hdr.i8count = 0
u.sfdir2.hdr.parent.i4 = 128                /* parent = root inode */
u.sfdir2.list[0].namelen = 15
u.sfdir2.list[0].offset = 0x30
u.sfdir2.list[0].name = "frame000000.tst"
u.sfdir2.list[0].inumber.i4 = 25165953
u.sfdir2.list[1].namelen = 15
u.sfdir2.list[1].offset = 0x50
u.sfdir2.list[1].name = "frame000001.tst"
u.sfdir2.list[1].inumber.i4 = 25165954
u.sfdir2.list[2].namelen = 15
u.sfdir2.list[2].offset = 0x70
u.sfdir2.list[2].name = "frame000002.tst"
u.sfdir2.list[2].inumber.i4 = 25165955
u.sfdir2.list[3].namelen = 15
u.sfdir2.list[3].offset = 0x90
u.sfdir2.list[3].name = "frame000003.tst"
u.sfdir2.list[3].inumber.i4 = 25165956
```

The raw data on disk with the first entry highlighted. The six byte header precedes the first entry:

```
xfs_db> type text
xfs_db> p
00: 49 4e 41 ed 01 01 00 02 00 00 00 00 00 00 00 00 INA.....
10: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 02 .....
20: 44 ad 3a 83 1d a9 4a d0 44 ad 3a ab 0b c7 a7 d0 D.....J.D.....
30: 44 ad 3a ab 0b c7 a7 d0 00 00 00 00 00 00 00 00 5e D.....
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
50: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 04 00 00 00 00 80 0f 00 30 66 72 61 .....0fra
70: 6d 65 30 30 30 30 30 30 2e 74 73 74 01 80 00 81 me000000.tst....
80: 0f 00 50 66 72 61 6d 65 30 30 30 30 30 31 2e 74 ..Pframe000001.t
```



```

90: 73 74 01 80 00 82 0f 00 70 66 72 61 6d 65 30 30 st.....pframe00
a0: 30 30 30 32 2e 74 73 74 01 80 00 83 0f 00 90 66 0002.tst.....f
b0: 72 61 6d 65 30 30 30 30 33 2e 74 73 74 01 80 rame000003.tst..
c0: 00 84 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
...

```

Next, an entry is deleted (frame000001.tst), and any entries after the deleted entry are moved or compacted to "cover" the hole:

```

xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 1 (local)
core.nlinkv1 = 2
...
core.size = 72
core.nblocks = 0
core.extsize = 0
core.nextents = 0
...
u.sfdire2.hdr.count = 3
u.sfdire2.hdr.i8count = 0
u.sfdire2.hdr.parent.i4 = 128
u.sfdire2.list[0].namelen = 15
u.sfdire2.list[0].offset = 0x30
u.sfdire2.list[0].name = "frame000000.tst"
u.sfdire2.list[0].inumber.i4 = 25165953
u.sfdire2.list[1].namelen = 15
u.sfdire2.list[1].offset = 0x70
u.sfdire2.list[1].name = "frame000002.tst"
u.sfdire2.list[1].inumber.i4 = 25165955
u.sfdire2.list[2].namelen = 15
u.sfdire2.list[2].offset = 0x90
u.sfdire2.list[2].name = "frame000003.tst"
u.sfdire2.list[2].inumber.i4 = 25165956

```

Raw disk data, the space beyond the shortform entries is invalid and could be non-zero:

```

xfs_db> type text
xfs_db> p
00: 49 4e 41 ed 01 01 00 02 00 00 00 00 00 00 00 00 INA.....
10: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 03 .....
20: 44 b2 45 a2 09 fd e4 50 44 b2 45 a3 12 ee b5 d0 D.E....PD.E....
30: 44 b2 45 a3 12 ee b5 d0 00 00 00 00 00 00 00 00 48 D.E.....H
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
50: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 03 00 00 00 00 80 0f 00 30 66 72 61 .....0fra
70: 6d 65 30 30 30 30 30 30 2e 74 73 74 01 80 00 81 me000000.tst....
80: 0f 00 70 66 72 61 6d 65 30 30 30 30 30 32 2e 74 ..pframe000002.t
90: 73 74 01 80 00 83 0f 00 90 66 72 61 6d 65 30 30 st.....frame00
a0: 30 30 30 33 2e 74 73 74 01 80 00 84 0f 00 90 66 0003.tst.....f
b0: 72 61 6d 65 30 30 30 30 33 2e 74 73 74 01 80 rame000003.tst..
c0: 00 84 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
...

```

TODO: 8-byte inode number example

Block Directories

When the shortform directory space exceeds the space in an inode, the directory data is moved into a new single directory block outside the inode. The inode's format is changed from "local" to "extent". Following is a list of points about block directories.

- All directory data is stored within the one directory block, including "." and ".." entries which are mandatory.
- The block also contains "leaf" and "freespace index" information.
- The location of the block is defined by the inode's [in-core extent list](#): the `di_u.u_bmx[0]` value. The file offset in the extent must always be zero and the `length = (directory block size / filesystem block size)`. The block number points to the filesystem block containing the directory data.
- Block directory data is stored in the following structures:

```
#define XFS_DIR2_DATA_FD_COUNT 3

typedef struct xfs_dir2_block {
    xfs_dir2_data_hdr_t      hdr;
    xfs_dir2_data_union_t    u[1];
    xfs_dir2_leaf_entry_t    leaf[1];
    xfs_dir2_block_tail_t    tail;
} xfs_dir2_block_t;

typedef struct xfs_dir2_data_hdr {
    __uint32_t                magic;
    xfs_dir2_data_free_t      bestfree[XFS_DIR2_DATA_FD_COUNT];
} xfs_dir2_data_hdr_t;

typedef struct xfs_dir2_data_free {
    xfs_dir2_data_off_t       offset;
    xfs_dir2_data_off_t       length;
} xfs_dir2_data_free_t;

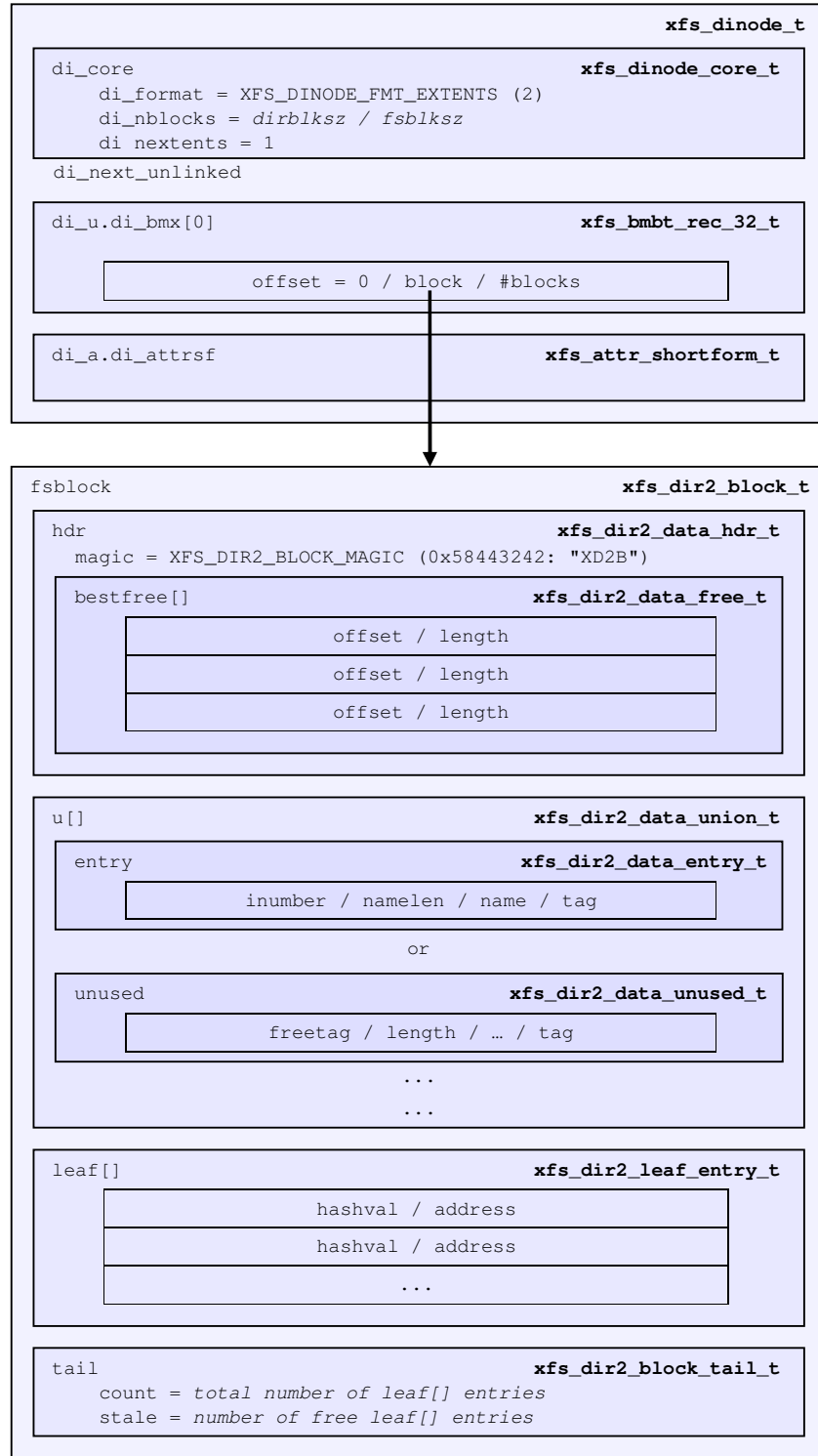
typedef union {
    xfs_dir2_data_entry_t     entry;
    xfs_dir2_data_unused_t    unused;
} xfs_dir2_data_union_t;

typedef struct xfs_dir2_data_entry {
    xfs_ino_t                  inumber;
    __uint8_t                  namelen;
    __uint8_t                  name[1];
    xfs_dir2_data_off_t        tag;
} xfs_dir2_data_entry_t;

typedef struct xfs_dir2_data_unused {
    __uint16_t                 freetag; /* 0xffff */
    xfs_dir2_data_off_t        length;
    xfs_dir2_data_off_t        tag;
} xfs_dir2_data_unused_t;

typedef struct xfs_dir2_leaf_entry {
    xfs_dahash_t                hashval;
    xfs_dir2_dataptr_t          address;
} xfs_dir2_leaf_entry_t;

typedef struct xfs_dir2_block_tail {
    __uint32_t                  count;
    __uint32_t                  stale;
} xfs_dir2_block_tail_t;
```



- The tag in the `xfs_dir2_data_entry_t` structure stores its offset from the start of the block.
- Start of a free space region is marked with the `xfs_dir2_data_unused_t` structure where the `freetag` is `0xffff`. The `freetag` and `length` overwrites the `inumber` for an entry. The tag is located at `length - sizeof(tag)` from the start of the unused entry on-disk.

- The `bestfree` array in the header points to as many as three of the largest spaces of free space within the block for storing new entries sorted by largest to third largest. If there are less than 3 empty regions, the remaining `bestfree` elements are zeroed. The `offset` specifies the offset from the start of the block in bytes, and the `length` specifies the size of the free space in bytes. The location each points to must contain the above `xfs_dir2_data_unused_t` structure. As a block cannot exceed 64KB in size, each is a 16-bit value. `bestfree` is used to optimise the time required to locate space to create an entry. It saves scanning through the block to find a location suitable for every entry created.
- The `tail` structure specifies the number of elements in the `leaf` array and the number of `stale` entries in the array. The `tail` is always located at the end of the block. The `leaf` data immediately precedes the `tail` structure.
- The `leaf` array, which grows from the end of the block just before the `tail` structure, contains an array of hash/address pairs for quickly looking up a name by a hash value. Hash values are covered by the introduction to directories. The `address` on-disk is the offset into the block divided by 8 (`XFS_DIR2_DATA_ALIGN`). Hash/address pairs are stored on disk to optimise lookup speed for large directories. If they were not stored, the hashes have to be calculated for all entries each time a lookup occurs in a directory.

xfs_db Example:

A directory is created with 8 entries, directory block size = filesystem block size:

```
xfs_db> sb 0
xfs_db> p
magicnum = 0x58465342
blocksize = 4096
...
dirblklog = 0
...
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 2 (extents)
core.nlinkv1 = 2
...
core.size = 4096
core.nblocks = 1
core.extsize = 0
core.nextents = 1
...
u.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,2097164,1,0]
```

Go to the "startblock" and show the raw disk data:

```
xfs_db> dblock 0
xfs_db> type text
xfs_db> p
000:  58 44 32 42 01 30 0e 78 00 00 00 00 00 00 00 00  XD2B.0.x.....
010:  00 00 00 00 02 00 00 80 01 2e 00 00 00 00 00 10  .....
020:  00 00 00 00 00 00 00 80 02 2e 2e 00 00 00 00 20  .....
030:  00 00 00 00 02 00 00 81 0f 66 72 61 6d 65 30 30  .....frame00
040:  30 30 30 30 2e 74 73 74 80 8e 59 00 00 00 00 30  0000.tst..Y...0
050:  00 00 00 00 02 00 00 82 0f 66 72 61 6d 65 30 30  .....frame00
060:  30 30 30 31 2e 74 73 74 d0 ca 5c 00 00 00 00 50  0001.tst.....P
070:  00 00 00 00 02 00 00 83 0f 66 72 61 6d 65 30 30  .....frame00
080:  30 30 30 32 2e 74 73 74 00 00 00 00 00 00 00 70  0002.tst.....p
090:  00 00 00 00 02 00 00 84 0f 66 72 61 6d 65 30 30  .....frame00
0a0:  30 30 30 33 2e 74 73 74 00 00 00 00 00 00 00 90  0003.tst.....
0b0:  00 00 00 00 02 00 00 85 0f 66 72 61 6d 65 30 30  .....frame00
```

```

0c0: 30 30 30 34 2e 74 73 74 00 00 00 00 00 00 00 b0 0004.tst.....
0d0: 00 00 00 00 02 00 00 86 0f 66 72 61 6d 65 30 30 .....frame00
0e0: 30 30 30 35 2e 74 73 74 00 00 00 00 00 00 00 d0 0005.tst.....
0f0: 00 00 00 00 02 00 00 87 0f 66 72 61 6d 65 30 30 .....frame00
100: 30 30 30 36 2e 74 73 74 00 00 00 00 00 00 00 f0 0006.tst.....
110: 00 00 00 00 02 00 00 88 0f 66 72 61 6d 65 30 30 .....frame00
120: 30 30 30 37 2e 74 73 74 00 00 00 00 00 00 01 10 0007.tst.....
130: ff ff 0e 78 00 00 00 00 00 00 00 00 00 00 00 00 ...x.....

```

The "leaf" and "tail" structures are stored at the end of the block, so as the directory grows, the middle is filled in:

```

fa0: 00 00 00 00 00 00 01 30 00 00 00 2e 00 00 00 02 .....0.....
fb0: 00 00 17 2e 00 00 00 04 83 a0 40 b4 00 00 00 0e .....
fc0: 93 a0 40 b4 00 00 00 12 a3 a0 40 b4 00 00 00 06 .....
fd0: b3 a0 40 b4 00 00 00 0a c3 a0 40 b4 00 00 00 1e .....
fe0: d3 a0 40 b4 00 00 00 22 e3 a0 40 b4 00 00 00 16 .....
ff0: f3 a0 40 b4 00 00 00 1a 00 00 00 0a 00 00 00 00 .....

```

In a readable format:

```

xfs_db> type dir2
xfs_db> p
bhdr.magic = 0x58443242
bhdr.bestfree[0].offset = 0x130
bhdr.bestfree[0].length = 0xe78
bhdr.bestfree[1].offset = 0
bhdr.bestfree[1].length = 0
bhdr.bestfree[2].offset = 0
bhdr.bestfree[2].length = 0
bu[0].inumber = 33554560
bu[0].namelen = 1
bu[0].name = "."
bu[0].tag = 0x10
bu[1].inumber = 128
bu[1].namelen = 2
bu[1].name = ".."
bu[1].tag = 0x20
bu[2].inumber = 33554561
bu[2].namelen = 15
bu[2].name = "frame000000.tst"
bu[2].tag = 0x30
bu[3].inumber = 33554562
bu[3].namelen = 15
bu[3].name = "frame000001.tst"
bu[3].tag = 0x50
...
bu[8].inumber = 33554567
bu[8].namelen = 15
bu[8].name = "frame000006.tst"
bu[8].tag = 0xf0
bu[9].inumber = 33554568
bu[9].namelen = 15
bu[9].name = "frame000007.tst"
bu[9].tag = 0x110
bu[10].freetag = 0xffff
bu[10].length = 0xe78
bu[10].tag = 0x130
bleaf[0].hashval = 0x2e
bleaf[0].address = 0x2
bleaf[1].hashval = 0x172e
bleaf[1].address = 0x4
bleaf[2].hashval = 0x83a040b4
bleaf[2].address = 0xe
...

```

```

bleaf[8].hashval = 0xe3a040b4
bleaf[8].address = 0x16
bleaf[9].hashval = 0xf3a040b4
bleaf[9].address = 0x1a
btail.count = 10
btail.stale = 0

```

Note that with block directories, all `xfs_db` fields are preceded with "b".

For a simple lookup example, the hash of `frame000000.tst` is `0xb3a040b4`. Looking up that value, we get an address of `0x6`. Multiply that by 8, it becomes offset `0x30` and the inode at that point is `33554561`.

When we remove an entry from the middle (`frame000004.tst`), we can see how the freespace details are adjusted:

```

bhdr.magic = 0x58443242
bhdr.bestfree[0].offset = 0x130
bhdr.bestfree[0].length = 0xe78
bhdr.bestfree[1].offset = 0xb0
bhdr.bestfree[1].length = 0x20
bhdr.bestfree[2].offset = 0
bhdr.bestfree[2].length = 0
...
bu[5].inumber = 33554564
bu[5].namelen = 15
bu[5].name = "frame000003.tst"
bu[5].tag = 0x90
bu[6].freetag = 0xffff
bu[6].length = 0x20
bu[6].tag = 0xb0
bu[7].inumber = 33554566
bu[7].namelen = 15
bu[7].name = "frame000005.tst"
bu[7].tag = 0xd0
...
bleaf[7].hashval = 0xd3a040b4
bleaf[7].address = 0x22
bleaf[8].hashval = 0xe3a040b4
bleaf[8].address = 0
bleaf[9].hashval = 0xf3a040b4
bleaf[9].address = 0x1a
btail.count = 10
btail.stale = 1

```

A new "bestfree" value is added for the entry, the start of the entry is marked as unused with `0xffff` (which overwrites the inode number for an actual entry), and the length of the space. The tag remains intact at the `offset+length - sizeof(tag)`. The address for the hash is also cleared. The affected areas are highlighted below:

```

090: 00 00 00 00 02 00 00 84 0f 66 72 61 6d 65 30 30 .....frame00
0a0: 30 30 30 33 2e 74 73 74 00 00 00 00 00 00 00 90 0003.tst.....
0b0: ff ff 00 20 02 00 00 85 0f 66 72 61 6d 65 30 30 .....frame00
0c0: 30 30 30 34 2e 74 73 74 00 00 00 00 00 00 00 b0 0004.tst.....
0d0: 00 00 00 00 02 00 00 86 0f 66 72 61 6d 65 30 30 .....frame00
0e0: 30 30 30 35 2e 74 73 74 00 00 00 00 00 00 00 d0 0005.tst.....
...
fb0: 00 00 17 2e 00 00 00 04 83 a0 40 b4 00 00 00 0e .....
fc0: 93 a0 40 b4 00 00 00 12 a3 a0 40 b4 00 00 00 06 .....
fd0: b3 a0 40 b4 00 00 00 0a c3 a0 40 b4 00 00 00 1e .....
fe0: d3 a0 40 b4 00 00 00 22 e3 a0 40 b4 00 00 00 00 .....
ff0: f3 a0 40 b4 00 00 00 1a 00 00 00 0a 00 00 00 01 .....

```

Leaf Directories

Once a [Block Directory](#) has filled the block, the directory data is changed into a new format. It still uses [extents](#) and the same basic structures, but the "data" and "leaf" are split up into their own extents. The "leaf" information only occupies one extent. As "leaf" information is more compact than "data" information, more than one "data" extent is common.

- Block to Leaf conversions retain the existing block for the data entries and allocate a new block for the leaf and freespace index information.
- As with all directories, data blocks must start at logical offset zero.
- The "leaf" block has a special offset defined by `XFS_DIR2_LEAF_OFFSET`. Currently, this is 32GB and in the extent view, a block offset of 32GB/sb_blocksize. On a 4KB block filesystem, this is 0x800000 (8388608 decimal).
- The "data" extents have a new header (no "leaf" data):

```
typedef struct xfs_dir2_data {
    xfs_dir2_data_hdr_t      hdr;
    xfs_dir2_data_union_t    u[1];
} xfs_dir2_data_t;
```

- The "leaf" extent uses the following structures:

```
typedef struct xfs_dir2_leaf {
    xfs_dir2_leaf_hdr_t      hdr;
    xfs_dir2_leaf_entry_t    ents[1];

    xfs_dir2_data_off_t      bests[1];
    xfs_dir2_leaf_tail_t     tail;
} xfs_dir2_leaf_t;
```

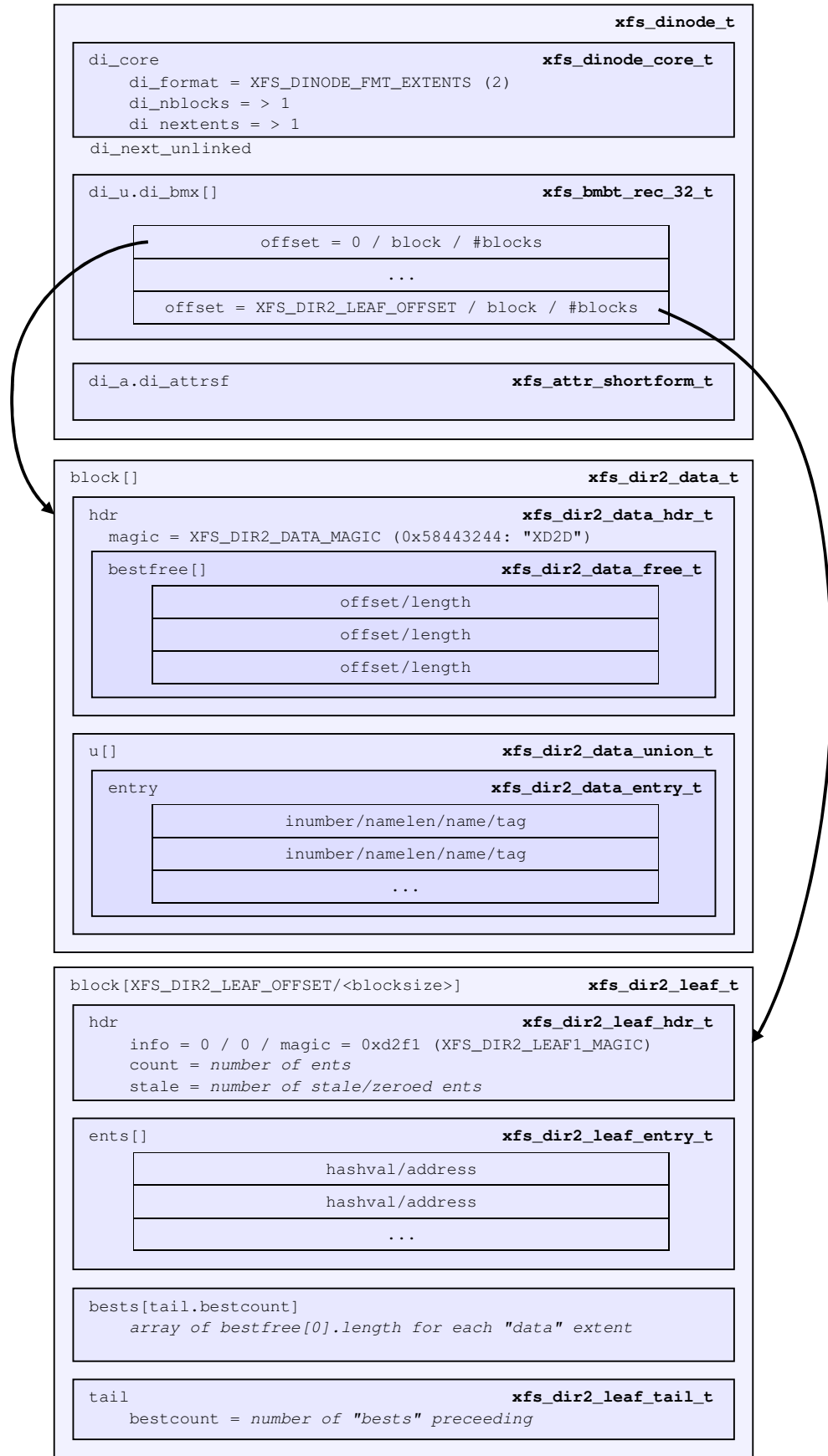
```
typedef struct xfs_dir2_leaf_hdr {
    xfs_da_blkinfo_t         info;
    __uint16_t               count;
    __uint16_t               stale;
} xfs_dir2_leaf_hdr_t;
```

```
typedef struct xfs_dir2_leaf_tail {
    __uint32_t               bestcount;
} xfs_dir2_leaf_tail_t;
```

- The leaves use the `xfs_da_blkinfo_t` filesystem block header. This header is used for directory and [extended attribute](#) leaves and B+tree nodes:

```
typedef struct xfs_da_blkinfo {
    __be32                   forw;
    __be32                   back;
    __be16                   magic;
    __be16                   pad;
} xfs_da_blkinfo_t;
```

- The size of the `ents` array is specified by `hdr.count`.
- The size of the `bests` array is specified by the `tail.bestcount` which is also the number of "data" blocks for the directory. The `bests` array maintains each data block's `bestfree[0].length` value.



xfs_db Example:

For this example, a directory was created with 256 entries (frame000000.tst to frame000255.tst) and then deleted some files (frame00005*, frame00018* and frame000240.tst) to show free list characteristics.

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 2 (extents)
core.nlinkv1 = 2
...
core.size = 12288
core.nblocks = 4
core.extsize = 0
core.nextents = 3
...
u.bmx[0-2] = [startoff, startblock, blockcount, extentflag]
           0: [0, 4718604, 1, 0]
           1: [1, 4718610, 2, 0]
           2: [8388608, 4718605, 1, 0]
```

As can be seen in this example, three blocks are used for "data" in two extents, and the "leaf" extent has a logical offset of 8388608 blocks (32GB).

Examining the first block:

```
xfs_db> dblock 0
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x670
dhdr.bestfree[0].length = 0x140
dhdr.bestfree[1].offset = 0xff0
dhdr.bestfree[1].length = 0x10
dhdr.bestfree[2].offset = 0
dhdr.bestfree[2].length = 0
du[0].inumber = 75497600
du[0].namelen = 1
du[0].name = "."
du[0].tag = 0x10
du[1].inumber = 128
du[1].namelen = 2
du[1].name = ".."
du[1].tag = 0x20
du[2].inumber = 75497601
du[2].namelen = 15
du[2].name = "frame000000.tst"
du[2].tag = 0x30
du[3].inumber = 75497602
du[3].namelen = 15
du[3].name = "frame000001.tst"
du[3].tag = 0x50
...
du[51].inumber = 75497650
du[51].namelen = 15
du[51].name = "frame000049.tst"
du[51].tag = 0x650
du[52].freetag = 0xffff
du[52].length = 0x140
du[52].tag = 0x670
du[53].inumber = 75497661
du[53].namelen = 15
du[53].name = "frame000060.tst"
```

```

du[53].tag = 0x7b0
...
du[118].inumber = 75497758
du[118].namelen = 15
du[118].name = "frame000125.tst"
du[118].tag = 0xfd0
du[119].freetag = 0xffff
du[119].length = 0x10
du[119].tag = 0xff0

```

Note that the `xfs_db` field output is preceded by a "d" for "data".

The next "data" block:

```

xfs_db> dblock 1
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x6d0
dhdr.bestfree[0].length = 0x140
dhdr.bestfree[1].offset = 0xe50
dhdr.bestfree[1].length = 0x20
dhdr.bestfree[2].offset = 0xff0
dhdr.bestfree[2].length = 0x10
du[0].inumber = 75497759
du[0].namelen = 15
du[0].name = "frame000126.tst"
du[0].tag = 0x10
...
du[53].inumber = 75497844
du[53].namelen = 15
du[53].name = "frame000179.tst"
du[53].tag = 0x6b0
du[54].freetag = 0xffff
du[54].length = 0x140
du[54].tag = 0x6d0
du[55].inumber = 75497855
du[55].namelen = 15
du[55].name = "frame000190.tst"
du[55].tag = 0x810
...
du[104].inumber = 75497904
du[104].namelen = 15
du[104].name = "frame000239.tst"
du[104].tag = 0xe30
du[105].freetag = 0xffff
du[105].length = 0x20
du[105].tag = 0xe50
du[106].inumber = 75497906
du[106].namelen = 15
du[106].name = "frame000241.tst"
du[106].tag = 0xe70
...
du[117].inumber = 75497917
du[117].namelen = 15
du[117].name = "frame000252.tst"
du[117].tag = 0xfd0
du[118].freetag = 0xffff
du[118].length = 0x10
du[118].tag = 0xff0

```

And the last data block:

```

xfs_db> dblock 2
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x70
dhdr.bestfree[0].length = 0xf90

```

```

dhdr.bestfree[1].offset = 0
dhdr.bestfree[1].length = 0
dhdr.bestfree[2].offset = 0
dhdr.bestfree[2].length = 0
du[0].inumber = 75497918
du[0].namelen = 15
du[0].name = "frame000253.tst"
du[0].tag = 0x10
du[1].inumber = 75497919
du[1].namelen = 15
du[1].name = "frame000254.tst"
du[1].tag = 0x30
du[2].inumber = 75497920
du[2].namelen = 15
du[2].name = "frame000255.tst"
du[2].tag = 0x50
du[3].freetag = 0xffff
du[3].length = 0xf90
du[3].tag = 0x70

```

Examining the "leaf" block (with the fields preceded by an "l" for "leaf"):

The directory before deleting some entries:

```

xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 0
lhdr.info.back = 0
lhdr.info.magic = 0xd2f1
lhdr.count = 258
lhdr.stale = 0
lbests[0-2] = 0:0x10 1:0x10 2:0xf90
lents[0].hashval = 0x2e
lents[0].address = 0x2
lents[1].hashval = 0x172e
lents[1].address = 0x4
lents[2].hashval = 0x23a04084
lents[2].address = 0x116
...
lents[257].hashval = 0xf3a048bc
lents[257].address = 0x366
ltail.bestcount = 3

```

Note how the `lbests` array correspond with the `bestfree[0].length` values in the "data" blocks:

```

xfs_db> dblock 0
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0xff0
dhdr.bestfree[0].length = 0x10
...
xfs_db> dblock 1
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0xff0
dhdr.bestfree[0].length = 0x10
...
xfs_db> dblock 2
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x70
dhdr.bestfree[0].length = 0xf90

```

Now after the entries have been deleted:

```
xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 0
lhdr.info.back = 0
lhdr.info.magic = 0xd2f1
lhdr.count = 258
lhdr.stale = 21
lbests[0-2] = 0:0x140 1:0x140 2:0xf90
lents[0].hashval = 0x2e
lents[0].address = 0x2
lents[1].hashval = 0x172e
lents[1].address = 0x4
lents[2].hashval = 0x23a04084
lents[2].address = 0x116
...
```

As can be seen, the `lbests` values have been update to contain each `hdr.bestfree[0].length` values. The leaf's `hdr.stale` value has also been updated to specify the number of stale entries in the array. The stale entries have an address of zero.

TODO: Need an example for where new entries get inserted with several large free spaces.

Node Directories

When the "leaf" information fills a block, the extents undergo another separation. All "freeindex" information moves into its own extent. Like [Leaf Directories](#), the "leaf" block maintained the best free space information for each "data" block. This is not possible with more than one leaf.

- The "data" blocks stay the same as leaf directories.
- The "leaf" blocks eventually change into a B+tree with the generic B+tree header pointing to directory "leaves" as described in Leaf Directories. The top-level blocks are called "nodes". It can exist in a state where there is still a single leaf block before it's split. Interpretation of the node vs. leaf blocks has to be performed by inspecting the magic value in the header. The combined leaf/freeindex blocks has a magic value of `XFS_DIR2_LEAF1_MAGIC (0xd2f1)`, a node directory's leaf/leaves have a magic value of `XFS_DIR2_LEAFN_MAGIC (0xd2ff)` and intermediate nodes have a magic value of `XFS_DA_NODE_MAGIC (0xfebe)`.
- The new "freeindex" block(s) only contains the bests for each data block.
- The freeindex block uses the following structures:

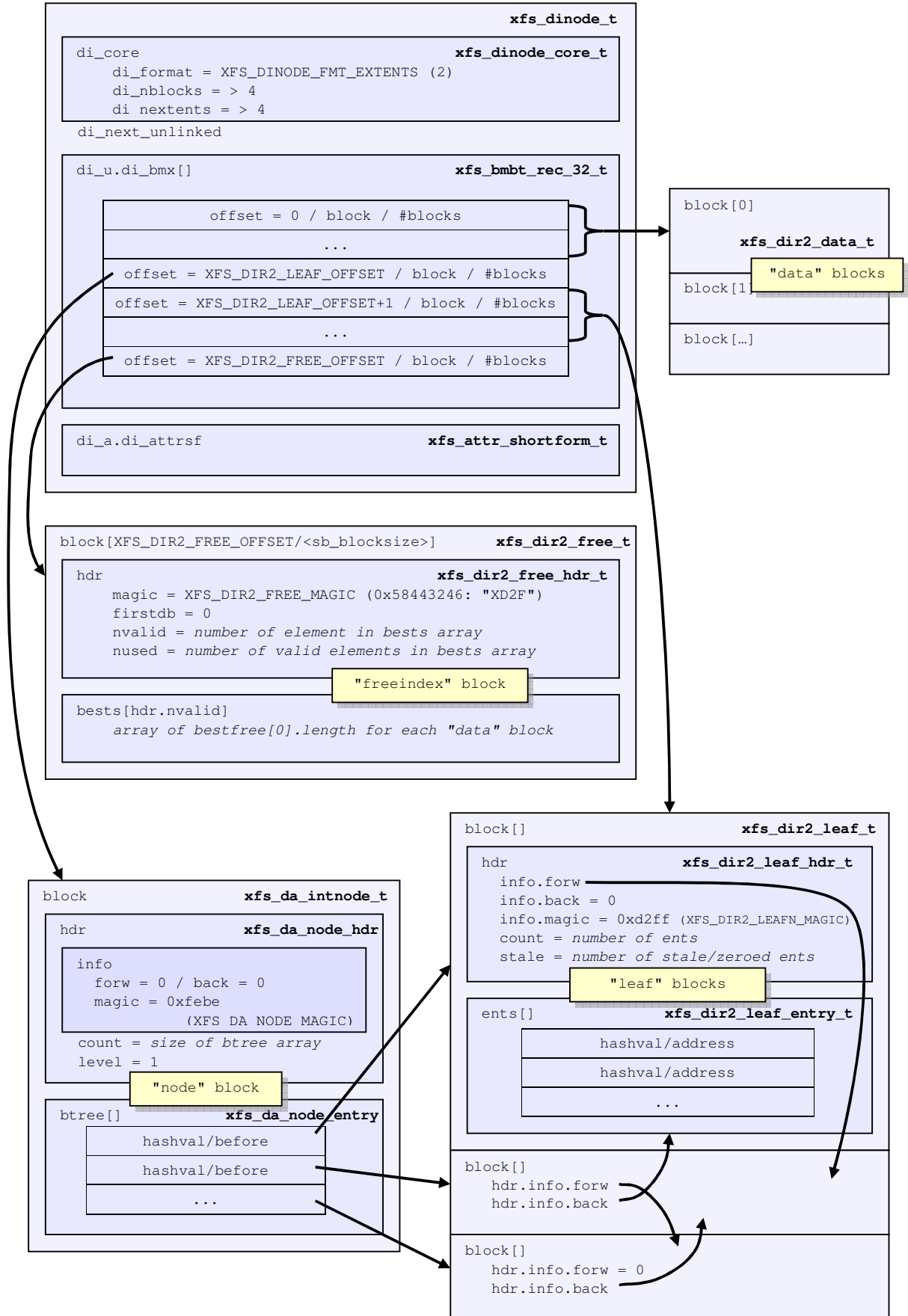
```
typedef struct xfs_dir2_free_hdr {
    __uint32_t      magic;
    __int32_t       firstdb;
    __int32_t       nvalid;
    __int32_t       nused;
} xfs_dir2_free_hdr_t;

typedef struct xfs_dir2_free {
    xfs_dir2_free_hdr_t  hdr;
    xfs_dir2_data_off_t  bests[1];
} xfs_dir2_free_t;
```

- The location of the leaf blocks can be in any order, the only way to determine the appropriate is by the node block hash/before values. Given a hash to lookup, you read the node's `btree` array and first `hashval` in the array that exceeds the given hash and it can then be found in the block pointed to by the `before` value.

```
typedef struct xfs_da_intnode {
    struct xfs_da_node_hdr {
        xfs_da_blkinfo_t  info;
        __uint16_t        count;
        __uint16_t        level;
    } hdr;
    struct xfs_da_node_entry {
        xfs_dahash_t      hashval;
        xfs_dablk_t       before;
    } btree[1];
} xfs_da_intnode_t;
```

- The freeindex's `bests` array starts from the end of the block and grows to the start of the block.
- When an data block becomes unused (ie. all entries in it have been deleted), the block is freed, the data extents contain a hole, and the freeindex's `hdr.nused` value is decremented and the associated `bests[]` entry is set to `0xffff`.
- As the first data block always contains "." and "..", it's invalid for the directory to have a hole at the start.
- The freeindex's `hdr.nvalid` should always be the same as the number of allocated data directory blocks containing name/inode data and will always be less than or equal to `hdr.nused`. `hdr.nused` should be the same as the index of the last data directory block plus one (i.e. when the last data block is freed, `nused` and `nvalid` are decremented).



xfs_db Example:

With the node directory examples, we are using a filesystems with 4KB block size, and a 16KB directory size. The directory has over 2000 entries:

```
xfs_db> sb 0
xfs_db> p
magicnum = 0x58465342
blocksize = 4096
...
dirblklog = 2
...

xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 2 (extents)
...
core.size = 81920
core.nblocks = 36
core.extsize = 0
core.nextents = 8
...
u.bmx[0-7] = [startoff,startblock,blockcount,extentflag] 0:[0,7368,4,0]
1:[4,7408,4,0] 2:[8,7444,4,0] 3:[12,7480,4,0] 4:[16,7520,4,0]
5:[8388608,7396,4,0] 6:[8388612,7524,8,0] 7:[16777216,7516,4,0]
```

As can already be observed, all extents are allocated is multiples of 4 blocks.

Blocks 0 to 19 (16+4-1) are used for the data. Looking at blocks 16-19, it can seen that it's the same as the single-leaf format, except the `length` values are a lot larger to accommodate the increased directory block size:

```
xfs_db> dblock 16
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0xb0
dhdr.bestfree[0].length = 0x3f50
dhdr.bestfree[1].offset = 0
dhdr.bestfree[1].length = 0
dhdr.bestfree[2].offset = 0
dhdr.bestfree[2].length = 0
du[0].inumber = 120224
du[0].namelen = 15
du[0].name = "frame002043.tst"
du[0].tag = 0x10
du[1].inumber = 120225
du[1].namelen = 15
du[1].name = "frame002044.tst"
du[1].tag = 0x30
du[2].inumber = 120226
du[2].namelen = 15
du[2].name = "frame002045.tst"
du[2].tag = 0x50
du[3].inumber = 120227
du[3].namelen = 15
du[3].name = "frame002046.tst"
du[3].tag = 0x70
du[4].inumber = 120228
du[4].namelen = 15
du[4].name = "frame002047.tst"
du[4].tag = 0x90
du[5].freetag = 0xffff
du[5].length = 0x3f50
```

```
du[5].tag = 0
```

Next, the "node" block, the fields are preceded with 'n' for node blocks:

```
xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 0
nhdr.info.back = 0
nhdr.info.magic = 0xfebe
nhdr.count = 2
nhdr.level = 1
nbtree[0-1] = [hashval,before] 0:[0xa3a440ac,8388616] 1:[0xf3a440bc,8388612]
```

The following leaf blocks have been allocated once as XFS knows it needs at two blocks when allocating a B+tree, so the length is 8 fsblocks. For all hashes < 0xa3a440ac, they are located in the directory offset 8388616 and hashes below 0xf3a440bc are in offset 8388612. Hashes above f3a440bc don't exist in this directory.

```
xfs_db> dblock 8388616
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 8388612
lhdr.info.back = 0
lhdr.info.magic = 0xd2ff
lhdr.count = 1023
lhdr.stale = 0
lents[0].hashval = 0x2e
lents[0].address = 0x2
lents[1].hashval = 0x172e
lents[1].address = 0x4
lents[2].hashval = 0x23a04084
lents[2].address = 0x116
...
lents[1021].hashval = 0xa3a440a4
lents[1021].address = 0x1fa2
lents[1022].hashval = 0xa3a440ac
lents[1022].address = 0x1fca

xfs_db> dblock 8388612
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 0
lhdr.info.back = 8388616
lhdr.info.magic = 0xd2ff
lhdr.count = 1027
lhdr.stale = 0
lents[0].hashval = 0xa3a440b4
lents[0].address = 0x1f52
lents[1].hashval = 0xa3a440bc
lents[1].address = 0x1f7a
...
lents[1025].hashval = 0xf3a440b4
lents[1025].address = 0x1f66
lents[1026].hashval = 0xf3a440bc
lents[1026].address = 0x1f8e
```

An example lookup using xfs_db:

```
xfs_db> hash frame001845.tst
0xf3a26094
```

Doing a binary search through the array, we get address 0x1ce6, which is offset 0xe730. Each fsblock is 4KB in size (0x1000), so it will be offset 0x730 into directory offset 14. From the extent map, this will be fsblock 7482:

```
xfs_db> fsblock 7482
```



```
xfs_db> type text
xfs_db> p
...
730:  00 00 00 00 00 01 d4 da 0f 66 72 61 6d 65 30 30  ....frame00
740:  31 38 34 35 2e 74 73 74 00 00 00 00 00 00 27 30  1845.tst.....0
```

Looking at the freeindex information (fields with an 'f' tag):

```
xfs_db> fsblock 7516
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 0
fhdr.nvalid = 5
fhdr.nused = 5
fbests[0-4] = 0:0x10 1:0x10 2:0x10 3:0x10 4:0x3f50
```

Like the [Leaf Directory](#), each of the `fbests` values correspond to each data block's `bestfree[0].length` value.

The raw disk layout, old data is not cleared after the array. The `fbests` array is highlighted:

```
xfs_db> type text
xfs_db> p
000:  58 44 32 46 00 00 00 00 00 00 00 05 00 00 05  XD2F.....
010:  00 10 00 10 00 10 00 10 3f 50 00 00 1f 01 ff ff  ....P.....
```

TODO: Example with a hole in the middle

B+ tree Directories

When the extent map in an inode grows beyond the inode's space, the inode format is changed to a "btree". The inode contains a filesystem block point to the B+tree extent map for the directory's blocks. The B+tree extents contain the extent map for the "data", "node", "leaf" and "freeindex" information as described in [Node Directories](#).

Refer to the previous section on [B+tree Data Extents](#) for more information on XFS B+tree extents.

The following situations and changes can apply over Node Directories, and apply here as inode extents generally cannot contain the number of directory blocks that B+trees can handle:

- The node/leaf trees can be more than one level deep.
- More than one freeindex block may exist, but this will be quite rare. It would require hundreds of thousand files with quite long file names (or millions with shorter names) to get a second freeindex block.

xfstools Example:

A directory has been created with 200,000 entries with each entry being 100 characters long. The filesystem block size and directory block size are 4KB:

```

xfstools> inode 772
xfstools> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 3 (btree)
...
core.size = 22757376
core.nblocks = 6145
core.extsize = 0
core.nextents = 234
core.naextents = 0
core.forkoff = 0
...
u.bmbt.level = 1
u.bmbt.numrecs = 1
u.bmbt.keys[1] = [startoff] 1:[0]
u.bmbt.ptrs[1] = 1:89

xfstools> fsblock 89
xfstools> type bmapbtd
xfstools> p
magic = 0x424d4150
level = 0
numrecs = 234
leftsib = null
rightsib = null
recs[1-234] = [startoff, startblock, blockcount, extentflag]
  1:[0,53,1,0] 2:[1,55,13,0] 3:[14,69,1,0] 4:[15,72,13,0]
  5:[28,86,2,0] 6:[30,90,21,0] 7:[51,112,1,0] 8:[52,114,11,0]
  ...
 125:[5177,902,15,0] 126:[5192,918,6,0] 127:[5198,524786,358,0]
 128:[8388608,54,1,0] 129:[8388609,70,2,0] 130:[8388611,85,1,0]
  ...
 229:[8389164,917,1,0] 230:[8389165,924,19,0] 231:[8389184,944,9,0]
 232:[16777216,68,1,0] 233:[16777217,7340114,1,0] 234:[16777218,5767362,1,0]

```

We have 128 extents and a total of 5555 blocks being used to store name/inode pairs. With only about 2000 values that can be stored in the freeindex block, 3 blocks have been allocated for this information. The `firstdb` field specifies the starting directory block number for each array:

```
xfs_db> dblock 16777216
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 0
fhdr.nvalid = 2040
fhdr.nused = 2040
fbests[0-2039] = ...
```

```
xfs_db> dblock 16777217
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 2040
fhdr.nvalid = 2040
fhdr.nused = 2040
fbests[0-2039] = ...
```

```
xfs_db> dblock 16777218
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 4080
fhdr.nvalid = 1476
fhdr.nused = 1476
fbests[0-1475] = ...
```

Looking at the root node in the node block, it's a pretty deep tree:

```
xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 0
nhdr.info.back = 0
nhdr.info.magic = 0xfebe
nhdr.count = 2
nhdr.level = 2
nbtrees[0-1] = [hashval,before] 0:[0x6bbf6f39,8389121] 1:[0xfbbf7f79,8389120]
```

```
xfs_db> dblock 8389121
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 8389120
nhdr.info.back = 0
nhdr.info.magic = 0xfebe
nhdr.count = 263
nhdr.level = 1
nbtrees[0-262] = ... 262:[0x6bbf6f39,8388928]
```

```
xfs_db> dblock 8389120
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 0
nhdr.info.back = 8389121
nhdr.info.magic = 0xfebe
nhdr.count = 319
nhdr.level = 1
nbtrees[0-318] = [hashval,before] 0:[0x70b14711,8388919] ...
```

The leaves at each the end of a node always point to the end leaves in adjacent nodes. Directory block 8388928 forward pointer is to block 8388919, and vice versa as highlighted in the following example:

```
xfs_db> dblock 8388928
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 8388919
lhdr.info.back = 8388937
lhdr.info.magic = 0xd2ff
...

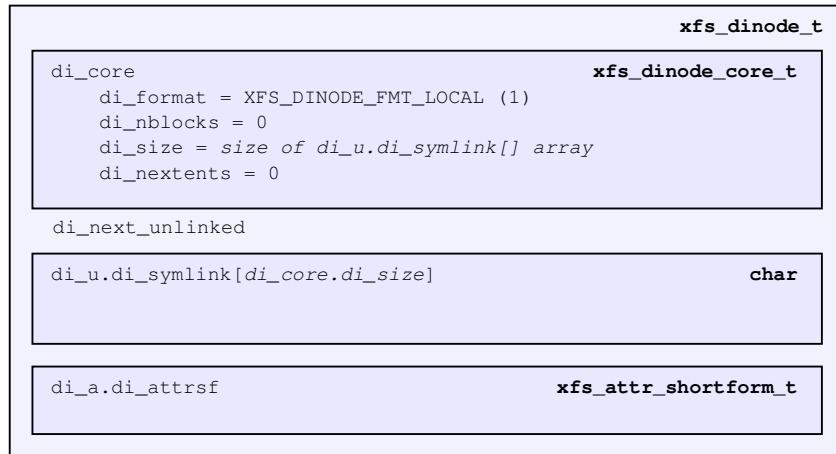
xfs_db> dblock 8388919
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 8388706
lhdr.info.back = 8388928
lhdr.info.magic = 0xd2ff
...
```

Symbolic Links

Symbolic links to a file can be stored in one of two formats: "local" and "extents". The length of the symlink contents is always specified by the inode's `di_size` value.

Shortform Symbolic Links

Symbolic links are stored with the "local" `di_format` if the symbolic link can fit within the inode's data fork. The link data is an array of characters (`di_symlink` array in the data fork union).



xfx_db Example:

A short symbolic link to a file is created:

```

xfx_db> inode <inode#>
xfx_db> p
core.magic = 0x494e
core.mode = 0120777
core.version = 1
core.format = 1 (local)
...
core.size = 12
core.nblocks = 0
core.extsize = 0
core.nextents = 0
...
u.symlink = "small_target"
  
```

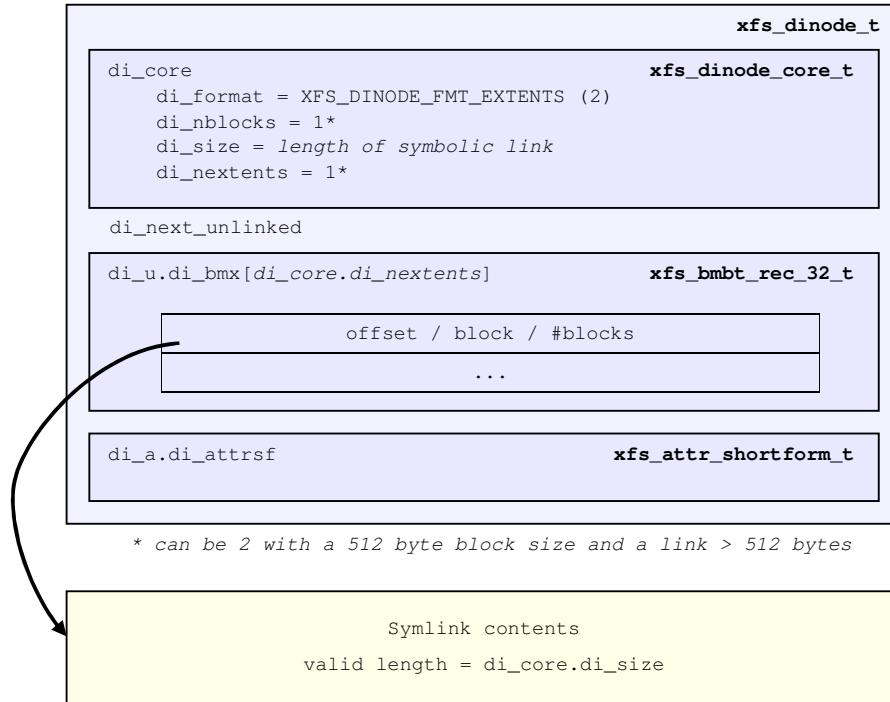
Raw on-disk data with the link contents highlighted:

```

xfx_db> type text
xfx_db> p
00:  49 4e a1 ff 01 01 00 01 00 00 00 00 00 00 00 00  IN.....
10:  00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01  .....
20:  44 bc e1 c7 03 c4 d4 18 44 bc e1 c7 03 c4 d4 18  D.....D.....
30:  44 bc e1 c7 03 c4 d4 18 00 00 00 00 00 00 00 0c  D.....
40:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
50:  00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00  .....
60:  ff ff ff ff 73 6d 61 6c 6c 5f 74 61 72 67 65 74  ...small_target
70:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
...
  
```

Extent Symbolic Links

If the length of the symbolic link exceeds the space available in the inode's data fork, the link is moved to a new filesystem block and the inode's `di_format` is changed to "extents". The location of the block(s) is specified by the data fork's `di_bmx[]` array. In the significant majority of cases, this will be in one filesystem block as a symlink cannot be longer than 1024 characters.



xfs_db Example:

A longer link is created (greater than 156 bytes):

```

xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0120777
core.version = 1
core.format = 2 (extents)
...
core.size = 182
core.nblocks = 1
core.extsize = 0
core.nextents = 1
...
u.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,37530,1,0]

xfs_db> dblock 0
xfs_db> type symlink
xfs_db> p
"symlink contents..."

```

Extended Attributes

Extended attributes implement the ability for a user to attach name:value pairs to inodes within the XFS filesystem. They could be used to store meta-information about the file.

The attribute names can be up to 256 bytes in length, terminated by the first 0 byte. The intent is that they be printable ASCII (or other character set) names for the attribute. The values can be up to 64KB of arbitrary binary data. Some XFS internal attributes (eg. parent pointers) use non-printable names for the attribute.

Access Control Lists (ACLs) and Data Migration Facility (DMF) use extended attributes to store their associated metadata with an inode.

XFS uses two disjoint attribute name spaces associated with every inode. They are the root and user address spaces. The root address space is accessible only to the superuser, and then only by specifying a flag argument to the function call. Other users will not see or be able to modify attributes in the root address space. The user address space is protected by the normal file permissions mechanism, so the owner of the file can decide who is able to see and/or modify the value of attributes on any particular file.

To view extended attributes from the command line, use the `getfattr` command. To set or delete extended attributes, use the `setfattr` command. ACLs control should use the `getfacl` and `setfacl` commands.

XFS attributes supports three namespaces: "user", "trusted" (or "root" using IRIX terminology) and "secure".

The location of the attribute fork in the inode's literal area is specified by the `di_forkoff` value in the inode's core. If this value is zero, the inode does not contain any extended attributes. Non-zero, the byte offset into the literal area = `di_forkoff * 8`, which also determines the 2048 byte maximum size for an inode. Attributes must be allocated on a 64-bit boundary on the disk except shortform attributes (they are tightly packed). To determine the offset into the inode itself, add 100 (0x64) to `di_forkoff * 8`.

The following four sections describe each of the on-disk formats.

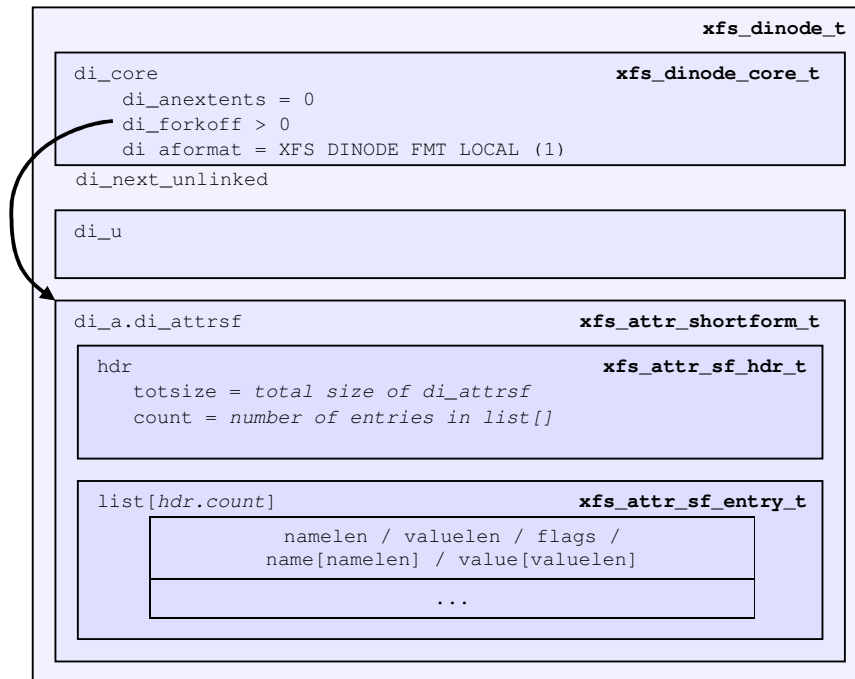
Shortform Attributes

When the all extended attributes can fit within the inode's attribute fork, the inode's `di_aformat` is set to "local" and the attributes are stored in the inode's literal area starting at offset `di_forkoff * 8`.

Shortform attributes use the following structures:

```
typedef struct xfs_attr_shortform {
    struct xfs_attr_sf_hdr {
        __be16      totalsize;
        __u8        count;
    } hdr;
    struct xfs_attr_sf_entry {
        __uint8_t   namelen;
        __uint8_t   valuelen;
        __uint8_t   flags;
        __uint8_t   nameval[1];
    } list[1];
} xfs_attr_shortform_t;

typedef struct xfs_attr_sf_hdr xfs_attr_sf_hdr_t;
typedef struct xfs_attr_sf_entry xfs_attr_sf_entry_t;
```



- `namelen` and `valuelen` specify the size of the two byte arrays containing the name and value pairs. `valuelen` is zero for extended attributes with no value.
- `nameval[]` is a single array where it's size is the sum of `namelen` and `valuelen`. The names and values are not null terminated on-disk. The value immediately follows the name in the array.
- `flags` specifies the namespace for the attribute (0 = "user"):

Flag	Description
XFS_ATTR_ROOT	The attribute's namespace is "trusted".
XFS_ATTR_SECURE	The attribute's namespace is "secure".

xfs_db Example:

A file is created and two attributes are set:

```
# setfattr -n user.empty few_attr
# setfattr -n trusted.trust -v vall few_attr
```

Using `xfs_db`, we dump the inode:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0100644
...
core.naextents = 0
core.forkoff = 15
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 24
a.sfattr.hdr.count = 2
a.sfattr.list[0].namelen = 5
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty"
a.sfattr.list[1].namelen = 5
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust"
a.sfattr.list[1].value = "vall"
```

We can determine the actual inode offset to be 220 (15 x 8 + 100) or `0xdc`.

Examining the raw dump, the second attribute is highlighted:

```
xfs_db> type text
xfs_db> p
00: 49 4e 81 a4 01 02 00 01 00 00 00 00 00 00 00 00  IN.....
10: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 02  .....
20: 44 be 1a be 38 d1 26 98 44 be 1a be 38 d1 26 98  D...8...D...8...
30: 44 be 1a e1 3a 9a ea 18 00 00 00 00 00 00 00 04  D.....
40: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 01  .....
50: 00 00 0f 01 00 00 00 00 00 00 00 00 00 00 00 00  .....
60: ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 12  .....
70: 53 a0 00 01 00 00 00 00 00 00 00 00 00 00 00 00  S.....
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00  .....
e0: 05 00 00 65 6d 70 74 79 05 04 02 74 72 75 73 74  ...empty...trust
f0: 76 61 6c 31 00 00 00 00 00 00 00 00 00 00 00 00  vall.....
```

Adding another attribute with `attr1`, the format is converted to extents and `di_forkoff` remains unchanged (and all those zeros in the dump above remain unused):

```
xfs_db> inode <inode#>
xfs_db> p
...
core.naextents = 1
core.forkoff = 15
core.aformat = 2 (extents)
...
a.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,37534,1,0]
```

Performing the same steps with `attr2`, adding one attribute at a time, you can see `di_forkoff` change as attributes are added:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.naextents = 0
core.forkoff = 15
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 17
a.sfattr.hdr.count = 1
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
```

Attribute added:

```
xfs_db> p
...
core.naextents = 0
core.forkoff = 15
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 31
a.sfattr.hdr.count = 2
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
a.sfattr.list[1].namelen = 7
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust_a"
a.sfattr.list[1].value = "vall"
```

Another attribute is added:

```
xfs_db> p
...
core.naextents = 0
core.forkoff = 13
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 52
a.sfattr.hdr.count = 3
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
a.sfattr.list[1].namelen = 7
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust_a"
a.sfattr.list[1].value = "vall"
a.sfattr.list[2].namelen = 6
a.sfattr.list[2].valuelen = 12
a.sfattr.list[2].root = 0
a.sfattr.list[2].secure = 0
a.sfattr.list[2].name = "second"
a.sfattr.list[2].value = "second_value"
```

One more is added:

```
xfs_db> p
core.naextents = 0
core.forkoff = 10
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 69
a.sfattr.hdr.count = 4
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
a.sfattr.list[1].namelen = 7
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust_a"
a.sfattr.list[1].value = "vall"
a.sfattr.list[2].namelen = 6
a.sfattr.list[2].valuelen = 12
a.sfattr.list[2].root = 0
a.sfattr.list[2].secure = 0
a.sfattr.list[2].name = "second"
a.sfattr.list[2].value = "second_value"
a.sfattr.list[3].namelen = 6
a.sfattr.list[3].valuelen = 8
a.sfattr.list[3].root = 0
a.sfattr.list[3].secure = 1
a.sfattr.list[3].name = "policy"
a.sfattr.list[3].value = "contents"
```

A raw dump is shown to compare with the attr1 dump on a prior page, the header is highlighted:

```
xfs_db> type text
xfs_db> p
00: 49 4e 81 a4 01 02 00 01 00 00 00 00 00 00 00 00 IN.....
10: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 05 .....
20: 44 be 24 cd 0f b0 96 18 44 be 24 cd 0f b0 96 18 D.....D.....
30: 44 be 2d f5 01 62 7a 18 00 00 00 00 00 00 00 04 D....bz.....
40: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01 .....
50: 00 00 0a 01 00 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 01 .....
70: 41 c0 00 01 00 00 00 00 00 00 00 00 00 00 00 00 A.....
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
b0: 00 00 00 00 00 45 04 00 0a 00 00 65 6d 70 74 79 .....E.....empty
c0: 5f 61 74 74 72 07 04 02 74 72 75 73 74 5f 61 76 .attr...trust.av
d0: 61 6c 31 06 0c 00 73 65 63 6f 6e 64 73 65 63 6f all...secondseco
e0: 6e 64 5f 76 61 6c 75 65 06 08 04 70 6f 6c 69 63 nd.value...polic
f0: 79 63 6f 6e 74 65 6e 74 73 64 5f 76 61 6c 75 65 ycontentsd.value
```

It can be clearly seen that attr2 allows many more attributes to be stored in an inode before they are moved to another filesystem block.

Leaf Attributes

When an inode's attribute fork space is used up with [shortform attributes](#) and more are added, the attribute format is migrated to "extents".

Extent based attributes use hash/index pairs to speed up an attribute lookup. The first part of the "leaf" contains an array of fixed size hash/index pairs with the flags stored as well. The remaining part of the leaf block contains the array name/value pairs, where each element varies in length.

Each leaf is based on the `xfs_da_blkinfo_t` block header declared in [Leaf Directories](#). The structure encapsulating all other structures in the `xfs_attr_leafblock_t`.

The structures involved are:

```
typedef struct xfs_attr_leaf_map {
    __be16    base;
    __be16    size;
} xfs_attr_leaf_map_t;

typedef struct xfs_attr_leaf_hdr {
    xfs_da_blkinfo_t    info;
    __be16              count;
    __be16              usedbytes;
    __be16              firstused;
    __u8                holes;
    __u8                pad1;
    xfs_attr_leaf_map_t    freemap[3];
} xfs_attr_leaf_hdr_t;

typedef struct xfs_attr_leaf_entry {
    __be32    hashval;
    __be16    nameidx;
    __u8      flags;
    __u8      pad2;
} xfs_attr_leaf_entry_t;

typedef struct xfs_attr_leaf_name_local {
    __be16    valuelen;
    __u8      namelen;
    __u8      nameval[1];
} xfs_attr_leaf_name_local_t;

typedef struct xfs_attr_leaf_name_remote {
    __be32    valueblk;
    __be32    valuelen;
    __u8      namelen;
    __u8      name[1];
} xfs_attr_leaf_name_remote_t;

typedef struct xfs_attr_leafblock {
    xfs_attr_leaf_hdr_t    hdr;
    xfs_attr_leaf_entry_t  entries[1];
    xfs_attr_leaf_name_local_t    namelist;
    xfs_attr_leaf_name_remote_t    valuelist;
} xfs_attr_leafblock_t;
```

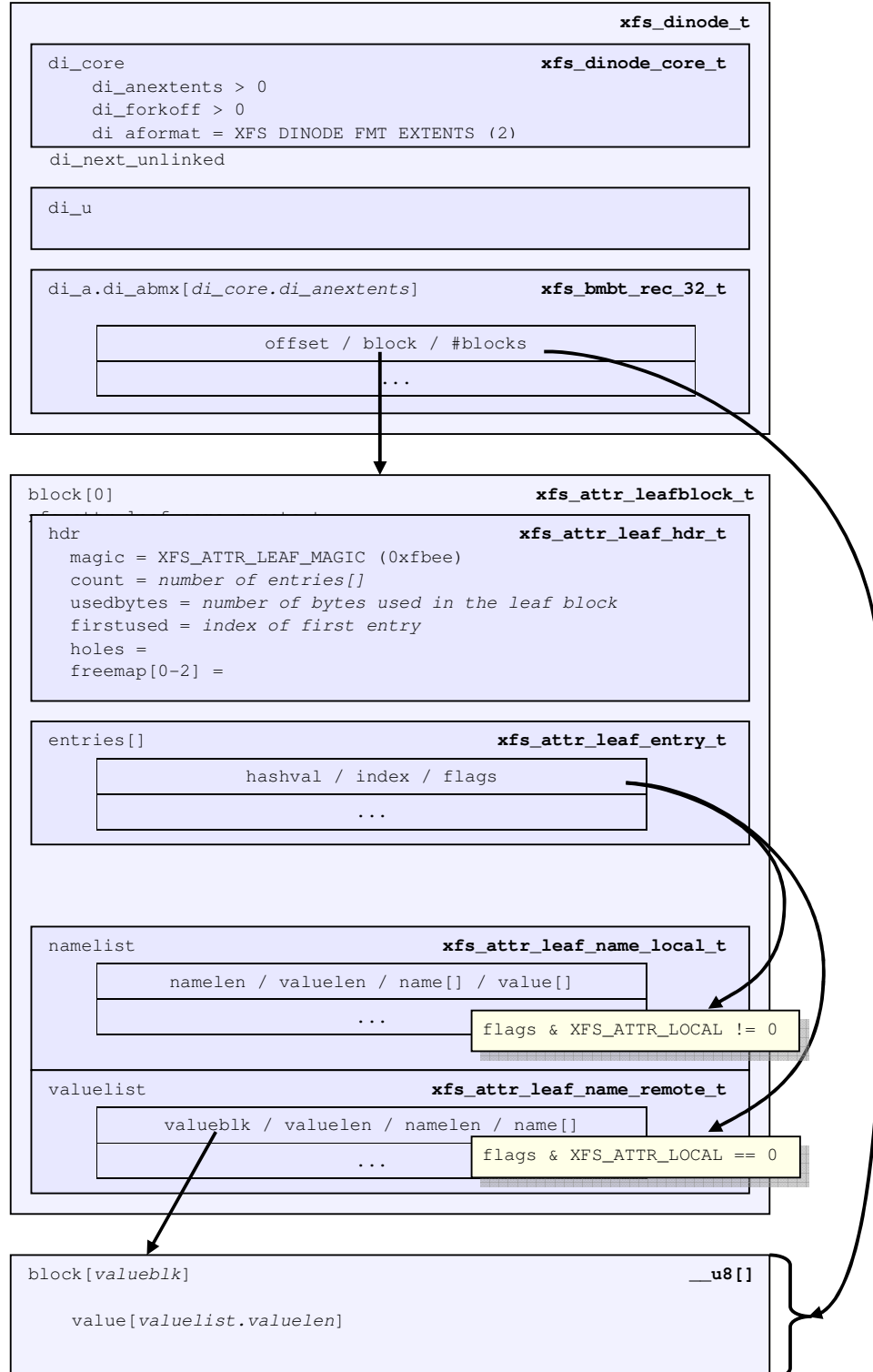
Each leaf header uses the following magic number:

```
#define XFS_ATTR_LEAF_MAGIC    0xfbee
```

The hash/index elements in the `entries[]` array are packed from the top of the block. Name/values grow from the bottom but are not packed. The `freemap` contains run-length-encoded entries for the free bytes after the `entries[]` array, but only the three largest runs are stored (smaller runs are dropped). When the `freemap` doesn't show enough space for an allocation, name/value area is

compacted and allocation is tried again. If there still isn't enough space, then the block is split. The name/value structures (both local and remote versions) must be 32-bit aligned.

For attributes with small values (ie. the value can be stored within the leaf), the `XFS_ATTR_LOCAL` flag is set for the attribute. The entry details are stored using the `xfs_attr_leaf_name_local_t` structure. For large attribute values that cannot be stored within the leaf, separate filesystem blocks are allocated to store the value. They use the `xfs_attr_leaf_name_remote_t` structure.



Both local and remote entries can be interleaved as they are only addressed by the hash/index entries. The flag is stored with the hash/index pairs so the appropriate structure can be used.

Since duplicate hash keys are possible, for each hash that matches during a lookup, the actual name string must be compared.

An “incomplete” bit is also used for attribute flags. It shows that an attribute is in the middle of being created and should not be shown to the user if we crash during the time that the bit is set. The bit is cleared when attribute has finished being setup. This is done because some large attributes cannot be created inside a single transaction.

xfs_db Example:

A single 30KB extended attribute is added to an inode:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.nblocks = 9
core.nextents = 0
core.naextents = 1
core.forkoff = 15
core.aformat = 2 (extents)
...
a.bmx[0] = [startoff, startblock, blockcount, extentflag]
          0:[0, 37535, 9, 0]

xfs_db> ablock 0
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfbee
hdr.count = 1
hdr.usedbytes = 20
hdr.firstused = 4076
hdr.holes = 0
hdr.freemap[0-2] = [base, size] 0:[40, 4036] 1:[0, 0] 2:[0, 0]
entries[0] = [hashval, nameidx, incomplete, root, secure, local]
             0:[0xfcf89d4f, 4076, 0, 0, 0, 0]
nvlst[0].valueblk = 0x1
nvlst[0].valuelen = 30692
nvlst[0].namelen = 8
nvlst[0].name = "big_attr"
```

Attribute blocks 1 to 8 (filesystem blocks 37536 to 37543) contain the raw binary value data for the attribute.

Index 4076 (0xfec) is the offset into the block where the name/value information is. As can be seen by the value, it's at the end of the block:

```
xfs_db> type text
xfs_db> p
000:  00 00 00 00 00 00 00 00 00 fb ee 00 00 00 01 00 14  .....
010:  0f ec 00 00 00 28 0f c4 00 00 00 00 00 00 00 00  .....
020:  fc f8 9d 4f 0f ec 00 00 00 00 00 00 00 00 00 00  ...O.....
030:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
...
fe0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  .....
ff0:  00 00 77 e4 08 62 69 67 5f 61 74 74 72 00 00 00  ..w..big.attr..
```

A 30KB attribute and a couple of small attributes are added to a file:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.nblocks = 10
core.extsize = 0
core.nextents = 1
core.naextents = 2
core.forkoff = 15
core.aformat = 2 (extents)
...
u.bmx[0] = [startoff, startblock, blockcount, extentflag]
          0:[0,81857,1,0]
a.bmx[0-1] = [startoff, startblock, blockcount, extentflag]
             0:[0,81858,1,0]
             1:[1,182398,8,0]

xfs_db> ablock 0
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfbee
hdr.count = 3
hdr.usedbytes = 52
hdr.firstused = 4044
hdr.holes = 0
hdr.freemap[0-2] = [base, size] 0:[56,3988] 1:[0,0] 2:[0,0]
entries[0-2] = [hashval, nameidx, incomplete, root, secure, local]
              0:[0x1e9d3934, 4044, 0, 0, 0, 1]
              1:[0x1e9d3937, 4060, 0, 0, 0, 1]
              2:[0xfcf89d4f, 4076, 0, 0, 0, 0]
nvlst[0].valuelen = 6
nvlst[0].namelen = 5
nvlst[0].name = "attr2"
nvlst[0].value = "value2"
nvlst[1].valuelen = 6
nvlst[1].namelen = 5
nvlst[1].name = "attr1"
nvlst[1].value = "value1"
nvlst[2].valueblk = 0x1
nvlst[2].valuelen = 30692
nvlst[2].namelen = 8
nvlst[2].name = "big_attr"
```

As can be seen in the entries array, the two small attributes have the local flag set and the values are printed.

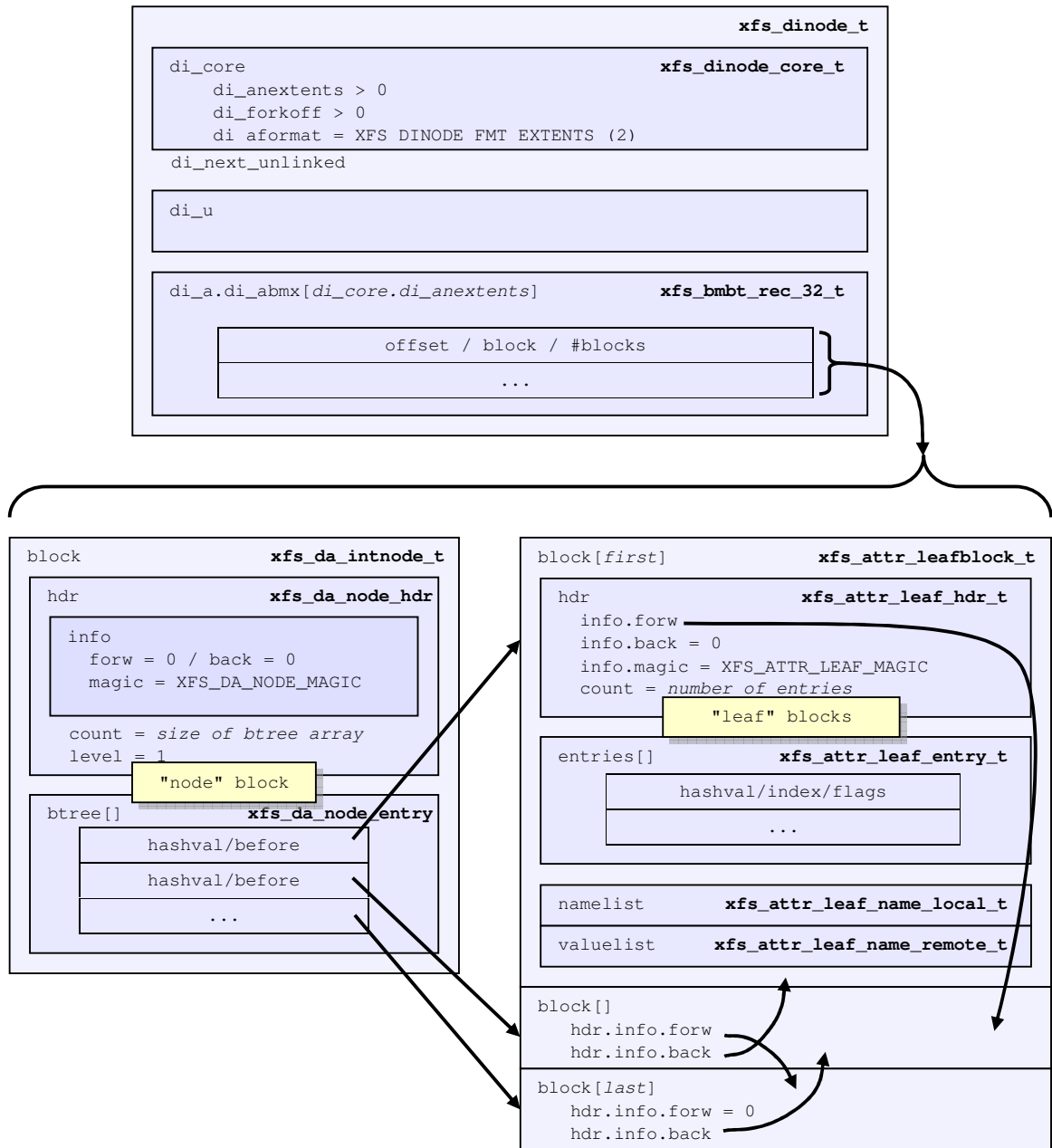
A raw disk dump shows the attributes. The last attribute added is highlighted (offset 4044 or 0xfcc):

```
000: 00 00 00 00 00 00 00 00 fb ee 00 00 00 03 00 34 .....4
010: 0f cc 00 00 00 38 0f 94 00 00 00 00 00 00 00 .....8.....
020: 1e 9d 39 34 0f cc 01 00 1e 9d 39 37 0f dc 01 00 ..94.....97....
030: fc f8 9d 4f 0f ec 00 00 00 00 00 00 00 00 00 ...0.....
040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
...
fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 05 61 .....a
fd0: 74 74 72 32 76 61 6c 75 65 32 00 00 00 06 05 61 ttr2value2.....a
fe0: 74 74 72 31 76 61 6c 75 65 31 00 00 00 00 00 01 ttr1value1.....
ff0: 00 00 77 e4 08 62 69 67 5f 61 74 74 72 00 00 00 ..w..big.attr...
```

Node Attributes

When the number of attributes exceeds the space that can fit in one filesystem block (ie. hash, flag, name and local values), the first attribute block becomes the root of a B+tree where the leaves contain the hash/name/value information that was stored in a single leaf block. The inode's attribute format itself remains extent based. The nodes use the `xfs_da_intnode_t` structure introduced in [Node Directories](#).

The location of the attribute leaf blocks can be in any order, the only way to determine the appropriate is by the node block hash/before values. Given a hash to lookup, you read the node's `btree` array and first `hashval` in the array that exceeds the given hash and it can then be found in the block pointed to by the `before` value.



xfst_db Example:

An inode with 1000 small attributes with the naming "attribute_n" where 'n' is a number:

```

xfst_db> inode <inode#>
xfst_db> p
...
core.nblocks = 15
core.nextents = 0
core.naextents = 1
core.forkoff = 15
core.aformat = 2 (extents)
...
a.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,525144,15,0]

xfst_db> ablock 0
xfst_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfebe
hdr.count = 14
hdr.level = 1
btree[0-13] = [hashval,before]
    0:[0x3435122d,1]
    1:[0x343550a9,14]
    2:[0x343553a6,13]
    3:[0x3436122d,12]
    4:[0x343650a9,8]
    5:[0x343653a6,7]
    6:[0x343691af,6]
    7:[0x3436d0ab,11]
    8:[0x3436d3a7,10]
    9:[0x3437122d,9]
   10:[0x3437922e,3]
   11:[0x3437d22a,5]
   12:[0x3e686c25,4]
   13:[0x3e686fad,2]

```

The hashes are in ascending order in the btree array, and if the hash for the attribute we are looking up is before the entry, we go to the addressed attribute block.

For example, to lookup attribute "attribute_267":

```

xfst_db> hash attribute_267
0x3437d1a8

```

In the root btree node, this falls between 0x3437922e and 0x3437d22a, therefore leaf 11 or attribute block 5 will contain the entry.

```

xfst_db> ablock 5
xfst_db> p
hdr.info.forw = 4
hdr.info.back = 3
hdr.info.magic = 0xfbee
hdr.count = 96
hdr.usedbytes = 2688
hdr.firstused = 1408
hdr.holes = 0
hdr.freemap[0-2] = [base,size] 0:[800,608] 1:[0,0] 2:[0,0]
entries[0-95] = [hashval,nameidx,incomplete,root,secure,local]
    0:[0x3437922f,4068,0,0,0,1]
    1:[0x343792a6,4040,0,0,0,1]
    2:[0x343792a7,4012,0,0,0,1]
    3:[0x343792a8,3984,0,0,0,1]
    ...
   82:[0x3437d1a7,2892,0,0,0,1]
   83:[0x3437d1a8,2864,0,0,0,1]

```

```

84: [0x3437d1a9, 2836, 0, 0, 0, 1]
...
95: [0x3437d22a, 2528, 0, 0, 0, 1]
nvlst[0].valuelen = 10
nvlst[0].namelen = 13
nvlst[0].name = "attribute_310"
nvlst[0].value = "value_310\d"
nvlst[1].valuelen = 10
nvlst[1].namelen = 13
nvlst[1].name = "attribute_309"
nvlst[1].value = "value_309\d"
nvlst[2].valuelen = 10
nvlst[2].namelen = 13
nvlst[2].name = "attribute_308"
nvlst[2].value = "value_308\d"
nvlst[3].valuelen = 10
nvlst[3].namelen = 13
nvlst[3].name = "attribute_307"
nvlst[3].value = "value_307\d"
...
nvlst[82].valuelen = 10
nvlst[82].namelen = 13
nvlst[82].name = "attribute_268"
nvlst[82].value = "value_268\d"
nvlst[83].valuelen = 10
nvlst[83].namelen = 13
nvlst[83].name = "attribute_267"
nvlst[83].value = "value_267\d"
nvlst[84].valuelen = 10
nvlst[84].namelen = 13
nvlst[84].name = "attribute_266"
nvlst[84].value = "value_266\d"
...

```

Each of the hash entries has XFS_ATTR_LOCAL flag set (1), which means the attribute's value follows immediately after the name. Raw disk of the name/value pair at offset 2864 (0xb30), highlighted with "value_267\d" following immediately after the name:

```

b00: 62 75 74 65 5f 32 36 35 76 61 6c 75 65 5f 32 36 bute.265value.26
b10: 35 0a 00 00 00 0a 0d 61 74 74 72 69 62 75 74 65 5.....attribute
b20: 5f 32 36 36 76 61 6c 75 65 5f 32 36 36 0a 00 00 .266value.266...
b30: 00 0a 0d 61 74 74 72 69 62 75 74 65 5f 32 36 37 ...attribute.267
b40: 76 61 6c 75 65 5f 32 36 37 0a 00 00 00 0a 0d 61 value.267.....a
b50: 74 74 72 69 62 75 74 65 5f 32 36 38 76 61 6c 75 ttribute.268valu
b60: 65 5f 32 36 38 0a 00 00 00 0a 0d 61 74 74 72 69 e.268.....attri
b70: 62 75 74 65 5f 32 36 39 76 61 6c 75 65 5f 32 36 bute.269value.26

```

Each entry starts on a 32-bit (4 byte) boundary, therefore the highlighted entry has 2 unused bytes after it.

B+ tree Attributes

When the attribute's extent map in an inode grows beyond the available space, the inode's attribute format is changed to a "btree". The inode contains root node of the extent B+tree which then address the leaves that contains the extent arrays for the attribute data. The attribute data itself in the allocated filesystem blocks use the same layout and structures as described in [Node Attributes](#).

Refer to the previous section on [B+tree Data Extents](#) for more information on XFS B+tree extents.

xfs_db Example:

Added 2000 attributes with 729 byte values to a file:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.nblocks = 640
core.extsize = 0
core.nextents = 1
core.naextents = 274
core.forkoff = 15
core.aformat = 3 (btree)
...
a.bmbt.level = 1
a.bmbt.numrecs = 2
a.bmbt.keys[1-2] = [startoff] 1:[0] 2:[219]
a.bmbt.pters[1-2] = 1:83162 2:109968

xfs_db> fsblock 83162
xfs_db> type bmapbtd
xfs_db> p
magic = 0x424d4150
level = 0
numrecs = 127
leftsib = null
rightsib = 109968
recs[1-127] = [startoff, startblock, blockcount, extentflag]
              1:[0, 81870, 1, 0]
              ...

xfs_db> fsblock 109968
xfs_db> type bmapbtd
xfs_db> p
magic = 0x424d4150
level = 0
numrecs = 147
leftsib = 83162
rightsib = null
recs[1-147] = [startoff, startblock, blockcount, extentflag]
              ...

xfs_db> ablock 0                (which is fsblock 81870)
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfebe
hdr.count = 2
hdr.level = 2
btree[0-1] = [hashval, before] 0:[0x343612a6, 513] 1:[0x3e686fad, 512]
```

The extent B+tree has two leaves that specify the 274 extents used for the attributes. Looking at the first block, it can be seen that the attribute B+tree is two levels deep. The two blocks at offset 513 and 512 (ie. access using the `ablock` command) are intermediate `xfs_da_intnode_t` nodes that index all the attribute leaves.

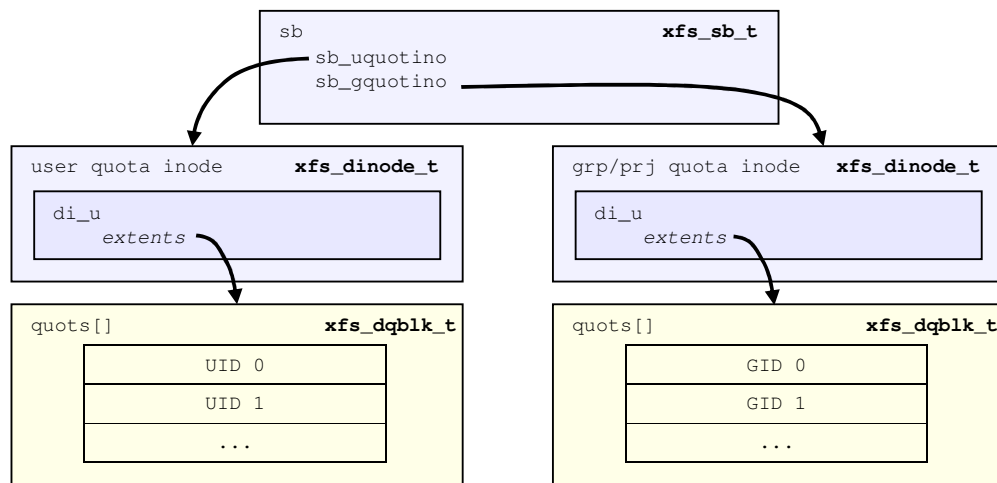
Internal Inodes

XFS allocates several inodes when a filesystem is created. These are internal and not accessible from the standard directory structure. These inodes are only accessible from the [superblock](#).

Quota Inodes

If quotas are used, two inodes are allocated for user and group quota management. If project quotas are used, these replace the group quota management and therefore uses the group quota inode.

- Project quota's primary purpose is to track and monitor disk usage for directories. For this to occur, the directory inode must have the `XFS_DIFLAG_PROJINHERIT` flag set so all inodes created underneath the directory inherit the project ID.
- Inodes and blocks owned by ID zero do not have enforced quotas, but only quota accounting.
- Extended attributes do not contribute towards the ID's quota .
- To access each ID's quota information in the file, seek to the ID offset multiplied by the size of `xfs_dqblk_t` (136 bytes).



Quota information stored in the two inodes (in [data extents](#)) are an array of the `xfs_dqblk_t` structure where there is one instance for each ID in the system:

```
typedef struct xfs_disk_dquot {
    __be16    d_magic;
    __u8     d_version;
    __u8     d_flags;
    __be32   d_id;
    __be64   d_blk_hardlimit;
    __be64   d_blk_softlimit;
    __be64   d_ino_hardlimit;
    __be64   d_ino_softlimit;
    __be64   d_bcount;
    __be64   d_icount;
    __be32   d_itimer;
    __be32   d_btimer;
    __be16   d_iwarns;
    __be16   d_bwarns;
    __be32   d_pad0;
    __be64   d_rt_bhardlimit;
    __be64   d_rt_bsoftlimit;
    __be64   d_rt_bcount;
};
```

```

        __be32                d_rtbtimer;
        __be16                d_rtbwarns;
        __be16                d_pad;
    } xfs_disk_dquot_t;

typedef struct xfs_dqblk {
    xfs_disk_dquot_t          dd_diskdq;
    char                      dd_fill[32];
} xfs_dqblk_t;

```

d_magic

Specifies the signature where these two bytes are 0x4451 (XFS_DQUOT_MAGIC), or "DQ" in ASCII.

d_version

Specifies the structure version, currently this is one (XFS_DQUOT_VERSION).

d_flags

Specifies which type of ID the structure applies to:

```

#define XFS_DQ_USER          0x0001
#define XFS_DQ_PROJ         0x0002
#define XFS_DQ_GROUP        0x0004

```

d_id

The ID for the quota structure. This will be a uid, gid or projid based on the value of d_flags.

d_blk_hardlimit

Specifies the hard limit for the number of filesystem blocks the ID can own. The ID will not be able to use more space than this limit. If it is attempted, ENOSPC will be returned.

d_blk_softlimit

Specifies the soft limit for the number of filesystem blocks the ID can own. The ID can temporarily use more space than by d_blk_softlimit up to d_blk_hardlimit. If the space is not freed by the time limit specified by ID zero's d_bt看mer value, the ID will be denied more space until the total blocks owned goes below d_blk_softlimit.

d_ino_hardlimit

Specifies the hard limit for the number of inodes the ID can own. The ID will not be able to create or own any more inodes if d_icount reaches this value.

d_ino_softlimit

Specifies the soft limit for the number of inodes the ID can own. The ID can temporarily create or own more inodes than specified by d_ino_softlimit up to d_ino_hardlimit. If the inode count is not reduced by the time limit specified by ID zero's d_itimer value, the ID will be denied from creating or owning more inodes until the count goes below d_ino_softlimit.

d_bcount

Specifies how many filesystem blocks are actually owned by the ID.

d_icount

Specifies how many inodes are actually owned by the ID.

d_itimer

Specifies the time when the ID's `d_icoount` exceeded `d_ino_softlimit`. The soft limit will turn into a hard limit after the elapsed time exceeds ID zero's `d_itimer` value. When `d_icoount` goes back below `d_ino_softlimit`, `d_itimer` is reset back to zero.

d_btimer

Specifies the time when the ID's `d_bcount` exceeded `d_blk_softlimit`. The soft limit will turn into a hard limit after the elapsed time exceeds ID zero's `d_btimer` value. When `d_bcount` goes back below `d_blk_softlimit`, `d_btimer` is reset back to zero.

d_iwarns**d_bwarns****d_rtwarns**

Specifies how many times a warning has been issued. Currently not used.

d_rtblock_hardlimit

Specifies the hard limit for the number of real-time blocks the ID can own. The ID cannot own more space on the real-time subvolume beyond this limit.

d_rtblock_softlimit

Specifies the soft limit for the number of real-time blocks the ID can own. The ID can temporarily own more space than specified by `d_rtblock_softlimit` up to `d_rtblock_hardlimit`. If `d_rtblockcount` is not reduced by the time limit specified by ID zero's `d_rtblocktimer` value, the ID will be denied from owning more space until the count goes below `d_rtblock_softlimit`.

d_rtblockcount

Specifies how many real-time blocks are currently owned by the ID.

d_rtblocktimer

Specifies the time when the ID's `d_rtblockcount` exceeded `d_rtblock_softlimit`. The soft limit will turn into a hard limit after the elapsed time exceeds ID zero's `d_rtblocktimer` value. When `d_rtblockcount` goes back below `d_rtblock_softlimit`, `d_rtblocktimer` is reset back to zero.

Real-time Inodes

There are two inodes allocated to managing the real-time device's space, the Bitmap Inode and the Summary Inode.

Real-Time Bitmap Inode

The Bitmap Inode tracks the used/free space in the real-time device using an old-style bitmap. One bit is allocated per real-time extent. The size of an extent is specified by the [superblock's](#) `sb_rextsize` value.

The number of blocks used by the bitmap inode is equal to the number of real-time extents (`sb_rextents`) divided by the block size (`sb_blocksize`) and bits per byte. This value is stored in `sb_rmblocks`. The `nblocks` and extent array for the inode should match this.

```
xfstypes_t      xfs_ino_t      sb_rbmino;
```

Real-Time Summary Inode

```
xfstypes_t      xfs_ino_t      sb_rsumino;
```

Journaling Log

TODO: