Apple File System Reference

Developer

Contents

About Apple File System

Apple File System is the default file format used on Apple platforms. Apple File System is the successor to HFS Plus, so some aspects of its design intentionally follow HFS Plus to enable data migration from HFS Plus to Apple File System. Other aspects of its design address limitations with HFS Plus and enable features like cloning files, snapshots, encryption, and sharing free space between volumes.

Most apps interact with the file system using high-level interfaces provided by Foundation, which means most developers don't need to read this document. This document is for developers of software that interacts with the file system directly, without using any frameworks or the operating system — for example, a disk recovery utility or an implementation of Apple File System on another platform. The on-disk data structures described in this document make up the file system; software that interacts with them defines corresponding in-memory data structures.

> **Note**
>
> If you need to boot from an Apple File System volume, but don't need to mount the volume or interact with the file system directly, read Booting from an Apple File System Partition.

Layered Design

The Apple File System is conceptually divided into two layers, the container layer and the file-system layer. The container layer organizes file-system layer information and stores higher level information, like volume metadata, snapshots of the volume, and encryption state. The file-system layer is made up of the data structures that store information, like directory structures, file metadata, and file content. Many types are prefixed with `nx_` or `j_`, which indicates that they're part of the container layer or the file-system layer, respectively. The abbreviated prefixes don't have a meaningful long form; they're an artifact of how Apple's implementation was developed.

There are several design differences between the layers. Container objects are larger, with a typical size measured in blocks, and contain padding fields that keep data aligned on 64-bit boundaries, to avoid the performance penalty of unaligned memory access. File-system objects are smaller, with a typical size measured in bytes, and are almost always packed to minimize space used.

Numbers in both layers are stored on disk in little-endian order. Objects in both layers begin with a common header that enables object-oriented design patterns in implementations of Apple File System, although the layers have different headers. Container layer objects begin with an instance of `obj_phys_t` and file-system objects begin with an instance of `j_key_t`,

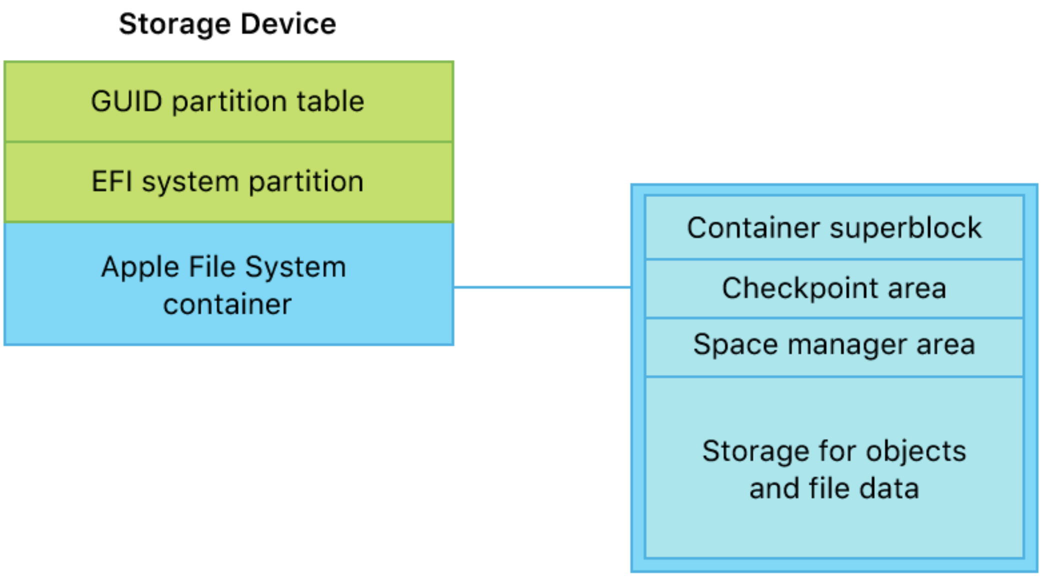
Container Layer

Container objects have an object identifier that you use to locate the object; the steps vary depending on how the object is stored:

- *Physical objects* are stored on disk at a particular physical block address.
- *Ephemeral objects* are stored in memory while the container is mounted and in a checkpoint when the container isn't mounted.
- *Virtual objects* are stored on disk at a location that you look up in an object map (an instance of `omap_phys_t`).

The object map includes a B-tree whose keys contain a transaction identifier and an object identifier and whose values contain a physical block address where the object is stored.

An Apple File System partition has a single container, which provides space management and crash protection. A container can contain multiple volumes (also known as file systems), each of which contains a directory structure for files and folders. For example, the figure below shows a storage device that has one Apple File System partition, and it shows the major divisions of the space inside that container.



Although there's only one container, there are several copies of the container superblock (an instance of `nx_superblock_t`) stored on disk. These copies hold the state of the container at past points in time. Block zero contains a copy of the container superblock that's used as part of the mounting process to find the checkpoints. Block zero is typically a copy of the latest container superblock, assuming the device was properly unmounted and was last modified by a correct Apple File System implementation. However, in practice, you use the block zero copy only to find the checkpoints and use the latest version from the checkpoint for everything else.

Within a container, the checkpoint mechanism and the copy-on-write approach to modifying objects enable crash protection. In-memory state is periodically written to disk in checkpoints, followed by a copy of the container superblock at that point in time. Checkpoint information is stored in two regions: The checkpoint descriptor area contains instances of `checkpoint_map_phys_t` and `nx_superblock_t`, and the checkpoint data area contains ephemeral objects that represent the in-memory state at the point in time when the checkpoint was written to disk.

When mounting a device, you use the most recent checkpoint information that's valid, as discussed in Mounting an Apple File System Partition. If the process of writing a checkpoint is interrupted, that checkpoint is invalid and therefore is ignored the next time the device is mounted, rolling the file system back to the last valid state. Because the checkpoint stores in-memory state, mounting an Apple File System partition includes reading the ephemeral objects from the checkpoint back into memory, re-creating that state in memory.

File-System Layer

File-system objects are made up of several records, and each record is stored as a key and value in a B-tree (an instance of `btree_node_phys_t`). For example, a typical directory object is made up of an inode record, several directory entry records, and an extended attributes record. A record contains an object identifier that's used to find it within the B-tree that contains it.

General-Purpose Types

Basic types that are used in a variety of contexts, and aren't associated with any particular functionality.

paddr_t

A physical address of an on-disk block.

```
typedef int64_t     paddr_t;
```

Negative numbers aren't valid addresses. This value is modeled as a signed integer to match IOKit.

prange_t

A range of physical addresses.

```
struct prange {
    paddr_t     pr_start_paddr;
    uint64_t    pr_block_count;
};
typedef struct prange prange_t;
```

pr_start_paddr

The first block in the range.

```
paddr_t pr_start_paddr;
```

pr_block_count

The number of blocks in the range.

```
uint64_t pr_block_count;
```

uuid_t

A universally unique identifier.

```
typedef unsigned char   uuid_t[16];
```

Objects

Depending on how they're stored, objects have some differences, the most important of which is the way you use an object identifier to find an object. At the container level, there are three storage methods for objects:

- *Ephemeral objects* are stored in memory for a mounted container, and are persisted across unmounts in a checkpoint. Ephemeral objects for a mounted partition can be modified in place while they're in memory, but they're always written back to disk as part of a new checkpoint. They're used for information that's frequently updated because of the performance benefits of in-place, in-memory changes.

- *Physical objects* are stored at a known block address on the disk, and are modified by writing the copy to a new location on disk. Because the object identifier for a physical object is its physical address, this copy-on-write behavior means that the modified copy has a different object identifier.

- *Virtual objects* are stored on disk at a block address that you look up using an object map. Virtual objects are also copied when they are modified; however, both the original and the modified copy have the same object identifier. When you look up a virtual object in an object map, you use a transaction identifier, in addition to the object identifier, to specify the point in time that you want.

Regardless of their storage, objects on disk are never modified in place, and modified copies of an object are always written to a new location on disk. To access an object, you need to know its storage and its identifier. For virtual objects, you also need a transaction identifier. The storage for an object is almost always implicit from the context in which that identifier appears. For example, the object identifier for the space manager is stored in the `nx_spaceman_oid` field of `nx_superblock_t`, and the documentation for that field says that the space manager is always an ephemeral object.

Object identifiers are unique inside the entire container, within their storage method. For example, no two virtual objects can have the same identifier — even when stored in different object maps — because their storage methods are the same. However, a virtual object and a physical object *can* have the same identifier because their storage methods are different. For information about determining the identifier for a new object, see `oid_t`.

When writing a new object to disk, fill all unused space in the block with zeros. Future versions of Apple File System add new fields at the end of a structure; zeroing out the uninitialized bytes makes it possible to determine whether data has been stored in a field that was added later, such as the `apfs_cloneinfo_xid` field of `apfs_superblock_t`.

obj_phys_t

A header used at the beginning of all objects.

```
struct obj_phys {
    uint8_t     o_cksum[MAX_CKSUM_SIZE];
    oid_t       o_oid;
    xid_t       o_xid;
    uint32_t    o_type;
    uint32_t    o_subtype;
};
typedef struct obj_phys obj_phys_t;

#define MAX_CKSUM_SIZE              8
```

o_cksum

The Fletcher 64 checksum of the object.

```
uint8_t o_cksum[MAX_CKSUM_SIZE];
```

o_oid

The object's identifier.

```
oid_t o_oid;
```

o_xid

The identifier of the most recent transaction that this object was modified in.

```
xid_t o_xid;
```

o_type

The object's type and flags.

```
uint32_t o_type;
```

An object type is a 32-bit value: The low 16 bits indicate the type using the values listed in Object Types, and the high 16 bits are flags using the values listed in Object Type Flags.

o_subtype

The object's subtype.

```
uint32_t o_subtype;
```

For the values used in this field, see Object Types.

Subtypes indicate the type of data stored in a data structure such as a B-tree. For example, a node in a B-tree that contains volume records has a type of `OBJECT_TYPE_BTREE_NODE` and a subtype of `OBJECT_TYPE_FS`.

MAX_CKSUM_SIZE

The number of bytes used for an object checksum.

```
#define MAX_CKSUM_SIZE 8
```

Supporting Data Types

Types used as unique identifiers within an object.

```
typedef uint64_t    oid_t;
typedef uint64_t    xid_t;
```

oid_t

An object identifier.

```
typedef uint64_t oid_t;
```

Objects are identified by this number as follows:

- For a physical object, its identifier is the logical block address on disk where the object is stored.
- For an ephemeral object, its identifier is a number.
- For a virtual object, its identifier is a number.

For more information about physical, ephemeral, or virtual objects, see Objects.

To determine the identifier for a new physical object, find a free block using the space manager, and use that block's address. To determine the identifier for a new ephemeral or virtual object, check the value of `nx_superblock_t.nx_next_oid`. New ephemeral and virtual object identifiers must be monotonically increasing.

> **Note**
>
> Although both ephemeral and virtual objects use `nx_next_oid` field of `nx_superblock_t` in Apple's implementation, this isn't guaranteed or required. Ephemeral and virtual objects are stored in different places, so it's valid to encounter (or create) an ephemeral object and a virtual object that have the same identifier.

xid_t

A transaction identifier.

```
typedef uint64_t xid_t;
```

Transactions are uniquely identified by a monotonically increasing number.

The number zero isn't a valid transaction identifier. Implementations of Apple File System can use it as a sentinel value in memory — for example, to refer to the current transaction — but must not let it appear on disk.

This data type is sufficiently large that you aren't expected to ever run out of transaction identifiers. For example, if you created 1,000,000 transactions per second, it would take more than 5,000 centuries to exhaust the available transaction identifiers.

If a new transaction identifier isn't available, that's an unrecoverable error. Identifiers aren't allowed to restart from one or to be reused.

Object Identifier Constants

Constants used for virtual objects that always have a given identifier.

```
#define OID_NX_SUPERBLOCK           1

#define OID_INVALID                 0ULL
#define OID_RESERVED_COUNT          1024
```

OID_NX_SUPERBLOCK

The ephemeral object identifier for the container superblock.

```
#define OID_NX_SUPERBLOCK 1
```

Although the container superblock is stored in memory like other ephemeral objects, it isn't saved on disk in the same area. For details, see Mounting an Apple File System Partition.

OID_INVALID

An invalid object identifier.

```
#define OID_INVALID 0ULL
```

OID_RESERVED_COUNT

The number of object identifiers that are reserved for objects with a fixed object identifier.

```
#define OID_RESERVED_COUNT 1024
```

This range of identifiers is reserved for physical, virtual, and ephemeral objects.

Currently, the only object with a reserved identifier is the container superblock, as described in `OID_NX_SUPERBLOCK`. All other object identifiers less than `OID_RESERVED_COUNT` are reserved by Apple.

Object Type Masks

Bit masks used to access specific portions of an object type.

```
#define OBJECT_TYPE_MASK                  0x0000ffff
#define OBJECT_TYPE_FLAGS_MASK            0xffff0000

#define OBJ_STORAGETYPE_MASK              0xc0000000
#define OBJECT_TYPE_FLAGS_DEFINED_MASK    0xf8000000
```

OBJECT_TYPE_MASK

The bit mask used to access the type.

```
#define OBJECT_TYPE_MASK 0x0000ffff
```

For the values that appear in this bit field, see Object Types.

OBJECT_TYPE_FLAGS_MASK

The bit mask used to access the flags.

```
#define OBJECT_TYPE_FLAGS_MASK 0xffff0000
```

For the values that appear in this bit field, see Object Type Flags.

OBJ_STORAGETYPE_MASK

The bit mask used to access the storage portion of the object type.

```
#define OBJ_STORAGETYPE_MASK 0xc0000000
```

For the values that appear in this bit field, see Object Type Flags.

OBJECT_TYPE_FLAGS_DEFINED_MASK

A bit mask of all bits for which flags are defined.

```
#define OBJECT_TYPE_FLAGS_DEFINED_MASK 0xf8000000
```

Object Types

Values used as types and subtypes by the `obj_phys_t` structure.

```
#define OBJECT_TYPE_NX_SUPERBLOCK        0x00000001

#define OBJECT_TYPE_BTREE                0x00000002
#define OBJECT_TYPE_BTREE_NODE           0x00000003

#define OBJECT_TYPE_SPACEMAN             0x00000005
#define OBJECT_TYPE_SPACEMAN_CAB         0x00000006
#define OBJECT_TYPE_SPACEMAN_CIB         0x00000007
#define OBJECT_TYPE_SPACEMAN_BITMAP      0x00000008
#define OBJECT_TYPE_SPACEMAN_FREE_QUEUE  0x00000009

#define OBJECT_TYPE_EXTENT_LIST_TREE     0x0000000a
#define OBJECT_TYPE_OMAP                 0x0000000b
#define OBJECT_TYPE_CHECKPOINT_MAP       0x0000000c

#define OBJECT_TYPE_FS                   0x0000000d
#define OBJECT_TYPE_FSTREE               0x0000000e
#define OBJECT_TYPE_BLOCKREFTREE         0x0000000f
#define OBJECT_TYPE_SNAPMETATREE         0x00000010

#define OBJECT_TYPE_NX_REAPER            0x00000011
#define OBJECT_TYPE_NX_REAP_LIST         0x00000012
#define OBJECT_TYPE_OMAP_SNAPSHOT        0x00000013
#define OBJECT_TYPE_EFI_JUMPSTART        0x00000014
#define OBJECT_TYPE_FUSION_MIDDLE_TREE   0x00000015
#define OBJECT_TYPE_NX_FUSION_WBC        0x00000016
#define OBJECT_TYPE_NX_FUSION_WBC_LIST   0x00000017
#define OBJECT_TYPE_ER_STATE             0x00000018

#define OBJECT_TYPE_GBITMAP              0x00000019
#define OBJECT_TYPE_GBITMAP_TREE         0x0000001a
#define OBJECT_TYPE_GBITMAP_BLOCK        0x0000001b
```

```
#define OBJECT_TYPE_ER_RECOVERY_BLOCK   0x0000001c
#define OBJECT_TYPE_SNAP_META_EXT       0x0000001d
#define OBJECT_TYPE_INTEGRITY_META      0x0000001e
#define OBJECT_TYPE_FEXT_TREE           0x0000001f
#define OBJECT_TYPE_RESERVED_20         0x00000020

#define OBJECT_TYPE_INVALID             0x00000000
#define OBJECT_TYPE_TEST                0x000000ff

#define OBJECT_TYPE_CONTAINER_KEYBAG    'keys'
#define OBJECT_TYPE_VOLUME_KEYBAG       'recs'
#define OBJECT_TYPE_MEDIA_KEYBAG        'mkey'
```

The value of `obj_phys_t.o_type & OBJECT_TYPE_MASK` is one of these constants.

OBJECT_TYPE_NX_SUPERBLOCK

A container superblock (`nx_superblock_t`).

```
#define OBJECT_TYPE_NX_SUPERBLOCK 0x00000001
```

OBJECT_TYPE_BTREE

A B-tree root node (`btree_node_phys_t`).

```
#define OBJECT_TYPE_BTREE 0x00000002
```

OBJECT_TYPE_BTREE_NODE

A B-tree node (`btree_node_phys_t`).

```
#define OBJECT_TYPE_BTREE_NODE 0x00000003
```

OBJECT_TYPE_SPACEMAN

A space manager (`spaceman_phys_t`).

```
#define OBJECT_TYPE_SPACEMAN 0x00000005
```

OBJECT_TYPE_SPACEMAN_CAB

A chunk-info address block (`cib_addr_block`) used by the space manager.

```
#define OBJECT_TYPE_SPACEMAN_CAB 0x00000006
```

OBJECT_TYPE_SPACEMAN_CIB

A chunk-info block (`chunk_info_block`) used by the space manager.

```
#define OBJECT_TYPE_SPACEMAN_CIB 0x00000007
```

OBJECT_TYPE_SPACEMAN_BITMAP

A free-space bitmap used by the space manager.

```
#define OBJECT_TYPE_SPACEMAN_BITMAP 0x00000008
```

OBJECT_TYPE_SPACEMAN_FREE_QUEUE

A free-space queue (a mapping from `spaceman_free_queue_key_t` to `spaceman_free_queue_t`), used by the space manager.

```
#define OBJECT_TYPE_SPACEMAN_FREE_QUEUE 0x00000009
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_EXTENT_LIST_TREE

An extents-list tree (a mapping from `paddr_t` to `prange_t`).

```
#define OBJECT_TYPE_EXTENT_LIST_TREE 0x0000000a
```

The keys are an offset into the logical start of the extent, and the value is the physical location where that data is stored.

This type is used only as a subtype of a tree.

OBJECT_TYPE_OMAP

As a type, an object map (`omap_phys_t`); as a subtype, a tree that stores the records of an object map (a mapping from `omap_key_t` to `omap_val_t`).

```
#define OBJECT_TYPE_OMAP 0x0000000b
```

OBJECT_TYPE_CHECKPOINT_MAP

A checkpoint map (`checkpoint_map_phys_t`).

```
#define OBJECT_TYPE_CHECKPOINT_MAP 0x0000000c
```

OBJECT_TYPE_FS

A volume (`apfs_superblock_t`).

```
#define OBJECT_TYPE_FS 0x0000000d
```

OBJECT_TYPE_FSTREE

A tree containing file-system records.

```
#define OBJECT_TYPE_FSTREE 0x0000000e
```

This type is used only as a subtype of a tree.

The keys and values stored in the tree vary. Each key begins with `j_key_t`, which contains a field that indicates the type of that key and its value.

OBJECT_TYPE_BLOCKREFTREE

A tree containing extent references (a mapping from `j_phys_ext_key_t` to `j_phys_ext_val_t`).

```
#define OBJECT_TYPE_BLOCKREFTREE 0x0000000f
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_SNAPMETATREE

A tree containing snapshot metadata for a volume (a mapping from `j_snap_metadata_key_t` to `j_snap_metadata_val_t`).

```
#define OBJECT_TYPE_SNAPMETATREE 0x00000010
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_NX_REAPER

A reaper (`nx_reaper_phys_t`).

```
#define OBJECT_TYPE_NX_REAPER 0x00000011
```

OBJECT_TYPE_NX_REAP_LIST

A reaper list (`nx_reap_list_phys_t`).

```
#define OBJECT_TYPE_NX_REAP_LIST 0x00000012
```

OBJECT_TYPE_OMAP_SNAPSHOT

A tree containing information about snapshots of an object map (a mapping from `xid_t` to `omap_snapshot_t`).

```
#define OBJECT_TYPE_OMAP_SNAPSHOT 0x00000013
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_EFI_JUMPSTART

EFI information used for booting (`nx_efi_jumpstart_t`).

```
#define OBJECT_TYPE_EFI_JUMPSTART 0x00000014
```

OBJECT_TYPE_FUSION_MIDDLE_TREE

A tree used for Fusion devices to track blocks from the hard drive that are cached on the solid-state drive (a mapping from `fusion_mt_key_t` to `fusion_mt_val_t`).

```
#define OBJECT_TYPE_FUSION_MIDDLE_TREE 0x00000015
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_NX_FUSION_WBC

A write-back cache state (fusion_wbc_phys_t) used for Fusion devices.

```
#define OBJECT_TYPE_NX_FUSION_WBC 0x00000016
```

OBJECT_TYPE_NX_FUSION_WBC_LIST

A write-back cache list (fusion_wbc_list_phys_t) used for Fusion devices.

```
#define OBJECT_TYPE_NX_FUSION_WBC_LIST 0x00000017
```

OBJECT_TYPE_ER_STATE

An encryption-rolling state (er_state_phys_t).

```
#define OBJECT_TYPE_ER_STATE 0x00000018
```

OBJECT_TYPE_GBITMAP

A general-purpose bitmap (gbitmap_phys_t).

```
#define OBJECT_TYPE_GBITMAP 0x00000019
```

OBJECT_TYPE_GBITMAP_TREE

A B-tree of general-purpose bitmaps (a mapping from uint64_t to uint64_t).

```
#define OBJECT_TYPE_GBITMAP_TREE 0x0000001a
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_GBITMAP_BLOCK

A block containing a general-purpose bitmap (gbitmap_block_phys_t).

```
#define OBJECT_TYPE_GBITMAP_BLOCK 0x0000001b
```

OBJECT_TYPE_ER_RECOVERY_BLOCK

Information that can be used to recover from a system crash if one occurs during the encryption rolling process (er_recovery_block_phys_t).

```
#define OBJECT_TYPE_ER_RECOVERY_BLOCK 0x0000001c
```

OBJECT_TYPE_SNAP_META_EXT

Additional metadata about snapshots (snap_meta_ext_obj_phys_t.)

```
#define OBJECT_TYPE_SNAP_META_EXT 0x0000001d
```

OBJECT_TYPE_INTEGRITY_META

An integrity metadata object (integrity_meta_phys_t).

```
#define OBJECT_TYPE_INTEGRITY_META 0x0000001e
```

OBJECT_TYPE_FEXT_TREE

A B-tree of file extents (a mapping from fext_tree_key_t to fext_tree_val_t).

```
#define OBJECT_TYPE_FEXT_TREE 0x0000001f
```

This type is used only as a subtype of a tree.

OBJECT_TYPE_RESERVED_20

Reserved.

```
#define OBJECT_TYPE_RESERVED_20 0x00000020
```

OBJECT_TYPE_INVALID

As a type, an invalid object; as a subtype, an object with no subtype.

```
#define OBJECT_TYPE_INVALID 0x00000000
```

OBJECT_TYPE_TEST

Reserved for testing.

```
#define OBJECT_TYPE_TEST 0x000000ff
```

Don't create objects of this type on disk. If you find an object of this type in production, file a bug against the Apple File System implementation.

This type isn't reserved by Apple; non-Apple implementations of Apple File System can use it during testing.

OBJECT_TYPE_CONTAINER_KEYBAG

A container's keybag (media_keybag_t).

```
#define OBJECT_TYPE_CONTAINER_KEYBAG 'keys'
```

OBJECT_TYPE_VOLUME_KEYBAG

A volume's keybag (media_keybag_t).

```
#define OBJECT_TYPE_VOLUME_KEYBAG 'recs'
```

OBJECT_TYPE_MEDIA_KEYBAG

A media keybag (media_keybag_t).

```
#define OBJECT_TYPE_MEDIA_KEYBAG 'mkey'
```

Object Type Flags

The flags used in the object type to provide additional information.

```
#define OBJ_VIRTUAL                   0x00000000
#define OBJ_EPHEMERAL                 0x80000000
#define OBJ_PHYSICAL                  0x40000000

#define OBJ_NOHEADER                  0x20000000
#define OBJ_ENCRYPTED                 0x10000000
#define OBJ_NONPERSISTENT             0x08000000
```

The value of `obj_phys_t.o_type & OBJECT_TYPE_FLAGS_MASK` uses these constants. The value of `obj_phys_t.o_type & OBJ_STORAGETYPE_MASK` uses only `OBJ_VIRTUAL`, `OBJ_EPHEMERAL`, and `OBJ_PHYSICAL`.

The flags on an object's type must indicate whether the object is virtual, ephemeral, or physical by setting either the `OBJ_EPHEMERAL` or `OBJ_PHYSICAL` flag, or setting neither flag. An object type that contains both flags is invalid.

The absence of both flags indicates a virtual object. The `OBJ_VIRTUAL` constant is defined to allow code that tests for virtual objects to match code testing for physical or ephemeral objects, even though there's no corresponding bit set in the object's type. For example:

```
obj_phys_t obj = /* assume this exists */
if   ((obj.o_type & OBJ_STORAGETYPE_MASK) == OBJ_VIRTUAL)   { ... }
elif ((obj.o_type & OBJ_STORAGETYPE_MASK) == OBJ_EPHEMERAL) { ... }
elif ((obj.o_type & OBJ_STORAGETYPE_MASK) == OBJ_PHYSICAL)  { ... }
else { /* error */ }
```

OBJ_VIRTUAL

A virtual object.

```
#define OBJ_VIRTUAL 0x00000000
```

OBJ_EPHEMERAL

An ephemeral object.

```
#define OBJ_EPHEMERAL 0x80000000
```

OBJ_PHYSICAL

A physical object.

```
#define OBJ_PHYSICAL 0x40000000
```

OBJ_NOHEADER

An object stored without an `obj_phys_t` header.

```
#define OBJ_NOHEADER 0x20000000
```

This flag is used, for example, by the space manager's bitmap.

OBJ_ENCRYPTED

An encrypted object.

```
#define OBJ_ENCRYPTED 0x10000000
```

OBJ_NONPERSISTENT

An ephemeral object that isn't persisted across unmounting.

```
#define OBJ_NONPERSISTENT 0x08000000
```

Objects with this flag never appear on disk. If you find an object of this type in production, file a bug against the Apple File System implementation.

This flag isn't reserved by Apple; non-Apple implementations of Apple File System can mark their runtime-only data structures with `OBJ_NONPERSISTENT | OBJ_EPHEMERAL`.

EFI Jumpstart

A partition formatted using the Apple File System contains an embedded EFI driver that's used to boot a machine from that partition.

Booting from an Apple File System Partition

You can locate the EFI driver by reading a few data structures, starting at a known physical address on disk. You don't need any support for reading or mounting Apple File System to locate the EFI driver. This design intentionally simplifies the steps needed to boot, which means the code needed to boot a piece of hardware or virtualization software can likewise be simpler. To boot using the embedded EFI driver, do the following:

1. Read physical block zero from the partition. This block contains a copy of the container superblock, which is an instance of `nx_superblock_t`.
2. Read the `nx_o` field of the superblock, which is an instance of `obj_phys_t`. Then read the `o_cksum` field of the `nx_o` field of the superblock, which contains the Fletcher 64 checksum of the object. Verify that the checksum is correct.
3. Read the `nx_magic` field of the superblock. Verify that the field's value is `NX_MAGIC` (the four-character code `'BSXN'`).
4. Read the `nx_efi_jumpstart` field of the superblock. This field contains the physical block address (also referred to as the physical object identifier) for the EFI jumpstart information, which is an instance of `nx_efi_jumpstart_t`.
5. Read the `nej_magic` field of the EFI jumpstart information. Verify that the field's value is `NX_EFI_JUMPSTART_MAGIC` (the four-character code `'RDSJ'`).
6. Read the `nej_o` field of the EFI jumpstart information, which is an instance of `obj_phys_t`. Then read the `o_cksum` field of the `nej_o` field of the jumpstart information, which contains the Fletcher 64 checksum of the object. Verify that the checksum is correct.
7. Read the `nej_version` field of the EFI jumpstart information. This field contains the EFI jumpstart version number. Verify that the field's value is `NX_EFI_JUMPSTART_VERSION` (the number one).
8. Read the `nej_efi_file_len` field of the jumpstart information. This field contains the length, in bytes, of the embedded EFI driver. Allocate a contiguous block of memory of at least that size, which you'll later use to store the EFI driver.
9. Read the `nej_num_extents` field of the jumpstart information, and then read that number of `prange_t` records from the `nej_rec_extents` field.
10. Read each extent of the EFI driver into memory, contiguously, in the order they're listed.
11. Load the EFI driver and start executing it.

Implementation Outline

The code listing below shows one way to boot using the embedded EFI driver, assuming the functions listed at the beginning are defined.

```
nx_superblock_t* read_superblock(int address) {
```

```
    // Read the given physical block from disk
    // and return its contents as a pointer to an nx_superblock_t.
}

nx_efi_jumpstart_t* read_jumpstart(int address) {
    // Read the given physical block from disk
    // and return its contents as a pointer to an nx_efi_jumpstart_t.
}

void* read_block(int address) {
    // Read the given physical block from disk
    // and return a pointer to its contents.
}

uint8_t* fletcher64_checksum(void* object) {
    // Calculate and return a Fletcher 64 checksum.
}

void assert_arrays_equal(int length, uint8_t* x, uint8_t* y) {
    // Assert that the given arrays contain the same data.
}

void load_and_execute(void* address) {
    // Load the EFI driver at the specified address
    // and then start executing it.
}

int main() {
    nx_superblock_t* superblock = read_superblock(0);
    assert(superblock->nx_o.o_cksum == fletcher64_checksum(&superblock));
    assert(superblock->nx_magic == 'BSXN');

    paddr_t jumpstart_address = superblock->nx_efi_jumpstart;
    nx_efi_jumpstart_t* jumpstart = read_jumpstart(jumpstart_address);

    uint8_t* checksum = fletcher64_checksum(&jumpstart);
    assert_arrays_equal(MAX_CKSUM_SIZE, jumpstart->nej_o.o_cksum, checksum);
    assert(jumpstart->nej_version == 1);

    void* efi_driver = malloc(jumpstart->nej_efi_file_len);
    void* efi_driver_cursor = efi_driver;

    for (int i = 0; i < jumpstart->nej_num_extents; i++) {
        prange_t efi_extent_address = jumpstart->nej_rec_extents[i];
        for (int j = 0; j < efi_extent_address.pr_block_count; j++) {
            void* efi_block = read_block(efi_extent_address.pr_start_paddr + j);
            memcpy(efi_driver_cursor, efi_block, superblock->nx_block_size);
            efi_driver_cursor += superblock->nx_block_size;
```

```
        }
    }

    load_and_execute(efi_driver);

    return 0;
}
```

nx_efi_jumpstart_t

Information about the embedded EFI driver that's used to boot from an Apple File System partition.

```
struct nx_efi_jumpstart {
    obj_phys_t  nej_o;
    uint32_t    nej_magic;
    uint32_t    nej_version;
    uint32_t    nej_efi_file_len;
    uint32_t    nej_num_extents;
    uint64_t    nej_reserved[16];
    prange_t    nej_rec_extents[];
};
typedef struct nx_efi_jumpstart nx_efi_jumpstart_t;
#define NX_EFI_JUMPSTART_MAGIC      'RDSJ'
#define NX_EFI_JUMPSTART_VERSION    1
```

nej_o

The object's header.

```
obj_phys_t nej_o;
```

nej_magic

A number that can be used to verify that you're reading an instance of `nx_efi_jumpstart_t`.

```
uint32_t nej_magic;
```

The value of this field is always `NX_EFI_JUMPSTART_MAGIC`.

nej_version

The version of this data structure.

```
uint32_t nej_version;
```

The value of this field is always `NX_EFI_JUMPSTART_VERSION`.

nej_efi_file_len

The size, in bytes, of the embedded EFI driver.

```
uint32_t nej_efi_file_len;
```

nej_num_extents

The number of extents in the array.

```
uint32_t nej_num_extents;
```

nej_reserved

Reserved.

```
uint64_t nej_reserved[16];
```

Populate this field with zero when you create a new instance, and preserve its value when you modify an existing instance.

nej_rec_extents

The locations where the EFI driver is stored.

```
prange_t nej_rec_extents[];
```

NX_EFI_JUMPSTART_MAGIC

The value of the `nej_magic` field.

```
#define NX_EFI_JUMPSTART_MAGIC 'RDSJ'
```

This magic number was chosen because in hex dumps it appears as "JSDR", which is an abbreviated form of *jumpstart driver record*.

NX_EFI_JUMPSTART_VERSION

The version number for the EFI jumpstart.

```
#define NX_EFI_JUMPSTART_VERSION 1
```

Partition UUIDs

Partition types used in GUID partition table entries.

```
#define APFS_GPT_PARTITION_UUID     "7C3457EF-0000-11AA-AA11-00306543ECAC"
```

APFS_GPT_PARTITION_UUID

The partition type for a partition that contains an Apple File System container.

```
#define APFS_GPT_PARTITION_UUID "7C3457EF-0000-11AA-AA11-00306543ECAC"
```

Container

The container includes several top-level objects that are shared by all of the container's volumes:

- *Checkpoint description and data areas* store ephemeral objects in a way that provides crash protection. At the end of each transaction, new state is saved by writing a checkpoint.
- *The space manager* keeps track of available space within the container and is used to allocate and free blocks that store objects and file data.
- *The reaper* manages the deletion of objects that are too large to be deleted in the time between transactions. It keeps track of the deletion state so these objects can be deleted across multiple transactions.

The container superblock describes the location of all of these objects.

Because a single container can have multiple volumes, configurations that would require multiple partitions under other file systems can usually share a single partition with Apple File System. For example, a drive can be configured with two bootable volumes — one with a shipping version of macOS and one with a beta version — as well as a user data volume. All three of these volumes share free space, meaning you don't have to decide ahead of time how to divide space between them.

Mounting an Apple File System Partition

To mount the volumes of a partition that's formatted using the Apple File System, do the following:

1. Read block zero of the partition. This block contains a copy of the container superblock (an instance of `nx_superblock_t`). It might be a copy of the latest version or an old version, depending on whether the drive was unmounted cleanly.
2. Use the block-zero copy of the container superblock to locate the checkpoint descriptor area by reading the `nx_xp_desc_base` field.
3. Read the entries in the checkpoint descriptor area, which are instances of `checkpoint_map_phys_t` or `nx_superblock_t`.
4. Find the container superblock that has the largest transaction identifier and isn't malformed. For example, confirm that its magic number and checksum are valid. That superblock and its checkpoint-mapping blocks comprise the *latest valid checkpoint*. The superblock's fields, like `nx_xp_desc_blocks` and `nx_data_len`, indicate which checkpoint-mapping blocks belong to that superblock.

   > **Note**
   >
   > The checkpoint description area is a ring buffer stored as an array. Walking backward from the latest valid superblock to read all of its checkpoint-mapping blocks sometimes requires wrapping around from the first block to the last block.

5. Read the ephemeral objects listed in the checkpoint from the checkpoint data area into memory. If any of the ephemeral objects is malformed, the checkpoint that contains that object is malformed; go back to the previous step and mount from an older checkpoint.

The details of this step vary. For example, if you're mounting the partition read-only and performance isn't a consideration, you can skip this step and read from the checkpoint every time you need to access an ephemeral object.

6. Locate the container object map using the nx_omap_oid field of the container superblock.
7. Read the list of volumes from the nx_fs_oid field of the container superblock. If you're mounting only a particular volume, you can ignore the virtual object identifiers for the other volumes.
8. For each volume, look up the specified virtual object identifier in the container object map to locate the volume superblock (an instance of apfs_superblock_t). If you're mounting only a particular volume, you can skip this step for the other volumes.
9. For each volume, read the root file system tree's virtual object identifier from the apfs_root_tree_oid field, and then look it up in the volume object map indicated by the apfs_omap_oid field. If you're mounting only a particular volume, you can skip this step for the other volumes.
10. Walk the root file system tree as needed by your implementation to mount the file system.

nx_superblock_t

A container superblock.

```
struct nx_superblock {
    obj_phys_t  nx_o;
    uint32_t    nx_magic;
    uint32_t    nx_block_size;
    uint64_t    nx_block_count;

    uint64_t    nx_features;
    uint64_t    nx_readonly_compatible_features;
    uint64_t    nx_incompatible_features;

    uuid_t      nx_uuid;

    oid_t       nx_next_oid;
    xid_t       nx_next_xid;

    uint32_t    nx_xp_desc_blocks;
    uint32_t    nx_xp_data_blocks;
    paddr_t     nx_xp_desc_base;
    paddr_t     nx_xp_data_base;
    uint32_t    nx_xp_desc_next;
    uint32_t    nx_xp_data_next;
    uint32_t    nx_xp_desc_index;
    uint32_t    nx_xp_desc_len;
    uint32_t    nx_xp_data_index;
    uint32_t    nx_xp_data_len;

    oid_t       nx_spaceman_oid;
    oid_t       nx_omap_oid;
```

```
    oid_t       nx_reaper_oid;

    uint32_t    nx_test_type;

    uint32_t    nx_max_file_systems;
    oid_t       nx_fs_oid[NX_MAX_FILE_SYSTEMS];
    uint64_t    nx_counters[NX_NUM_COUNTERS];
    prange_t    nx_blocked_out_prange;
    oid_t       nx_evict_mapping_tree_oid;
    uint64_t    nx_flags;
    paddr_t     nx_efi_jumpstart;
    uuid_t      nx_fusion_uuid;
    prange_t    nx_keylocker;
    uint64_t    nx_ephemeral_info[NX_EPH_INFO_COUNT];

    oid_t       nx_test_oid;

    oid_t       nx_fusion_mt_oid;
    oid_t       nx_fusion_wbc_oid;
    prange_t    nx_fusion_wbc;

    uint64_t    nx_newest_mounted_version;

    prange_t    nx_mkb_locker;

};
typedef struct nx_superblock nx_superblock_t;

#define NX_MAGIC                        'BSXN'
#define NX_MAX_FILE_SYSTEMS             100

#define NX_EPH_INFO_COUNT               4
#define NX_EPH_MIN_BLOCK_COUNT          8
#define NX_MAX_FILE_SYSTEM_EPH_STRUCTS  4
#define NX_TX_MIN_CHECKPOINT_COUNT      4
#define NX_EPH_INFO_VERSION_1           1
```

Note that all fields are 64-bit aligned.

nx_o

The object's header.

```
obj_phys_t nx_o;
```

nx_magic

A number that can be used to verify that you're reading an instance of `nx_superblock_t`.

```
uint32_t nx_magic;
```

The value of this field is always NX_MAGIC.

nx_block_size

The logical block size used in the Apple File System container.

```
uint32_t nx_block_size;
```

This size is often the same as the block size used by the underlying storage device, but it can also be an integer multiple of the device's block size.

nx_block_count

The total number of logical blocks available in the container.

```
uint64_t nx_block_count;
```

nx_features

A bit field of the optional features being used by this container.

```
uint64_t nx_features;
```

For the values used in this bit field, see Optional Container Feature Flags.

If your implementation doesn't implement an optional feature that's in use, ignore that feature in this list and mount the container's volumes as usual.

nx_readonly_compatible_features

A bit field of the read-only compatible features being used by this container.

```
uint64_t nx_readonly_compatible_features;
```

For the values used in this bit field, see Read-Only Compatible Container Feature Flags.

If your implementation doesn't implement a read-only compatible feature that's in use, mount the container's volumes as read-only.

nx_incompatible_features

A bit field of the backward-incompatible features being used by this container.

```
uint64_t nx_incompatible_features;
```

For the values used in this bit field, see Incompatible Container Feature Flags.

If your implementation doesn't implement a read-only feature that's in use, it must not mount the container's volumes.

nx_uuid

The universally unique identifier of this container.

```
uuid_t nx_uuid;
```

nx_next_oid

The next object identifier to be used for a new ephemeral or virtual object.

```
oid_t nx_next_oid;
```

nx_next_xid

The next transaction to be used.

```
xid_t nx_next_xid;
```

nx_xp_desc_blocks

The number of blocks used by the checkpoint descriptor area.

```
uint32_t nx_xp_desc_blocks;
```

The highest bit of this number is used as a flag, as discussed in `nx_xp_desc_base`. Ignore that bit when accessing this field as a count.

nx_xp_data_blocks

The number of blocks used by the checkpoint data area.

```
uint32_t nx_xp_data_blocks;
```

The highest bit of this number is used as a flag, as discussed in `nx_xp_data_base`. Ignore that bit when accessing this field as a count.

nx_xp_desc_base

Either the base address of the checkpoint descriptor area or the physical object identifier of a tree that contains the address information.

```
paddr_t nx_xp_desc_base;
```

If the highest bit of `nx_xp_desc_blocks` is zero, the checkpoint descriptor area is contiguous and this field contains the address of the first block. Otherwise, the checkpoint descriptor area isn't contiguous and this field contains the physical object identifier of a B-tree. The tree's keys are block offsets into the checkpoint descriptor area, and its values are instances of `prange_t` that contain the fragment's size and location.

nx_xp_data_base

Either the base address of the checkpoint data area or the physical object identifier of a tree that contains the address information.

```
paddr_t nx_xp_data_base;
```

If the highest bit of `nx_xp_data_blocks` is zero, the checkpoint data area is contiguous and this field contains the address of the first block. Otherwise, the checkpoint data area isn't contiguous and this field contains the object identifier of a B-tree. The tree's keys are block offsets into the checkpoint data area, and its values are instances of `prange_t` that contain the fragment's size and location.

nx_xp_desc_next

The next index to use in the checkpoint descriptor area.

```
uint32_t nx_xp_desc_next;
```

If the superblock is part of a checkpoint, this field must have a value. Otherwise, ignore the value of this field when reading, and use zero as the value when creating a new instance. For example, this field has no meaning for the copy of the superblock that's stored in block zero.

nx_xp_data_next

The next index to use in the checkpoint data area.

```
uint32_t nx_xp_data_next;
```

If the superblock is part of a checkpoint, this field must have a value. Otherwise, ignore the value of this field when reading, and use zero as the value when creating a new instance. For example, this field has no meaning for the copy of the superblock that's stored in block zero.

nx_xp_desc_index

The index of the first valid item in the checkpoint descriptor area.

```
uint32_t nx_xp_desc_index;
```

If the superblock is part of a checkpoint, this field must have a value. Otherwise, ignore the value of this field when reading, and use zero as the value when creating a new instance. For example, this field has no meaning for the copy of the superblock that's stored in block zero.

nx_xp_desc_len

The number of blocks in the checkpoint descriptor area used by the checkpoint that this superblock belongs to.

```
uint32_t nx_xp_desc_len;
```

If the superblock is part of a checkpoint, this field must have a value. Otherwise, ignore the value of this field when reading, and use zero as the value when creating a new instance. For example, this field has no meaning for the copy of the superblock that's stored in block zero.

nx_xp_data_index

The index of the first valid item in the checkpoint data area.

```
uint32_t nx_xp_data_index;
```

If the superblock is part of a checkpoint, this field must have a value. Otherwise, ignore the value of this field when reading, and use zero as the value when creating a new instance. For example, this field has no meaning for the copy of the superblock that's stored in block zero.

nx_xp_data_len

The number of blocks in the checkpoint data area used by the checkpoint that this superblock belongs to.

```
uint32_t nx_xp_data_len;
```

If the superblock is part of a checkpoint, this field must have a value. Otherwise, ignore the value of this field when reading, and use zero as the value when creating a new instance. For example, this field has no meaning for the copy of the superblock that's stored in block zero.

nx_spaceman_oid

The ephemeral object identifier for the space manager.

```
oid_t nx_spaceman_oid;
```

nx_omap_oid

The physical object identifier for the container's object map.

```
oid_t nx_omap_oid;
```

nx_reaper_oid

The ephemeral object identifier for the reaper.

```
oid_t nx_reaper_oid;
```

nx_test_type

Reserved for testing.

```
uint32_t nx_test_type;
```

This field never has a value other than zero on disk. If you find another value in production, file a bug against the Apple File System implementation.

This field isn't reserved by Apple; non-Apple implementations of Apple File System can use it to store an object type during testing.

nx_max_file_systems

The maximum number of volumes that can be stored in this container.

```
uint32_t nx_max_file_systems;
```

To calculate this value, divide the size of the container by 512 MiB and round up. For example, a container with 1.3 GiB of space can contain three volumes. This value must not be larger than the value of `NX_MAX_FILE_SYSTEMS`.

nx_fs_oid

An array of virtual object identifiers for volumes.

```
oid_t nx_fs_oid[NX_MAX_FILE_SYSTEMS];
```

The objects' types are all `OBJECT_TYPE_BTREE` and their subtypes are all `OBJECT_TYPE_FSTREE`.

nx_counters

An array of counters that store information about the container.

```
uint64_t nx_counters[NX_NUM_COUNTERS];
```

These counters are primarily intended to help during development and debugging of Apple File System implementations. For the meaning of these counters, see nx_counter_id_t.

nx_blocked_out_prange

The physical range of blocks where space will not be allocated.

```
prange_t nx_blocked_out_prange;
```

This field is used with nx_evict_mapping_tree_oid while shrinking a partition. If nothing is currently blocked out, the value of nx_blocked_out_prange.pr_block_count is zero and the value of nx_blocked_out_prange.pr_start_paddr is ignored.

nx_evict_mapping_tree_oid

The physical object identifier of a tree used to keep track of objects that must be moved out of blocked-out storage.

```
oid_t nx_evict_mapping_tree_oid;
```

The keys in this tree are physical addresses of blocks that must be moved, and the values are instances of evict_mapping_val_t that describe where the blocks are being moved to.

This identifier is valid only while shrinking a partition. First, the blocks to be removed from the partition are added to the nx_blocked_out_prange field. Next, every object that's stored in a blocked-out range is added to this tree. Finally, every object in this tree has space allocated and is moved into the new space. Because the space manager honors the blocked-out range, data is never moved from one blocked-out address to another address that's also blocked out. After all data has been removed from the blocked-out range and this tree is empty, the partition shrinks and the block count of nx_blocked_out_prange is set to zero, which clears the field.

nx_flags

Other container flags.

```
uint64_t nx_flags;
```

For the values used in this bit field, see Container Flags.

nx_efi_jumpstart

The physical object identifier of the object that contains EFI driver data extents.

```
paddr_t nx_efi_jumpstart;
```

The object is an instance of nx_efi_jumpstart_t.

nx_fusion_uuid

The universally unique identifier of the container's Fusion set, or zero for non-Fusion containers.

```
uuid_t nx_fusion_uuid;
```

The hard drive and the solid-state drive each have a partition, which combine to make a single container. Each partition has its own copy of the container superblock at block zero, and each copy has the same value for the low 127 bits of this field. The highest bit is one for the Fusion set's main device and zero for the second-tier device.

nx_keylocker

The location of the container's keybag.

```
prange_t nx_keylocker;
```

The data at this location is an instance of `kb_locker_t`.

nx_ephemeral_info

An array of fields used in the management of ephemeral data.

```
uint64_t nx_ephemeral_info[NX_EPH_INFO_COUNT];
```

The first array entry records information about how the checkpoint data area's size was chosen as follows:

```
nx_ephemeral_info[0] = (min_block_count << 32)
                     | ((NX_MAX_FILE_SYSTEM_EPH_STRUCTS & 0xFFFF) << 16)
                     | NX_EPH_INFO_VERSION_1;
```

The value of `min_block_count` depends on the size of the container. If the container is larger than 128 MiB, it takes the value of `NX_EPH_MIN_BLOCK_COUNT`. Otherwise, it takes the value of `spaceman_phys_t.sm_fq[SFQ_MAIN].sfq_tree_node_limit` from the space manager.

nx_test_oid

Reserved for testing.

```
oid_t nx_test_oid;
```

This field never has a value other than zero on disk. If you find another value in production, file a bug against the Apple File System implementation.

This field isn't reserved by Apple; non-Apple implementations of Apple File System can use it to store an object identifier during testing.

nx_fusion_mt_oid

The physical object identifier of the Fusion middle tree (a B-tree mapping `fusion_mt_key_t` to `fusion_mt_val_t`), or zero if for non-Fusion drives.

```
oid_t nx_fusion_mt_oid;
```

nx_fusion_wbc_oid

The ephemeral object identifier of the Fusion write-back cache state (fusion_wbc_phys_t), or zero for non-Fusion drives.

```
oid_t nx_fusion_wbc_oid;
```

nx_fusion_wbc

The blocks used for the Fusion write-back cache area, or zero for non-Fusion drives.

```
prange_t nx_fusion_wbc;
```

nx_newest_mounted_version

Reserved.

```
uint64_t nx_newest_mounted_version;
```

Apple's implementation uses this field to record the newest version of the software that ever mounted the container. Other implementations of the Apple file System must not modify this field.

This integer is understood as a fixed-point decimal number of the form `aaaaaaa.bbb.ccc.ddd.eee` where a is a major version number and b, c, d, and e are minor versions.

nx_mkb_locker

Wrapped media key.

```
prange_t nx_mkb_locker;
```

NX_MAGIC

The value of the nx_magic field.

```
#define NX_MAGIC 'BSXN'
```

This magic number was chosen because in hex dumps it appears as "NXSB", which is an abbreviated form of *NX superblock*.

NX_MAX_FILE_SYSTEMS

The maximum number of volumes that can be in a single container.

```
#define NX_MAX_FILE_SYSTEMS 100
```

NX_EPH_INFO_COUNT

The length of the array in the nx_ephemeral_info field.

```
#define NX_EPH_INFO_COUNT 4
```


NX_EPH_MIN_BLOCK_COUNT

The default minimum size, in blocks, for structures that contain ephemeral data.

```
#define NX_EPH_MIN_BLOCK_COUNT 8
```

This value is used when choosing the size for a new container's checkpoint data area, and the value used is recorded in the `nx_ephemeral_info` field.

NX_MAX_FILE_SYSTEM_EPH_STRUCTS

The number of structures that contain ephemeral data that a volume can have.

```
#define NX_MAX_FILE_SYSTEM_EPH_STRUCTS   4
```

This value is used when choosing the size for a new container's checkpoint data area, and the value used is recorded in the `nx_ephemeral_info` field.

NX_TX_MIN_CHECKPOINT_COUNT

The minimum number of checkpoints that can fit in the checkpoint data area.

```
#define NX_TX_MIN_CHECKPOINT_COUNT       4
```

This value is used when choosing the size for a new container's checkpoint data area.

NX_EPH_INFO_VERSION_1

The version number for structures that contain ephemeral data.

```
#define NX_EPH_INFO_VERSION_1 1
```

This value is recorded in the `nx_ephemeral_info` field.

Container Flags

The flags used for general information about a container.

```
#define NX_RESERVED_1                0x00000001LL
#define NX_RESERVED_2                0x00000002LL
#define NX_CRYPTO_SW                 0x00000004LL
```

These flags are used by the `nx_flags` field of `nx_superblock_t`.

NX_RESERVED_1

Reserved.

```
#define NX_RESERVED_1 0x00000001LL
```

Don't set this flag, but preserve it if it's already set.

NX_RESERVED_2

Reserved.

```
#define NX_RESERVED_2 0x00000002LL
```

Don't add this flag to a container. If this flag is set, preserve it when reading the container, and remove it when modifying the container.

NX_CRYPTO_SW

The container uses software cryptography.

```
#define NX_CRYPTO_SW 0x00000004LL
```

If this flag is set, the `crypto_id` field on all instances of `j_file_extent_val_t` has a value of `CRYPTO_SW_ID`.

Note that a container that has no volumes never has this flag set, regardless of whether the container will use software cryptography for new volumes. If you are creating a new volume in this scenario, determine whether to use software or hardware cryptography by consulting the I/O Registry as discussed in IOKit Fundamentals.

Optional Container Feature Flags

The flags used to describe optional features of an Apple File System container.

```
#define NX_FEATURE_DEFRAG             0x0000000000000001ULL
#define NX_FEATURE_LCFD               0x0000000000000002ULL
#define NX_SUPPORTED_FEATURES_MASK  (NX_FEATURE_DEFRAG | NX_FEATURE_LCFD)
```

These flags are used by the `nx_features` field of `nx_superblock_t`.

NX_FEATURE_DEFRAG

The volumes in this container support defragmentation.

```
#define NX_FEATURE_DEFRAG 0x0000000000000001ULL
```

NX_FEATURE_LCFD

This container is using low-capacity Fusion Drive mode.

```
#define NX_FEATURE_LCFD 0x0000000000000002ULL
```

Low-capacity Fusion Drive mode is enabled when the solid-state drive has a smaller capacity and so the cache must be smaller.

NX_SUPPORTED_FEATURES_MASK

A bit mask of all the optional features.

```
#define NX_SUPPORTED_FEATURES_MASK (NX_FEATURE_DEFRAG | NX_FEATURE_LCFD)
```

Read-Only Compatible Container Feature Flags

The flags used to describe read-only compatible features of an Apple File System container.

```
#define NX_SUPPORTED_ROCOMPAT_MASK (0x0ULL)
```

These flags are used by the `nx_readonly_compatible_features` field of `nx_superblock_t`. There are currently none defined.

NX_SUPPORTED_ROCOMPAT_MASK

A bit mask of all read-only compatible features.

```
#define NX_SUPPORTED_ROCOMPAT_MASK (0x0ULL)
```

Incompatible Container Feature Flags

The flags used to describe backward-incompatible features of an Apple File System container.

```
#define NX_INCOMPAT_VERSION1        0x0000000000000001ULL
#define NX_INCOMPAT_VERSION2        0x0000000000000002ULL
#define NX_INCOMPAT_FUSION          0x0000000000000100ULL
#define NX_SUPPORTED_INCOMPAT_MASK  (NX_INCOMPAT_VERSION2 | NX_INCOMPAT_FUSION)
```

These flags are used by the `nx_incompatible_features` field of `nx_superblock_t`.

NX_INCOMPAT_VERSION1

The container uses version 1 of Apple File System, as implemented in macOS 10.12.

```
#define NX_INCOMPAT_VERSION1 0x0000000000000001ULL
```

> **Important**
>
> Version 1 of the Apple File System was a prerelease that's incompatible with later versions. This document describes only version 2 and later.

NX_INCOMPAT_VERSION2

The container uses version 2 of Apple File System, as implemented in macOS 10.13 and iOS 10.3.

```
#define NX_INCOMPAT_VERSION2 0x0000000000000002ULL
```

NX_INCOMPAT_FUSION

The container supports Fusion Drives.

```
#define NX_INCOMPAT_FUSION 0x0000000000000100ULL
```

NX_SUPPORTED_INCOMPAT_MASK

A bit mask of all the backward-incompatible features.

```
#define NX_SUPPORTED_INCOMPAT_MASK (NX_INCOMPAT_VERSION2 | NX_INCOMPAT_FUSION)
```

Block and Container Sizes

Constants used when choosing the size of a block or container.

The block size for a container is defined by the `nx_block_size` field of `nx_superblock_t`.

```
#define NX_MINIMUM_BLOCK_SIZE       4096
#define NX_DEFAULT_BLOCK_SIZE       4096
#define NX_MAXIMUM_BLOCK_SIZE       65536

#define NX_MINIMUM_CONTAINER_SIZE   1048576
```

NX_MINIMUM_BLOCK_SIZE

The smallest supported size, in bytes, for a block.

```
#define NX_MINIMUM_BLOCK_SIZE 4096
```

If you try to define a block size that's too small, some data structures won't be able to fit in a single block.

NX_DEFAULT_BLOCK_SIZE

The default size, in bytes, for a block.

```
#define NX_DEFAULT_BLOCK_SIZE 4096
```

NX_MAXIMUM_BLOCK_SIZE

The largest supported size, in bytes, for a block.

```
#define NX_MAXIMUM_BLOCK_SIZE 65536
```

If you try to define a block size that's too large, parts of the block will be outside of the range of a 16-bit address.

NX_MINIMUM_CONTAINER_SIZE

The smallest supported size, in bytes, for a container.

```
#define NX_MINIMUM_CONTAINER_SIZE 1048576
```

This value is slightly less that the capacity of a floppy disk. For a container this size, statically allocated metadata takes up about a third of the available space.

nx_counter_id_t

Indexes into a container superblock's array of counters.

```
typedef enum {
    NX_CNTR_OBJ_CKSUM_SET  = 0,
    NX_CNTR_OBJ_CKSUM_FAIL = 1,

    NX_NUM_COUNTERS = 32
} nx_counter_id_t;
```

These values are used as indexes into the array stored in the `nx_counters` field of `nx_superblock_t`.

NX_CNTR_OBJ_CKSUM_SET

The number of times a checksum has been computed while writing objects to disk.

```
NX_CNTR_OBJ_CKSUM_SET  = 0
```

NX_CNTR_OBJ_CKSUM_FAIL

The number of times an object's checksum was invalid when reading from disk.

```
NX_CNTR_OBJ_CKSUM_FAIL = 1
```

NX_NUM_COUNTERS

The maximum number of counters.

```
NX_NUM_COUNTERS = 32
```

checkpoint_mapping_t

A mapping from an ephemeral object identifier to its physical address in the checkpoint data area.

```
struct checkpoint_mapping {
    uint32_t    cpm_type;
    uint32_t    cpm_subtype;
    uint32_t    cpm_size;
    uint32_t    cpm_pad;
    oid_t       cpm_fs_oid;
    oid_t       cpm_oid;
    oid_t       cpm_paddr;
};
typedef struct checkpoint_mapping checkpoint_mapping_t;
```

cpm_type

The object's type.

```
uint32_t cpm_type;
```

An object type is a 32-bit value: The low 16 bits indicate the type using the values listed in Object Types, and the high 16 bits are flags using the values listed in Object Type Flags.

This field has the same meaning and behavior as the `o_type` field of `obj_phys_t`.

cpm_subtype

The object's subtype.

```
uint32_t cpm_subtype;
```

One of the values listed in Object Types.

Subtypes indicate the type of data stored in a data structure such as a B-tree. For example, a leaf node in a B-tree that contains file-system records has a type of `OBJECT_TYPE_BTREE_NODE` and a subtype of `OBJECT_TYPE_FSTREE`.

This field has the same meaning and behavior as the `o_subtype` field of `obj_phys_t`.

cpm_size

The size, in bytes, of the object.

```
uint32_t cpm_size;
```

cpm_pad

Reserved.

```
uint32_t cpm_pad;
```

Populate this field with zero when you create a new mapping, and preserve its value when you modify an existing mapping.

This field is padding.

cpm_fs_oid

The virtual object identifier of the volume that the object is associated with.

```
oid_t cpm_fs_oid;
```

cpm_oid

The ephemeral object identifier.

```
oid_t cpm_oid;
```

cpm_paddr

The address in the checkpoint data area where the object is stored.

```
oid_t cpm_paddr;
```

checkpoint_map_phys_t

A checkpoint-mapping block.

```
struct checkpoint_map_phys {
    obj_phys_t              cpm_o;
    uint32_t                cpm_flags;
```

```
    uint32_t                cpm_count;
    checkpoint_mapping_t    cpm_map[];
};
```

If a checkpoint needs to store more mappings than a single block can hold, the checkpoint has multiple checkpoint-mapping blocks stored contiguously in the checkpoint descriptor area. The last checkpoint-mapping block is marked with the `CHECKPOINT_MAP_LAST` flag.

`cpm_o`

The object's header.

```
obj_phys_t cpm_o;
```

`cpm_flags`

A bit field that contains additional information about the list of checkpoint mappings.

```
uint32_t cpm_flags;
```

For the values used in this bit field, see Checkpoint Flags.

`cpm_count`

The number of checkpoint mappings in the array.

```
uint32_t cpm_count;
```

`cpm_map`

The array of checkpoint mappings.

```
checkpoint_mapping_t cpm_map[];
```

Checkpoint Flags

The flags used by a checkpoint-mapping block.

```
#define CHECKPOINT_MAP_LAST             0x00000001
```

`CHECKPOINT_MAP_LAST`

A flag marking the last checkpoint-mapping block in a given checkpoint.

```
#define CHECKPOINT_MAP_LAST 0x00000001
```

`evict_mapping_val_t`

A range of physical addresses that data is being moved into.

```
struct evict_mapping_val {
    paddr_t     dst_paddr;
    uint64_t    len;
} __attribute__((packed));
```

```
typedef struct evict_mapping_val evict_mapping_val_t;
```

This data type is used by the evict-mapping tree, which is accessed through the `nx_evict_mapping_tree_oid` field of `nx_superblock_t`.

dst_paddr

The address where the destination starts.

```
paddr_t dst_paddr;
```

len

The number of blocks being moved.

```
uint64_t len;
```

Object Maps

An object map uses a B-tree to store a mapping from virtual object identifiers and transaction identifiers to the physical addresses where those objects are stored. The keys in the B-tree are instances of `omap_key_t` and the values are instances of `paddr_t`.

To access a virtual object using the object map, perform the following operations:

1. Determine which object map to use. Objects that are within a volume use that volume's object map, and all other objects use the container's object map.
2. Locate the object map for the volume by reading the `apfs_omap_oid` field of `apfs_superblock_t` or the `nx_omap_oid` field of `nx_superblock_t`.
3. Locate the B-tree for the object map by reading the `om_tree_oid` field of `omap_phys_t`.
4. Search the B-tree for a key whose object identifier is the same as the desired object identifier, and whose transaction identifier is less than or equal to the desired transaction identifier. If there are multiple keys that satisfy this test, use the key with the largest transaction identifier.
5. Using the table of contents entry, read the corresponding value for the key you found, which contains a physical address.
6. Read the object from disk at that address.

For example, assume the object map's B-tree contains the following mappings:

```
OID 588, XID 2101 -> Address 200
OID 588, XID 2202 -> Address 300
OID 588, XID 2300 -> Address 100
```

To access object 588 as of transaction 2300, you use the last entry — its object and transaction identifiers match exactly — and read physical address 100.

To access object 588 as of transaction 2290, you use the second entry. There's no entry with the transaction identifier 2290, and 2202 is the largest transaction identifier in the object map that's still less than 2290. That entry tells you to read physical address 300.

omap_phys_t

An object map.

```
struct omap_phys {
    obj_phys_t  om_o;
    uint32_t    om_flags;
    uint32_t    om_snap_count;
    uint32_t    om_tree_type;
    uint32_t    om_snapshot_tree_type;
    oid_t       om_tree_oid;
    oid_t       om_snapshot_tree_oid;
    xid_t       om_most_recent_snap;
    xid_t       om_pending_revert_min;
```

```
    xid_t       om_pending_revert_max;
};
typedef struct omap_phys omap_phys_t;
```

om_o

The object's header.

```
obj_phys_t om_o;
```

om_flags

The object map's flags.

```
uint32_t om_flags;
```

For the values used in this bit field, see Object Map Flags.

om_tree_type

The type of tree being used for object mappings.

```
uint32_t om_tree_type;
```

om_tree_oid

The virtual object identifier of the tree being used for object mappings.

```
oid_t om_tree_oid;
```

om_snapshot_tree_oid

The virtual object identifier of the tree being used to hold snapshot information.

```
oid_t om_snapshot_tree_oid;
```

om_snapshot_tree_type

The type of tree being used for snapshots.

```
uint32_t om_snapshot_tree_type;
```

om_snap_count

The number of snapshots that this object map has.

```
uint32_t om_snap_count;
```

om_most_recent_snap

The transaction identifier of the most recent snapshot that's stored in this object map.

```
xid_t om_most_recent_snap;
```

om_pending_revert_min

The smallest transaction identifier for an in-progress revert.

```
xid_t om_pending_revert_min;
```

om_pending_revert_max

The largest transaction identifier for an in-progress revert.

```
xid_t om_pending_revert_max;
```

omap_key_t

A key used to access an entry in the object map.

```
struct omap_key {
    oid_t       ok_oid;
    xid_t       ok_xid;
};
typedef struct omap_key omap_key_t;
```

ok_oid

The object identifier.

```
oid_t ok_oid;
```

ok_xid

The transaction identifier.

```
xid_t ok_xid;
```

omap_val_t

A value in the object map.

```
struct omap_val {
    uint32_t    ov_flags;
    uint32_t    ov_size;
    paddr_t     ov_paddr;
};
typedef struct omap_val omap_val_t;
```

ov_flags

A bit field of flags.

```
uint32_t ov_flags;
```

For the values used in this bit field, see Object Map Value Flags.

ov_size

The size, in bytes, of the object.

```
uint32_t ov_size;
```

This value must be a multiple of the container's logical block size. If the object is smaller than one logical block, the value of this field is the size of one logical block.

ov_paddr

The address of the object.

```
paddr_t ov_paddr;
```

omap_snapshot_t

Information about a snapshot of an object map.

```
struct omap_snapshot {
    uint32_t    oms_flags;
    uint32_t    oms_pad;
    oid_t       oms_oid;
};
typedef struct omap_snapshot omap_snapshot_t;
```

When accessing or storing a snapshot in the snapshot tree, use the transaction identifier as the key. This structure is the value stored in a snapshot tree.

oms_flags

The snapshot's flags.

```
uint32_t oms_flags;
```

For the values used in this bit field, see Snapshot Flags.

oms_pad

Reserved.

```
uint32_t oms_pad;
```

Populate this field with zero when you create a new snapshot, and preserve its value when you modify an existing snapshot.

This field is padding.

oms_oid

Reserved.

```
oid_t oms_oid;
```

Populate this field with zero when you create a new snapshot, and preserve its value when you modify an existing snapshot.

Object Map Value Flags

The flags used by entries in the object map.

```
#define OMAP_VAL_DELETED            0x00000001
#define OMAP_VAL_SAVED              0x00000002
#define OMAP_VAL_ENCRYPTED          0x00000004
#define OMAP_VAL_NOHEADER           0x00000008
#define OMAP_VAL_CRYPTO_GENERATION  0x00000010
```

`OMAP_VAL_DELETED`

The object has been deleted, and this mapping is a placeholder.

```
#define OMAP_VAL_DELETED 0x00000001
```

`OMAP_VAL_SAVED`

This object mapping shouldn't be replaced when the object is updated.

```
#define OMAP_VAL_SAVED 0x00000002
```

This flag is used only on mappings in an object map that's manually managed. In the current Apple implementation, it's never used.

See also the `OMAP_MANUALLY_MANAGED` flag.

`OMAP_VAL_ENCRYPTED`

The object is encrypted.

```
#define OMAP_VAL_ENCRYPTED 0x00000004
```

`OMAP_VAL_NOHEADER`

The object is stored without an `obj_phys_t` header.

```
#define OMAP_VAL_NOHEADER 0x00000008
```

`OMAP_VAL_CRYPTO_GENERATION`

A one-bit flag that tracks encryption configuration.

```
#define OMAP_VAL_CRYPTO_GENERATION 0x00000010
```

During the transition from an old encryption configuration to a new one, not all objects have been reencrypted using the new configuration. When the encryption configuration is changed, the object map's flag is toggled. After an object is reencrypted, the object's flag is also toggled.

If this flag doesn't match the flag on the object map, the encryption configuration has changed, but the object hasn't been reencrypted yet. Use the previous encryption configuration to decrypt the object.

See also OMAP_CRYPTO_GENERATION, which is used by the omap_phys_t field of om_flags.

Snapshot Flags

The flags used to describe the state of a snapshot.

```
#define OMAP_SNAPSHOT_DELETED       0x00000001
#define OMAP_SNAPSHOT_REVERTED      0x00000002
```

OMAP_SNAPSHOT_DELETED

The snapshot has been deleted.

```
#define OMAP_SNAPSHOT_DELETED 0x00000001
```

OMAP_SNAPSHOT_REVERTED

The snapshot has been deleted as part of a revert.

```
#define OMAP_SNAPSHOT_REVERTED 0x00000002
```

Object Map Flags

The flags used by object maps.

```
#define OMAP_MANUALLY_MANAGED       0x00000001
#define OMAP_ENCRYPTING             0x00000002
#define OMAP_DECRYPTING             0x00000004
#define OMAP_KEYROLLING             0x00000008
#define OMAP_CRYPTO_GENERATION      0x00000010

#define OMAP_VALID_FLAGS            0x0000001f
```

OMAP_MANUALLY_MANAGED

The object map doesn't support snapshots.

```
#define OMAP_MANUALLY_MANAGED 0x00000001
```

This flag must be set on the container's object map and is invalid on a volume's object map.

OMAP_ENCRYPTING

A transition is in progress from unencrypted storage to encrypted storage.

```
#define OMAP_ENCRYPTING 0x00000002
```

OMAP_DECRYPTING

A transition is in progress from encrypted storage to unencrypted storage.

```
#define OMAP_DECRYPTING 0x00000004
```

OMAP_KEYROLLING

A transition is in progress from encrypted storage using an old key to encrypted storage using a new key.

```
#define OMAP_KEYROLLING 0x00000008
```

OMAP_CRYPTO_GENERATION

A one-bit flag that tracks encryption configuration.

```
#define OMAP_CRYPTO_GENERATION 0x00000010
```

For information about how this flag is used to track the old and new encryption configuration, see OMAP_VAL_CRYPTO_GENERATION, which is used by the ov_flags field of omap_val_t.

OMAP_VALID_FLAGS

A bit mask of all valid object map flags.

```
#define OMAP_VALID_FLAGS 0x0000001f
```

Object Map Constants

Constants that specify size constraints of an object map.

```
#define OMAP_MAX_SNAP_COUNT         UINT32_MAX
```

OMAP_MAX_SNAP_COUNT

The maximum number of snapshots that can be stored in an object map.

```
#define OMAP_MAX_SNAP_COUNT UINT32_MAX
```

Object Map Reaper Phases

Phases used by the reaper when deleting objects that are stored in an object map.

```
#define OMAP_REAP_PHASE_MAP_TREE        1
#define OMAP_REAP_PHASE_SNAPSHOT_TREE   2
```

OMAP_REAP_PHASE_MAP_TREE

The reaper is deleting entries from the object mapping tree.

```
#define OMAP_REAP_PHASE_MAP_TREE 1
```

OMAP_REAP_PHASE_SNAPSHOT_TREE

The reaper is deleting entries from the snapshot tree.

```
#define OMAP_REAP_PHASE_SNAPSHOT_TREE 2
```

Volumes

A volume contains a file system, the files and metadata that make up that file system, and various supporting data structures like an object map.

apfs_superblock_t

A volume superblock.

```
struct apfs_superblock {
    obj_phys_t                  apfs_o;

    uint32_t                    apfs_magic;
    uint32_t                    apfs_fs_index;

    uint64_t                    apfs_features;
    uint64_t                    apfs_readonly_compatible_features;
    uint64_t                    apfs_incompatible_features;

    uint64_t                    apfs_unmount_time;

    uint64_t                    apfs_fs_reserve_block_count;
    uint64_t                    apfs_fs_quota_block_count;
    uint64_t                    apfs_fs_alloc_count;

    wrapped_meta_crypto_state_t apfs_meta_crypto;

    uint32_t                    apfs_root_tree_type;
    uint32_t                    apfs_extentref_tree_type;
    uint32_t                    apfs_snap_meta_tree_type;

    oid_t                       apfs_omap_oid;
    oid_t                       apfs_root_tree_oid;
    oid_t                       apfs_extentref_tree_oid;
    oid_t                       apfs_snap_meta_tree_oid;

    xid_t                       apfs_revert_to_xid;
    oid_t                       apfs_revert_to_sblock_oid;

    uint64_t                    apfs_next_obj_id;

    uint64_t                    apfs_num_files;
    uint64_t                    apfs_num_directories;
    uint64_t                    apfs_num_symlinks;
    uint64_t                    apfs_num_other_fsobjects;
    uint64_t                    apfs_num_snapshots;
```

```
    uint64_t                    apfs_total_blocks_alloced;
    uint64_t                    apfs_total_blocks_freed;

    uuid_t                      apfs_vol_uuid;
    uint64_t                    apfs_last_mod_time;

    uint64_t                    apfs_fs_flags;

    apfs_modified_by_t          apfs_formatted_by;
    apfs_modified_by_t          apfs_modified_by[APFS_MAX_HIST];

    uint8_t                     apfs_volname[APFS_VOLNAME_LEN];
    uint32_t                    apfs_next_doc_id;

    uint16_t                    apfs_role;
    uint16_t                    reserved;

    xid_t                       apfs_root_to_xid;
    oid_t                       apfs_er_state_oid;

    uint64_t                    apfs_cloneinfo_id_epoch;
    uint64_t                    apfs_cloneinfo_xid;

    oid_t                       apfs_snap_meta_ext_oid;

    uuid_t                      apfs_volume_group_id;

    oid_t                       apfs_integrity_meta_oid;

    oid_t                       apfs_fext_tree_oid;
    uint32_t                    apfs_fext_tree_type;

    uint32_t                    reserved_type;
    oid_t                       reserved_oid;
};

#define APFS_MAGIC                  'BSPA'
#define APFS_MAX_HIST               8
#define APFS_VOLNAME_LEN            256
```

apfs_o

The object's header.

```
obj_phys_t apfs_o;
```

apfs_magic

A number that can be used to verify that you're reading an instance of `apfs_superblock_t`.

```
uint32_t apfs_magic;
```

The value of this field is always APFS_MAGIC.

apfs_fs_index

The index of the object identifier for this volume's file system in the container's array of file systems.

```
uint32_t apfs_fs_index
```

The container's array is stored in the nx_fs_oid field of nx_superblock_t.

When a volume is being deleted, it's removed from the container's array of volumes before apfs_superblock_t object is destroyed. If you read this field of a volume that's being deleted, the specified entry in the array might have already been reused for another volume.

apfs_features

A bit field of the optional features being used by this volume.

```
uint64_t apfs_features;
```

For the values used in this bit field, see Optional Volume Feature Flags.

If your implementation doesn't support an optional feature that's in use, ignore that feature in this list and mount the volume as usual.

apfs_readonly_compatible_features

A bit field of the read-only compatible features being used by this volume.

```
uint64_t apfs_readonly_compatible_features;
```

For the values used in this bit field, see Read-Only Compatible Volume Feature Flags.

If your implementation doesn't support a read-only compatible feature that's in use, mount the volume as read-only.

apfs_incompatible_features

A bit field of the backward-incompatible features being used by this volume.

```
uint64_t apfs_incompatible_features;
```

For the values used in this bit field, see Incompatible Volume Feature Flags.

If your implementation doesn't support a backward-incompatible feature that's in use, it must not mount the volume.

apfs_unmount_time

The time that this volume was last unmounted.

```
uint64_t apfs_unmount_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

apfs_fs_reserve_block_count

The number of blocks that have been reserved for this volume to allocate.

```
uint64_t apfs_fs_reserve_block_count;
```

apfs_fs_quota_block_count

The maximum number of blocks that this volume can allocate.

```
uint64_t apfs_fs_quota_block_count;
```

apfs_fs_alloc_count

The number of blocks currently allocated for this volume's file system.

```
uint64_t apfs_fs_alloc_count;
```

apfs_meta_crypto

Information about the key used to encrypt metadata for this volume.

```
wrapped_meta_crypto_state_t apfs_meta_crypto;
```

On devices running macOS, the volume encryption key (VEK) is used to encrypt the metadata, as discussed in Accessing Encrypted Objects.

apfs_root_tree_type

The type of the root file-system tree.

```
uint32_t apfs_root_tree_type
```

The value is typically `OBJ_VIRTUAL | OBJECT_TYPE_BTREE`, with a subtype of `OBJECT_TYPE_FSTREE`. For possible values, see Object Types.

apfs_extentref_tree_type

The type of the extent-reference tree.

```
uint32_t apfs_extentref_tree_type
```

The value is typically `OBJ_PHYSICAL | OBJECT_TYPE_BTREE`, with a subtype of `OBJECT_TYPE_BLOCKREF`. For possible values, see Object Types.

apfs_snap_meta_tree_type

The type of the snapshot metadata tree.

```
uint32_t apfs_snap_meta_tree_type
```

The value is typically `OBJ_PHYSICAL | OBJECT_TYPE_BTREE`, with a subtype of `OBJECT_TYPE_BLOCKREF`. For possible values, see Object Types.

apfs_omap_oid

The physical object identifier of the volume's object map.

```
oid_t apfs_omap_oid;
```

apfs_root_tree_oid

The virtual object identifier of the root file-system tree.

```
oid_t apfs_root_tree_oid;
```

apfs_extentref_tree_oid

The physical object identifier of the extent-reference tree.

```
oid_t apfs_extentref_tree_oid;
```

When a snapshot is created, the current extent-reference tree is moved to the snapshot. A new, empty, extent-reference tree is created and its object identifier becomes the new value of this field.

apfs_snap_meta_tree_oid

The virtual object identifier of the snapshot metadata tree.

```
oid_t apfs_snap_meta_tree_oid;
```

apfs_revert_to_xid

The transaction identifier of a snapshot that the volume will revert to.

```
xid_t apfs_revert_to_xid;
```

When mounting a volume, if the value of this field nonzero, revert to the specified snapshot by deleting all snapshots after the specified transaction identifier and deleting the current state, and then setting this field to zero.

apfs_revert_to_sblock_oid

The physical object identifier of a volume superblock that the volume will revert to.

```
oid_t apfs_revert_to_sblock_oid;
```

When mounting a volume, if the `apfs_revert_to_xid` field is nonzero, ignore the value of this field. Otherwise, revert to the specified volume superblock.

apfs_next_obj_id

The next identifier that will be assigned to a file-system object in this volume.

```
uint64_t apfs_next_obj_id;
```

apfs_num_files

The number of regular files in this volume.

```
uint64_t apfs_num_files;
```

apfs_num_directories

The number of directories in this volume.

```
uint64_t apfs_num_directories;
```

apfs_num_symlinks

The number of symbolic links in this volume.

```
uint64_t apfs_num_symlinks;
```

apfs_num_other_fsobjects

The number of other files in this volume.

```
uint64_t apfs_num_other_fsobjects;
```

The value of this field includes all files that aren't included in the `apfs_num_symlinks`, `apfs_num_directories`, or `apfs_num_files` fields.

apfs_num_snapshots

The number of snapshots in this volume.

```
uint64_t apfs_num_snapshots;
```

apfs_total_blocks_alloced

The total number of blocks that have been allocated by this volume.

```
uint64_t apfs_total_blocks_alloced;
```

The value of this field increases when blocks are allocated, but isn't modified when they're freed. If the volume doesn't contain any files, the value of this field matches `apfs_total_blocks_freed`.

apfs_total_blocks_freed

The total number of blocks that have been freed by this volume.

```
uint64_t apfs_total_blocks_freed;
```

The value of this field isn't modified when blocks are allocated, but increases when they're freed. If the volume doesn't contain any files, the value of this field matches `apfs_total_blocks_alloced`.

apfs_vol_uuid

The universally unique identifier for this volume.

```
uuid_t apfs_vol_uuid;
```

apfs_last_mod_time

The time that this volume was last modified.

```
uint64_t apfs_last_mod_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

apfs_fs_flags

The volume's flags.

```
uint64_t apfs_fs_flags;
```

For the values used in this bit field, see Volume Flags.

apfs_formatted_by

Information about the software that created this volume.

```
apfs_modified_by_t apfs_formatted_by;
```

This field is set only once, when the volume is created.

apfs_modified_by

Information about the software that has modified this volume.

```
apfs_modified_by_t apfs_modified_by[APFS_MAX_HIST]
```

The newest element in this array is stored at index zero. To update this field when you modify a volume, move each element to the index that's larger by one, and then write the new modification information. When you create a new volume, fill the array's memory with zeros.

If the implementation's information is already the last entry in this field, you can update the field as usual (creating a duplicate), or leave the field's value unmodified. Both behaviors are permitted.

apfs_volname

The name of the volume, represented as a null-terminated UTF-8 string.

```
uint8_t apfs_volname[APFS_VOLNAME_LEN]
```

The `APFS_INCOMPAT_NON_UTF8_FNAMES` flag has no effect on this field's value.

apfs_next_doc_id

The next document identifier that will be assigned.

```
uint32_t apfs_next_doc_id
```

A document's identifier is stored in the INO_EXT_TYPE_DOCUMENT_ID extended field of the inode.

After assigning a new document identifier, increment this field by one. Valid document identifiers are greater than MIN_DOC_ID and less than UINT32_MAX − 1. If a new document identifier isn't available, that's an unrecoverable error. Identifiers aren't allowed to restart from one or to be reused.

apfs_role

The role of this volume within the container.

```
uint16_t apfs_role
```

For possible values, see Volume Roles.

reserved

Reserved.

```
uint16_t reserved
```

Populate this field with zero when you create a new volume, and preserve its value when you modify an existing volume.

apfs_root_to_xid

The transaction identifier of the snapshot to root from, or zero to root normally.

```
xid_t apfs_root_to_xid;
```

apfs_er_state_oid

The current state of encryption or decryption for a drive that's being encrypted or decrypted, or zero if no encryption change is in progress.

```
oid_t apfs_er_state_oid;
```

apfs_cloneinfo_id_epoch

The largest object identifier used by this volume at the time INODE_WAS_EVER_CLONED started storing valid information.

```
uint64_t apfs_cloneinfo_id_epoch;
```

If the value of this field is zero, all information stored using INODE_WAS_EVER_CLONED is valid. For information about how to this identifier is used, see INODE_WAS_EVER_CLONED.

This field was added to this data structure for macOS 10.13.3. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure. Because zero is a valid value for this field, check the value of apfs_cloneinfo_xid – if that field is also zero, the structure was created by an older implementation.

apfs_cloneinfo_xid

A transaction identifier used with apfs_cloneinfo_id_epoch.

```
uint64_t apfs_cloneinfo_xid;
```

When unmounting a volume, the value of this field is set to the latest transaction identifier, the same as the apfs_modified_by field. For information about how to this identifier is used, see INODE_WAS_EVER_CLONED.

This field was added to this data structure for macOS 10.13.3. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure.

apfs_snap_meta_ext_oid

The virtual object identifier of the extended snapshot metadata object.

```
oid_t apfs_snap_meta_ext_oid;
```

This field was added to this data structure for macOS 10.15. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure.

apfs_volume_group_id

The volume group the volume belongs to.

```
uuid_t apfs_volume_group_id;
```

If the volume doesn't belong to a volume group, the value of this field is zero and the APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE flag must not be set. Otherwise, the APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE flag must be set and this field must have a nonzero value.

This field was added to this data structure for macOS 10.15. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure.

apfs_integrity_meta_oid

The virtual object identifier of the integrity metadata object.

```
oid_t apfs_integrity_meta_oid;
```

If the value of this field is nonzero, the APFS_INCOMPAT_SEALED_VOLUME flag must also be set.

This field was added to this data structure for macOS 11. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure.

apfs_fext_tree_oid

The virtual object identifier of the file extent tree.

```
oid_t apfs_fext_tree_oid;
```

If the value of this field is nonzero, the APFS_INCOMPAT_SEALED_VOLUME flag must also be set.

This field was added to this data structure for macOS 11. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure.

apfs_fext_tree_type

The type of the file extent tree.

```
uint32_t apfs_fext_tree_type;
```

The value is typically `OBJ_PHYSICAL | OBJECT_TYPE_BTREE`, with a subtype of `OBJECT_TYPE_FEXT_TREE`. For possible values, see Object Types.

This field was added to this data structure for macOS 11. Older implementations of Apple File System store zero in this field when initializing an instance of the structure, and they preserve the field's value when modifying the structure.

reserved_type

Reserved.

```
uint32_t reserved_type;
```

reserved_oid

Reserved.

```
oid_t reserved_oid;
```

APFS_MAGIC

The value of the `apfs_magic` field.

```
#define APFS_MAGIC 'BSPA'
```

This magic number was chosen because in hex dumps it appears as "APSB", which is an abbreviated form of *APFS superblock*.

APFS_MAX_HIST

The number of entries stored in the `apfs_modified_by` field.

```
#define APFS_MAX_HIST 8
```

APFS_VOLNAME_LEN

The maximum length of the volume name stored in the `apfs_volname` field.

```
#define APFS_VOLNAME_LEN 256
```

apfs_modified_by_t

Information about a program that modified the volume.

```
struct apfs_modified_by {
    uint8_t     id[APFS_MODIFIED_NAMELEN];
    uint64_t    timestamp;
    xid_t       last_xid;
};
```

```
typedef struct apfs_modified_by apfs_modified_by_t;

#define APFS_MODIFIED_NAMELEN       32
```

This structure is used by the `apfs_modified_by` and `apfs_formatted_by` fields of `apfs_superblock_t`.

`id`

A string that identifies the program and its version.

```
uint8_t id[APFS_MODIFIED_NAMELEN];
```

`timestamp`

The time that the program last modified this volume.

```
uint64_t timestamp;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

`last_xid`

The last transaction identifier that's part of this program's modifications.

```
xid_t last_xid;
```

Volume Flags

The flags used to indicate volume status.

```
#define APFS_FS_UNENCRYPTED             0x00000001LL
#define APFS_FS_RESERVED_2              0x00000002LL
#define APFS_FS_RESERVED_4              0x00000004LL
#define APFS_FS_ONEKEY                  0x00000008LL
#define APFS_FS_SPILLEDOVER             0x00000010LL
#define APFS_FS_RUN_SPILLOVER_CLEANER   0x00000020LL
#define APFS_FS_ALWAYS_CHECK_EXTENTREF  0x00000040LL
#define APFS_FS_RESERVED_80             0x00000080LL
#define APFS_FS_RESERVED_100            0x00000100LL

#define APFS_FS_FLAGS_VALID_MASK        (APFS_FS_UNENCRYPTED \
                                       | APFS_FS_RESERVED_2 \
                                       | APFS_FS_RESERVED_4 \
                                       | APFS_FS_ONEKEY \
                                       | APFS_FS_SPILLEDOVER \
                                       | APFS_FS_RUN_SPILLOVER_CLEANER \
                                       | APFS_FS_ALWAYS_CHECK_EXTENTREF \
                                       | APFS_FS_RESERVED_80 \
                                       | APFS_FS_RESERVED_100)

#define APFS_FS_CRYPTOFLAGS             (APFS_FS_UNENCRYPTED \
```

```
                                        | APFS_FS_ONEKEY)
```

APFS_FS_UNENCRYPTED

The volume isn't encrypted.

```
#define APFS_FS_UNENCRYPTED 0x00000001LL
```

APFS_FS_RESERVED_2

Reserved.

```
#define APFS_FS_RESERVED_2 0x00000002LL
```

Don't set this flag, but preserve it if it's already set.

APFS_FS_RESERVED_4

Reserved.

```
#define APFS_FS_RESERVED_4 0x00000004LL
```

Don't set this flag, but preserve it if it's already set.

APFS_FS_ONEKEY

Files on the volume are all encrypted using the volume encryption key (VEK).

```
#define APFS_FS_ONEKEY 0x00000008LL
```

This flag is used only on devices running macOS; devices running iOS always use per-file encryption keys. When this flag is set, several encryption-related data structures store different information, as discussed in Accessing Encrypted Objects.

APFS_FS_SPILLEDOVER

The volume has run out of allocated space on the solid-state drive.

```
#define APFS_FS_SPILLEDOVER 0x00000010LL
```

See also `INODE_ALLOCATION_SPILLEDOVER`.

APFS_FS_RUN_SPILLOVER_CLEANER

The volume has spilled over and the spillover cleaner must be run.

```
#define APFS_FS_RUN_SPILLOVER_CLEANER 0x00000020LL
```

APFS_FS_ALWAYS_CHECK_EXTENTREF

The volume's extent reference tree is always consulted when deciding whether to overwrite an extent.

```
#define APFS_FS_ALWAYS_CHECK_EXTENTREF 0x00000040LL
```

APFS_FS_RESERVED_80

Reserved.

```
#define APFS_FS_RESERVED_80 0x00000080LL
```

APFS_FS_RESERVED_100

Reserved.

```
#define APFS_FS_RESERVED_100 0x00000100LL
```

APFS_FS_FLAGS_VALID_MASK

A bit mask of all volume flags.

```
#define APFS_FS_FLAGS_VALID_MASK (APFS_FS_UNENCRYPTED \
                               | APFS_FS_RESERVED_2 \
                               | APFS_FS_RESERVED_4 \
                               | APFS_FS_ONEKEY \
                               | APFS_FS_RUN_SPILLOVER_CLEANER \
                               | APFS_FS_ALWAYS_CHECK_EXTENTREF)
```

APFS_FS_CRYPTOFLAGS

A bit mask of all encryption-related volume flags.

```
#define APFS_FS_CRYPTOFLAGS (APFS_FS_UNENCRYPTED \
                          | APFS_FS_RESERVED_2 \
                          | APFS_FS_ONEKEY)
```

Volume Roles

The values used to indicate a volume's roles.

```
#define APFS_VOL_ROLE_NONE          0x0000

#define APFS_VOL_ROLE_SYSTEM        0x0001
#define APFS_VOL_ROLE_USER          0x0002
#define APFS_VOL_ROLE_RECOVERY      0x0004
#define APFS_VOL_ROLE_VM            0x0008
#define APFS_VOL_ROLE_PREBOOT       0x0010
#define APFS_VOL_ROLE_INSTALLER     0x0020

#define APFS_VOL_ROLE_DATA          (1 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_BASEBAND      (2 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_UPDATE        (3 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_XART          (4 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_HARDWARE      (5 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_BACKUP        (6 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_RESERVED_7    (7 << APFS_VOLUME_ENUM_SHIFT)
```

```
#define APFS_VOL_ROLE_RESERVED_8     (8 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_ENTERPRISE     (9 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_RESERVED_10    (10 << APFS_VOLUME_ENUM_SHIFT)
#define APFS_VOL_ROLE_PRELOGIN       (11 << APFS_VOLUME_ENUM_SHIFT)

#define APFS_VOLUME_ENUM_SHIFT       6
```

These values are used by the `apfs_role` field of `apfs_superblock_t`. A volume has at most one role.

For historical reasons, the underlying values of these constants have two variations. The roles whose constants use only the six least significant bits and the `APFS_VOL_ROLE_DATA` and `APFS_VOL_ROLE_BASEBAND` roles are supported by all versions of macOS and iOS. The remaining roles that are stored using the ten most significant bits are supported only by devices running macOS 10.15, iOS 13, and later.

`APFS_VOL_ROLE_NONE`

The volume has no defined role.

```
#define APFS_VOL_ROLE_NONE 0x0000
```

A volume whose role doesn't have a constant defined doesn't have any flags set.

`APFS_VOL_ROLE_SYSTEM`

The volume contains a root directory for the system.

```
#define APFS_VOL_ROLE_SYSTEM 0x0001
```

The file system for the system volume that contains the running OS is normally mounted at `/`. On devices running iOS and macOS 10.15 or later, the system volume is mounted read-only.

See also `APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE`, which is used to mount the system and user data as a single user-visible volume.

`APFS_VOL_ROLE_USER`

The volume contains users' home directories.

```
#define APFS_VOL_ROLE_USER 0x0002
```

`APFS_VOL_ROLE_RECOVERY`

The volume contains a recovery system.

```
#define APFS_VOL_ROLE_RECOVERY 0x0004
```

This is used the same way as a recovery partition on HFS-Plus.

`APFS_VOL_ROLE_VM`

The volume is used as swap space for virtual memory.

```
#define APFS_VOL_ROLE_VM 0x0008
```

The file system for a virtual-memory volume is mounted at `/var/vm`.

APFS_VOL_ROLE_PREBOOT

The volume contains files needed to boot from an encrypted volume.

```
#define APFS_VOL_ROLE_PREBOOT 0x0010
```

APFS_VOL_ROLE_INSTALLER

The volume is used by the OS installer.

```
#define APFS_VOL_ROLE_INSTALLER 0x0020
```

For example, the installer writes log files to this volume during the installation process.

APFS_VOL_ROLE_DATA

The volume contains mutable data.

```
#define APFS_VOL_ROLE_DATA (1 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running iOS and macOS 10.15 or later. It contains both user data and mutable system data. Immutable system data is stored on the volume with the `APFS_VOL_ROLE_SYSTEM` flag.

See also `APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE`, which is used to mount the system and user data as a single user-visible volume.

APFS_VOL_ROLE_BASEBAND

The volume is used by the radio firmware.

```
#define APFS_VOL_ROLE_BASEBAND (2 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running iOS.

APFS_VOL_ROLE_UPDATE

The volume is used by the software update mechanism.

```
#define APFS_VOL_ROLE_UPDATE (3 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running iOS.

APFS_VOL_ROLE_XART

The volume is used to manage OS access to secure user data.

```
#define APFS_VOL_ROLE_XART (4 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running iOS.

APFS_VOL_ROLE_HARDWARE

The volume is used for firmware data.

```
#define APFS_VOL_ROLE_HARDWARE (5 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running iOS.

APFS_VOL_ROLE_BACKUP

The volume is used by Time Machine to store backups.

```
#define APFS_VOL_ROLE_BACKUP (6 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running macOS.

APFS_VOL_ROLE_RESERVED_7

Reserved.

```
#define APFS_VOL_ROLE_SIDECAR (7 << APFS_VOLUME_ENUM_SHIFT)
```

APFS_VOL_ROLE_RESERVED_8

Reserved.

```
#define APFS_VOL_ROLE_RESERVED_8 (8 << APFS_VOLUME_ENUM_SHIFT)
```

APFS_VOL_ROLE_ENTERPRISE

This volume is used to store enterprise-managed data.

```
#define APFS_VOL_ROLE_ENTERPRISE (9 << APFS_VOLUME_ENUM_SHIFT)
```

For more information, see Managing Devices & Corporate Data on iOS.

APFS_VOL_ROLE_RESERVED_10

Reserved.

```
#define APFS_VOL_ROLE_RESERVED_10 (10 << APFS_VOLUME_ENUM_SHIFT)
```

APFS_VOL_ROLE_PRELOGIN

This volume is used to store system data used before login.

```
#define APFS_VOL_ROLE_PRELOGIN (11 << APFS_VOLUME_ENUM_SHIFT)
```

This role is used only on devices running macOS. The prelogin volume lets the system boot to the login screen, at which point the user can log in and the user’s password can be used to mount encrypted volumes.

APFS_VOLUME_ENUM_SHIFT

The bit shift used to separate the old and new enumeration cases.

```
#define APFS_VOLUME_ENUM_SHIFT 6
```

Optional Volume Feature Flags

The flags used to describe optional features of an Apple File System volume.

```
#define APFS_FEATURE_DEFRAG_PRERELEASE          0x00000001LL
#define APFS_FEATURE_HARDLINK_MAP_RECORDS       0x00000002LL
#define APFS_FEATURE_DEFRAG                     0x00000004LL
#define APFS_FEATURE_STRICTATIME                0x00000008LL
#define APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE    0x00000010LL

#define APFS_SUPPORTED_FEATURES_MASK    (APFS_FEATURE_DEFRAG \
                                       | APFS_FEATURE_DEFRAG_PRERELEASE \
                                       | APFS_FEATURE_HARDLINK_MAP_RECORDS \
                                       | APFS_FEATURE_STRICTATIME \
                                       | APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE)
```

These flags are used by the `apfs_features` field of `apfs_superblock_t`.

`APFS_FEATURE_DEFRAG_PRERELEASE`

Reserved.

```
#define APFS_FEATURE_DEFRAG_PRERELEASE 0x00000001LL
```

> **Warning**
>
> To avoid data corruption, this flag must not be set.
>
> This flag enabled a prerelease version of the defragmentation system in macOS 10.13 versions. It's ignored by macOS 10.13.6 and later.

`APFS_FEATURE_HARDLINK_MAP_RECORDS`

The volume has hardlink map records.

```
#define APFS_FEATURE_HARDLINK_MAP_RECORDS 0x00000002LL
```

For details about hardlink map records, see Siblings.

`APFS_FEATURE_DEFRAG`

The volume supports defragmentation.

```
#define APFS_FEATURE_DEFRAG 0x00000004LL
```

This flag is ignored by versions before macOS 10.14.

`APFS_FEATURE_STRICTATIME`

This volume updates file access times every time the file is read.

```
#define APFS_FEATURE_STRICTATIME 0x00000008LL
```

If this flag is set, the `access_time` field of `j_inode_val_t` is updated every time the file is read. Otherwise, that field is updated when the file is read, but only if its value is prior to the timestamp stored in the `mod_time` field.

APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE

This volume supports mounting a system and data volume as a single user-visible volume.

```
#define APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE 0x00000010LL
```

This feature is used by macOS 10.15 and later to combine a read-only system volume with its corresponding read-write user data volume. Both volumes have the same value for the `apfs_volume_group_id` field of `apfs_superblock_t`, which indicates they form a volume group.

If this flag is set, inode numbers on those volumes are assigned as follows: The volume whose role is `APFS_VOL_ROLE_DATA` uses inode numbers less than `UNIFIED_ID_SPACE_MARK`, and the volume whose role is `APFS_VOL_ROLE_SYSTEM` uses inode numbers `UNIFIED_ID_SPACE_MARK` and larger. The first 16 inode numbers for both the system and data volume are reserved, as described in Inode Numbers.

APFS_SUPPORTED_FEATURES_MASK

A bit mask of all the optional volume features.

```
#define APFS_SUPPORTED_FEATURES_MASK (APFS_FEATURE_DEFRAG \
                                    | APFS_FEATURE_DEFRAG_PRERELEASE \
                                    | APFS_FEATURE_HARDLINK_MAP_RECORDS \
                                    | APFS_FEATURE_STRICTATIME \
                                    | APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE)
```

Read-Only Compatible Volume Feature Flags

The flags used to describe read-only compatible features of an Apple File System volume.

```
#define APFS_SUPPORTED_ROCOMPAT_MASK    (0x0ULL)
```

These flags are used by the `apfs_readonly_compatible_features` field of `apfs_superblock_t`. There are currently none defined.

APFS_SUPPORTED_ROCOMPAT_MASK

A bit mask of all read-only compatible volume features.

```
#define APFS_SUPPORTED_ROCOMPAT_MASK (0x0ULL)
```

Incompatible Volume Feature Flags

The flags used to describe backward-incompatible features of an Apple File System volume.

```
#define APFS_INCOMPAT_CASE_INSENSITIVE          0x00000001LL
#define APFS_INCOMPAT_DATALESS_SNAPS            0x00000002LL
#define APFS_INCOMPAT_ENC_ROLLED                0x00000004LL
#define APFS_INCOMPAT_NORMALIZATION_INSENSITIVE 0x00000008LL
#define APFS_INCOMPAT_INCOMPLETE_RESTORE        0x00000010LL
#define APFS_INCOMPAT_SEALED_VOLUME             0x00000020LL
```

```
#define APFS_INCOMPAT_RESERVED_40               0x00000040LL

#define APFS_SUPPORTED_INCOMPAT_MASK    (APFS_INCOMPAT_CASE_INSENSITIVE \
                                       | APFS_INCOMPAT_DATALESS_SNAPS \
                                       | APFS_INCOMPAT_ENC_ROLLED \
                                       | APFS_INCOMPAT_NORMALIZATION_INSENSITIVE \
                                       | APFS_INCOMPAT_INCOMPLETE_RESTORE \
                                       | APFS_INCOMPAT_SEALED_VOLUME \
                                       | APFS_INCOMPAT_RESERVED_40)
```

These flags are used by the `apfs_incompatible_features` field of `apfs_superblock_t`.

APFS_INCOMPAT_CASE_INSENSITIVE

Filenames on this volume are case insensitive.

```
#define APFS_INCOMPAT_CASE_INSENSITIVE 0x00000001LL
```

APFS_INCOMPAT_DATALESS_SNAPS

At least one snapshot with no data exists for this volume.

```
#define APFS_INCOMPAT_DATALESS_SNAPS 0x00000002LL
```

APFS_INCOMPAT_ENC_ROLLED

This volume's encryption has changed keys at least once.

```
#define APFS_INCOMPAT_ENC_ROLLED 0x00000004LL
```

APFS_INCOMPAT_NORMALIZATION_INSENSITIVE

Filenames on this volume are normalization insensitive.

```
#define APFS_INCOMPAT_NORMALIZATION_INSENSITIVE 0x00000008LL
```

Normalization insensitivity is part of hashing filenames, as described in the `name_len_and_hash` field of `j_drec_hashed_key_t`.

APFS_INCOMPAT_INCOMPLETE_RESTORE

This volume is being restored, or a restore operation to this volume was uncleanly aborted.

```
#define APFS_INCOMPAT_INCOMPLETE_RESTORE 0x00000010LL
```

APFS_INCOMPAT_SEALED_VOLUME

This volume can't be modified.

```
#define APFS_INCOMPAT_SEALED_VOLUME 0x00000020LL
```

For more information, see Sealed Volumes.

APFS_INCOMPAT_RESERVED_40

Reserved.

```
#define APFS_INCOMPAT_RESERVED_40 0x00000040LL
```

APFS_SUPPORTED_INCOMPAT_MASK

A bit mask of all the backward-incompatible volume features.

```
#define APFS_SUPPORTED_INCOMPAT_MASK (APFS_INCOMPAT_CASE_INSENSITIVE \
                                    | APFS_INCOMPAT_DATALESS_SNAPS \
                                    | APFS_INCOMPAT_ENC_ROLLED \
                                    | APFS_INCOMPAT_NORMALIZATION_INSENSITIVE \
                                    | APFS_INCOMPAT_INCOMPLETE_RESTORE)
```

File-System Objects

A file-system object stores information about a part of the file system, like a directory or a file on disk. These objects are stored as one or more records. For example, the file-system object for a directory that contains two files is stored as three records: a record of type APFS_TYPE_INODE for the inode, and two records of type APFS_TYPE_DIR_REC for the directory entries. This record-based method of storing file-system objects helps make efficient use of disk space.

File-system records are stored as key/value pairs in a B-tree. The key contains information, like the object identifier and the record type, used to look up a record. Keys begin with an instance of j_key_t, and many records use j_key_t as their entire key.

For sorting file-system records — for example, to keep them ordered in a B-tree — the following comparison rules are used:

1. Compare the object identifiers numerically:

   ```
   j_key_t.obj_id_and_type & OBJ_ID_MASK
   ```

2. Compare the object types numerically:

   ```
   (j_key_t.obj_id_and_type & OBJ_TYPE_MASK) >> OBJ_TYPE_SHIFT
   ```

3. For extended attribute records and directory entry records, compare the names lexicographically:

   ```
   j_drec_key_t.name
   ```

Because all of the records for a file-system object have the same object identifier, all of the records that make up a single object are sorted next to each other.

The relationship between file-system objects and the records they're made up from is as follows:

Files

- APFS_TYPE_INODE Required
- APFS_TYPE_CRYPTO_STATE
- APFS_TYPE_DSTREAM_ID
- APFS_TYPE_EXTENT
- APFS_TYPE_FILE_EXTENT
- APFS_TYPE_SIBLING_LINK
- APFS_TYPE_XATTR

Directories

- APFS_TYPE_INODE Required
- APFS_TYPE_CRYPTO_STATE
- APFS_TYPE_DIR_REC
- APFS_TYPE_DIR_STATS
- APFS_TYPE_XATTR

Symbolic Links

- APFS_TYPE_INODE Required
- APFS_TYPE_XATTR Required

- APFS_TYPE_CRYPTO_STATE
- APFS_TYPE_DSTREAM_ID
- APFS_TYPE_EXTENT
- APFS_TYPE_FILE_EXTENT

There must be an extended attribute whose name is SYMLINK_EA_NAME and whose value is the path to the target file.

Snapshots

- APFS_TYPE_SNAP_METADATA Required
- APFS_TYPE_SNAP_NAME Required
- APFS_TYPE_CRYPTO_STATE
- APFS_TYPE_EXTENT

Sibling Maps

- APFS_TYPE_SIBLING_MAP Required

> **Tip**
>
> To simplify manipulating file-system objects, define custom types that combine the key and value of a record, and custom types that combine the object's records.

j_key_t

A header used at the beginning of all file-system keys.

```
struct j_key {
    uint64_t    obj_id_and_type;
} __attribute__((packed));
typedef struct j_key j_key_t;

#define OBJ_ID_MASK                 0x0fffffffffffffffULL
#define OBJ_TYPE_MASK               0xf000000000000000ULL
#define OBJ_TYPE_SHIFT              60

#define SYSTEM_OBJ_ID_MARK          0x0fffffff00000000ULL
```

All file-system objects have a key that begins with this information. The key for some object types have additional fields that follow this header, and other object types use `j_key_t` as their entire key.

The following record types use this structure as their key without adding any additional fields:

obj_id_and_type

A bit field that contains the object's identifier and its type.

```
uint64_t obj_id_and_type;
```

The object's identifier is a uint64_t value accessed as obj_id_and_type & OBJ_ID_MASK. The object's type is a uint8_t value accessed as (obj_id_and_type & OBJ_TYPE_MASK) >> OBJ_TYPE_SHIFT. The object's type is one of the constants defined by j_obj_types.

OBJ_ID_MASK

The bit mask used to access the object identifier.

```
#define OBJ_ID_MASK 0x0fffffffffffffffULL
```

OBJ_TYPE_MASK

The bit mask used to access the object type.

```
#define OBJ_TYPE_MASK 0xf000000000000000ULL
```

OBJ_TYPE_SHIFT

The bit shift used to access the object type.

```
#define OBJ_TYPE_SHIFT 60
```

SYSTEM_OBJ_ID_MARK

The smallest object identifier used by the system volume.

```
#define SYSTEM_OBJ_ID_MARK          0x0fffffff00000000ULL
```

In a volume group, objects with an identifier less than this number are part of the data volume, and objects with an identifier greater than or equal to this number are part of the system volume.

j_inode_key_t

The key half of a directory-information record.

```
struct j_inode_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_inode_key_t j_inode_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier, also known as its inode number. The type in the header is always APFS_TYPE_INODE.

j_inode_val_t

The value half of an inode record.

```
struct j_inode_val {
    uint64_t        parent_id;
    uint64_t        private_id;

    uint64_t        create_time;
    uint64_t        mod_time;
    uint64_t        change_time;
    uint64_t        access_time;

    uint64_t        internal_flags;

    union {
        int32_t     nchildren;
        int32_t     nlink;
    };

    cp_key_class_t  default_protection_class;
    uint32_t        write_generation_counter;
    uint32_t        bsd_flags;
    uid_t           owner;
    gid_t           group;
    mode_t          mode;
    uint16_t        pad1;
    uint64_t        uncompressed_size;
    uint8_t         xfields[];
} __attribute__((packed));
typedef struct j_inode_val j_inode_val_t;

typedef uint32_t    uid_t;
typedef uint32_t    gid_t;
```

parent_id

The identifier of the file system record for the parent directory.

```
uint64_t parent_id;
```

private_id

The unique identifier used by this file's data stream.

```
uint64_t private_id;
```

This identifier appears in the owning_obj_id field of j_phys_ext_val_t records that describe the extents where the data is stored.

For an inode that doesn't have data, the value of this field is the file-system object's identifier.

create_time

The time that this record was created.

```
uint64_t create_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

mod_time

The time that this record was last modified.

```
uint64_t mod_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

change_time

The time that this record's attributes were last modified.

```
uint64_t change_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

access_time

The time that this record was last accessed.

```
uint64_t access_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

For details about when this field is updated, see `APFS_FEATURE_STRICTATIME`.

internal_flags

The inode's flags.

```
uint64_t internal_flags;
```

For the values used in this bit field, see `j_inode_flags`.

nchildren

The number of directory entries.

```
int32_t nchildren;
```

This union field is valid only if the inode is a directory.

nlink

The number of hard links whose target is this inode.

```
int32_t nlink;
```

This union field is valid only if the inode isn't a directory.

Inodes with multiple hard links — as indicated by a value greater than one in this field — have additional invariants:

- The `parent_id` field refers to the parent directory of the primary link.
- The `name` field contains the name of the primary link.
- The `INO_EXT_TYPE_NAME` extended field contains the name of this link.
- The file-system object includes sibling-link records, as discussed in Siblings.

default_protection_class

The default protection class for this inode.

```
cp_key_class_t default_protection_class;
```

Files in this directory that have a protection class of `PROTECTION_CLASS_DIR_NONE` use the directory's default protection class.

write_generation_counter

A monotonically increasing counter that's incremented each time this inode or its data is modified.

```
uint32_t write_generation_counter;
```

This value is allowed to overflow and restart from zero.

bsd_flags

The inode's BSD flags.

```
uint32_t bsd_flags;
```

For information about these flags, see the `chflags(2)` man page and the `<sys/stat.h>` header file.

owner

The user identifier of the inode's owner.

```
uid_t owner;
```

group

The group identifier of the inode's group.

```
gid_t group;
```

mode

The file's mode.

```
mode_t mode;
```

For possible values, see File Modes.

pad1

Reserved.

```
uint16_t pad1;
```

Populate this field with zero when you create a new inode, and preserve its value when you modify an existing inode.

This field is padding.

uncompressed_size

The size of the file without compression.

```
uint64_t uncompressed_size;
```

This field is populated only for files that have the `INODE_HAS_UNCOMPRESSED_SIZE` flag set on the `internal_flags` field.

For files that don't have the flag set, this field is treated as padding: Populate this field with zero when you create a new inode, and preserve its value when you modify an existing inode.

xfields

The inode's extended fields.

```
uint8_t xfields[];
```

This location on disk contains several pieces of data that have variable sizes. For information about reading extended fields, see Extended Fields.

uid_t

A user identifier.

```
typedef uint32_t uid_t;
```

gid_t

A group identifier.

```
typedef uint32_t gid_t;
```

j_drec_key_t

The key half of a directory entry record.

```
struct j_drec_key {
    j_key_t     hdr;
    uint16_t    name_len;
    uint8_t     name[0];
} __attribute__((packed));
typedef struct j_drec_key j_drec_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always `APFS_TYPE_DIR_REC`.

name_len_and_hash

The length of the name, including the final null character (U+0000).

```
uint32_t name_len_and_hash;
```

name

The name, represented as a null-terminated UTF-8 string.

```
uint8_t name[0];
```

j_drec_hashed_key_t

The key half of a directory entry record, including a precomputed hash of its name.

```
struct j_drec_hashed_key {
    j_key_t     hdr;
    uint32_t    name_len_and_hash;
    uint8_t     name[0];
} __attribute__((packed));
typedef struct j_drec_hashed_key j_drec_hashed_key_t;

#define J_DREC_LEN_MASK             0x000003ff
#define J_DREC_HASH_MASK            0xfffff400
#define J_DREC_HASH_SHIFT           10
```

hdr

The record's header.

```
j_key_t hdr;
```


name_len_and_hash

The hash and length of the name.

```
uint32_t name_len_and_hash;
```

The length is a 10-bit unsigned integer, accessed as `name_len_and_hash & J_DREC_LEN_MASK`. The length includes the final null character (U+0000).

The hash is an unsigned 22-bit integer, accessed as `(name_len_and_hash & J_DREC_HASH_MASK) >> J_DREC_HASH_SHIFT`. The hash is computed as follows:

1. Start with the filename, represented as a null-terminated UTF-8 string.
2. Normalize the string using canonical decomposition (NFD).
3. Represent the normalized filename as a null-terminated UTF-32 string.
4. Compute the CRC-32C hash of the UTF-32 string.
5. Complement the bits of the hash.
6. Keep only the low 22 bits of the hash.

If you implement your own CRC function, rather than calling one from a library, you can omit both the complement operation that's part of computing a CRC and the complement operation in the instructions above.

name

The name, represented as a null-terminated UTF-8 string.

```
uint8_t name[0];
```

J_DREC_LEN_MASK

The bit mask used to access the length of the name.

```
#define J_DREC_LEN_MASK 0x000003ff
```

J_DREC_HASH_MASK

The bit mask used to access the hash of the name.

```
#define J_DREC_HASH_MASK 0xfffff400
```

J_DREC_HASH_SHIFT

The bit shift used to access the hash of the name.

```
#define J_DREC_HASH_SHIFT 10
```

j_drec_val_t

The value half of a directory entry record.

```
struct j_drec_val {
    uint64_t    file_id;
    uint64_t    date_added;
    uint16_t    flags;
```

```
    uint8_t     xfields[];
} __attribute__((packed));
typedef struct j_drec_val j_drec_val_t;
```

file_id

The identifier of the inode that this directory entry represents.

```
uint64_t file_id;
```

date_added

The time that this directory entry was added to the directory.

```
uint64_t date_added;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds. It's not updated when modifying the directory entry — for example, by renaming a file without moving it to a different directory.

flags

The directory entry's flags.

```
uint16_t flags;
```

The bits that are set in DREC_TYPE_MASK store the inode's file type, and the remaining bits are reserved. Populate the reserved bits with zeros when you create a new directory entry, and preserve their values when you modify an existing directory entry.

For possible values, see Directory Entry File Types.

xfields

The directory entry's extended fields.

```
uint8_t xfields[];
```

This location on disk contains several pieces of data that have variable sizes. For information about reading extended fields, see Extended Fields.

j_dir_stats_key_t

The key half of a directory-information record.

```
struct j_dir_stats_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_dir_stats_key j_dir_stats_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always APFS_TYPE_DIR_REC.

j_dir_stats_val_t

The value half of a directory-information record.

```
struct j_dir_stats_val {
    uint64_t    num_children;
    uint64_t    total_size;
    uint64_t    chained_key;
    uint64_t    gen_count;
} __attribute__((packed));
typedef struct j_dir_stats_val j_dir_stats_val_t;
```

num_children

The number of files and folders contained by the directory.

```
uint64_t num_children;
```

total_size

The total size, in bytes, of all the files stored in this directory and all of this directory's descendants.

```
uint64_t total_size;
```

Hard links contribute to the `total_size` of every directory they appear in.

chained_key

The parent directory's file system object identifier.

```
uint64_t chained_key;
```

gen_count

A monotonically increasing counter that's incremented each time this inode or any of its children is modified.

```
uint64_t gen_count;
```

Modifying the contents of a file requires updating the inode's modification time and write generation, which means this counter must be incremented for the directory that contains the file.

If this counter can't be incremented without overflow, that's an unrecoverable error.

j_xattr_key_t

The key half of an extended attribute record.

```
struct j_xattr_key {
    j_key_t     hdr;
    uint16_t    name_len;
    uint8_t     name[0];
} __attribute__((packed));
typedef struct j_xattr_key j_xattr_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always APFS_TYPE_XATTR.

name_len

The length of the extended attribute's name, including the final null character (U+0000).

```
uint16_t name_len;
```

name

The extended attribute's name, represented as a null-terminated UTF-8 string.

```
uint8_t name[0];
```

j_xattr_val_t

The value half of an extended attribute record.

```
struct j_xattr_val {
    uint16_t    flags;
    uint16_t    xdata_len;
    uint8_t     xdata[0];
} __attribute__((packed));
typedef struct j_xattr_val j_xattr_val_t;
```

flags

The extended attribute record's flags.

```
uint16_t flags;
```

For the values used in this bit field, see j_xattr_flags. Either the XATTR_DATA_EMBEDDED or XATTR_DATA_STREAM flag must be set.

xdata_len

The length of the extended attribute data.

```
uint16_t xdata_len;
```

If the XATTR_DATA_EMBEDDED flag is set, this field is the length of the data in the xdata field. Otherwise, this field is ignored.

xdata

The extended attribute data or the identifier of a data stream that contains the data.

```
uint8_t xdata[0];
```

If the XATTR_DATA_EMBEDDED flag is set, the extended attribute data is stored directly in this field. Otherwise, this field contains the identifier (uint64_t) for a data stream record that stores the extended attribute data. See also j_xattr_dstream_t.

File-System Constants

File-system objects use several groups of constants to define values for record types, reserved inode numbers, and flags and bit masks used in bit fields.

j_obj_types

The type of a file-system record.

```
typedef enum {
        APFS_TYPE_ANY              = 0,

        APFS_TYPE_SNAP_METADATA    = 1,
        APFS_TYPE_EXTENT           = 2,
        APFS_TYPE_INODE            = 3,
        APFS_TYPE_XATTR            = 4,
        APFS_TYPE_SIBLING_LINK     = 5,
        APFS_TYPE_DSTREAM_ID       = 6,
        APFS_TYPE_CRYPTO_STATE     = 7,
        APFS_TYPE_FILE_EXTENT      = 8,
        APFS_TYPE_DIR_REC          = 9,
        APFS_TYPE_DIR_STATS        = 10,
        APFS_TYPE_SNAP_NAME        = 11,
        APFS_TYPE_SIBLING_MAP      = 12,
        APFS_TYPE_FILE_INFO        = 13,

        APFS_TYPE_MAX_VALID        = 13,
        APFS_TYPE_MAX              = 15,

        APFS_TYPE_INVALID          = 15,
} j_obj_types;
```

This value is stored in the type bits of a `j_key_t` structure's `obj_id_and_type` field.

APFS_TYPE_ANY

A record of any type.

```
APFS_TYPE_ANY = 0
```

This enumeration case is used only in search queries and in tests when iterating over objects. It's not valid as the type of a file-system object.

APFS_TYPE_SNAP_METADATA

Metadata about a snapshot.

```
APFS_TYPE_SNAP_METADATA = 1
```

The key is an instance of `j_snap_metadata_key_t` and the value is an instance of `j_snap_metadata_val_t`.

APFS_TYPE_EXTENT

A physical extent record.

`APFS_TYPE_EXTENT = 2`

The key is an instance of `j_phys_ext_key_t` and the value is an instance of `j_phys_ext_val_t`.

APFS_TYPE_INODE

An inode.

`APFS_TYPE_INODE = 3`

The key is an instance of `j_inode_key_t` and the value is an instance of `j_inode_val_t`.

APFS_TYPE_XATTR

An extended attribute.

`APFS_TYPE_XATTR = 4`

The key is an instance of `j_xattr_key_t` and the value is an instance of `j_xattr_val_t`.

APFS_TYPE_SIBLING_LINK

A mapping from an inode to hard links that the inode is the target of.

`APFS_TYPE_SIBLING_LINK = 5`

The key is an instance of `j_sibling_key_t` and the value is an instance of `j_sibling_val_t`.

APFS_TYPE_DSTREAM_ID

A data stream.

`APFS_TYPE_DSTREAM_ID = 6`

The key is an instance of `j_dstream_id_key_t` and the value is an instance of `j_dstream_id_val_t`.

APFS_TYPE_CRYPTO_STATE

A per-file encryption state.

`APFS_TYPE_CRYPTO_STATE = 7`

The key is an instance of `j_crypto_key_t` and the value is an instance of `j_crypto_val_t`. This object type is used only by iOS devices, except for a placeholder object whose identifier is always `CRYPTO_SW_ID`.

APFS_TYPE_FILE_EXTENT

A physical extent record for a file.

`APFS_TYPE_FILE_EXTENT = 8`

The key is an instance of `j_file_extent_key_t` and the value is an instance of `j_file_extent_val_t`.

APFS_TYPE_DIR_REC

A directory entry.

`APFS_TYPE_DIR_REC = 9`

The key is an instance of `j_drec_key_t` and the value is an instance of `j_drec_val_t`.

APFS_TYPE_DIR_STATS

Information about a directory.

`APFS_TYPE_DIR_STATS = 10`

The key is an instance of `j_dir_stats_key_t` and the value is an instance of `j_drec_val_t`.

APFS_TYPE_SNAP_NAME

The name of a snapshot.

`APFS_TYPE_SNAP_NAME = 11`

The key is an instance of `j_snap_name_key_t` and the value is an instance of `j_snap_name_val_t`.

APFS_TYPE_SIBLING_MAP

A mapping from a hard link to its target inode.

`APFS_TYPE_SIBLING_MAP = 12`

The key is an instance of `j_sibling_map_key_t` and the value is an instance of `j_sibling_map_val_t`.

APFS_TYPE_FILE_INFO

Additional information about file data.

`APFS_TYPE_FILE_INFO = 13`

The key is an instance of `j_file_info_key_t` and the value is an instance of `j_file_info_val_t`.

APFS_TYPE_MAX_VALID

The largest valid value for a file-system object's type.

`APFS_TYPE_MAX_VALID = 13`

APFS_TYPE_MAX

The largest value for a file-system object's type.

`APFS_TYPE_MAX = 15`

APFS_TYPE_INVALID

An invalid object type.

APFS_TYPE_INVALID = 15

j_obj_kinds

The kind of a file-system record.

```
typedef enum {
        APFS_KIND_ANY             = 0,
        APFS_KIND_NEW             = 1,
        APFS_KIND_UPDATE          = 2,
        APFS_KIND_DEAD            = 3,
        APFS_KIND_UPDATE_REFCNT   = 4,

        APFS_KIND_INVALID         = 255
} j_obj_kinds;
```

This value is stored in the kind bits of a j_phys_ext_val_t structure's len_and_kind field.

APFS_KIND_ANY

A record of any kind.

APFS_KIND_ANY = 0

This value isn't valid as the kind of a file-system record on disk. However, implementations of Apple File System can use it internally — for example, in search queries and in tests when iterating over objects.

APFS_KIND_NEW

A new record.

APFS_KIND_NEW = 1

This record adds data that isn't part of any snapshots.

APFS_KIND_UPDATE

An updated record.

APFS_KIND_UPDATE = 2

This record changes data that's part of an existing snapshot.

APFS_KIND_DEAD

A record that's being deleted.

APFS_KIND_DEAD = 3

This value isn't valid as the kind of a file-system record on disk. However, implementations of Apple File System can use it internally.

APFS_KIND_UPDATE_REFCNT

An update to the reference count of a record.

```
APFS_KIND_UPDATE_REFCNT = 4
```

This value isn't valid as the kind of a file-system record on disk. However, implementations of Apple File System can use it internally.

APFS_KIND_INVALID

An invalid record kind.

```
APFS_KIND_INVALID = 255
```

j_inode_flags

The flags used by inodes.

```
typedef enum {
    INODE_IS_APFS_PRIVATE           = 0x00000001,
    INODE_MAINTAIN_DIR_STATS        = 0x00000002,
    INODE_DIR_STATS_ORIGIN          = 0x00000004,
    INODE_PROT_CLASS_EXPLICIT       = 0x00000008,
    INODE_WAS_CLONED                = 0x00000010,
    INODE_FLAG_UNUSED               = 0x00000020,
    INODE_HAS_SECURITY_EA           = 0x00000040,
    INODE_BEING_TRUNCATED           = 0x00000080,
    INODE_HAS_FINDER_INFO           = 0x00000100,
    INODE_IS_SPARSE                 = 0x00000200,
    INODE_WAS_EVER_CLONED           = 0x00000400,
    INODE_ACTIVE_FILE_TRIMMED       = 0x00000800,
    INODE_PINNED_TO_MAIN            = 0x00001000,
    INODE_PINNED_TO_TIER2           = 0x00002000,
    INODE_HAS_RSRC_FORK             = 0x00004000,
    INODE_NO_RSRC_FORK              = 0x00008000,
    INODE_ALLOCATION_SPILLEDOVER    = 0x00010000,
    INODE_FAST_PROMOTE              = 0x00020000,
    INODE_HAS_UNCOMPRESSED_SIZE     = 0x00040000,
    INODE_IS_PURGEABLE              = 0x00080000,
    INODE_WANTS_TO_BE_PURGEABLE     = 0x00100000,
    INODE_IS_SYNC_ROOT              = 0x00200000,
    INODE_SNAPSHOT_COW_EXEMPTION    = 0x00400000,


    INODE_INHERITED_INTERNAL_FLAGS  = (INODE_MAINTAIN_DIR_STATS \
                                     | INODE_SNAPSHOT_COW_EXEMPTION),

    INODE_CLONED_INTERNAL_FLAGS     = (INODE_HAS_RSRC_FORK \
                                     | INODE_NO_RSRC_FORK \
```

```
                                    | INODE_HAS_FINDER_INFO \
                                    | INODE_SNAPSHOT_COW_EXEMPTION),
} j_inode_flags;

#define APFS_VALID_INTERNAL_INODE_FLAGS (INODE_IS_APFS_PRIVATE \
                                        | INODE_MAINTAIN_DIR_STATS \
                                        | INODE_DIR_STATS_ORIGIN \
                                        | INODE_PROT_CLASS_EXPLICIT \
                                        | INODE_WAS_CLONED \
                                        | INODE_HAS_SECURITY_EA \
                                        | INODE_BEING_TRUNCATED \
                                        | INODE_HAS_FINDER_INFO \
                                        | INODE_IS_SPARSE \
                                        | INODE_WAS_EVER_CLONED \
                                        | INODE_ACTIVE_FILE_TRIMMED \
                                        | INODE_PINNED_TO_MAIN \
                                        | INODE_PINNED_TO_TIER2 \
                                        | INODE_HAS_RSRC_FORK \
                                        | INODE_NO_RSRC_FORK \
                                        | INODE_ALLOCATION_SPILLEDOVER \
                                        | INODE_FAST_PROMOTE \
                                        | INODE_HAS_UNCOMPRESSED_SIZE \
                                        | INODE_IS_PURGEABLE \
                                        | INODE_WANTS_TO_BE_PURGEABLE \
                                        | INODE_IS_SYNC_ROOT \
                                        | INODE_SNAPSHOT_COW_EXEMPTION)

#define APFS_INODE_PINNED_MASK (INODE_PINNED_TO_MAIN | INODE_PINNED_TO_TIER2)
```

INODE_IS_APFS_PRIVATE

The inode is used internally by an implementation of Apple File System.

```
INODE_IS_APFS_PRIVATE = 0x00000001
```

Inodes with this flag set aren't considered part of the volume. They can't be cloned, renamed, or deleted. They're ignored by operations like counting the number of files on disk, and they're hidden from the user during operations like listing the files of a directory.

This flag isn't reserved by Apple; implementations of the Apple File System must set this flag on any inodes they create for their own record keeping. However, to prevent implementations from interfering with each other, an implementation modifies inodes with this flag only if the implementation created that inode.

Apple's implementation uses this flag for temporary files.

See also PRIV_DIR_INO_NUM.

INODE_MAINTAIN_DIR_STATS

The inode tracks the size of all of its children.

INODE_MAINTAIN_DIR_STATS = 0x00000002

This flag is only valid on a directory, and must also be set on the directory's subdirectories.

When removing the INODE_MAINTAIN_DIR_STATS flag from a directory, walk its subdirectories and remove it from any directories that inherited it from this directory. Directories that have the INODE_DIR_STATS_ORIGIN flag set, and subdirectories of those directories, continue to have the INODE_MAINTAIN_DIR_STATS flag set, because they don't inherit it from this directory.

INODE_DIR_STATS_ORIGIN

The inode has the INODE_MAINTAIN_DIR_STATS flag set explicitly, not due to inheritance.

INODE_DIR_STATS_ORIGIN = 0x00000004

More than one directory in a hierarchy can have this flag set.

INODE_PROT_CLASS_EXPLICIT

The inode's data protection class was set explicitly when the inode was created.

INODE_PROT_CLASS_EXPLICIT = 0x00000008

INODE_WAS_CLONED

The inode was created by cloning another inode.

INODE_WAS_CLONED = 0x00000010

INODE_FLAG_UNUSED

Reserved.

INODE_FLAG_UNUSED = 0x00000020

Leave this flag unset when you create a new inode, and preserve its value when you modify an existing inode.

INODE_HAS_SECURITY_EA

The inode has an access control list.

INODE_HAS_SECURITY_EA = 0x00000040

INODE_BEING_TRUNCATED

The inode was truncated.

INODE_BEING_TRUNCATED = 0x00000080

This flag is used as follows to allow the truncation operation to complete after a crash:

1. The system is asked to truncate an inode
2. This flag is set on the inode
3. The system starts truncating the file
4. A crash occurs

5. In the post-crash recovery process, this flag is detected
6. The system finishes truncating the inode

Note that after a crash, the truncation operation might not resume until the next time the inode is accessed.

`INODE_HAS_FINDER_INFO`

The inode has a Finder info extended field.

`INODE_HAS_FINDER_INFO = 0x00000100`

See also `INO_EXT_TYPE_FINDER_INFO`.

`INODE_IS_SPARSE`

The inode has a sparse byte count extended field.

`INODE_IS_SPARSE = 0x00000200`

See also `INO_EXT_TYPE_SPARSE_BYTES`.

`INODE_WAS_EVER_CLONED`

The inode has been cloned at least once.

`INODE_WAS_EVER_CLONED = 0x00000400`

If this flag is set, the blocks on disk that store this inode might also be in use with another inode. For example, when deleting this inode, you need to check reference counts before deallocating storage.

Versions of macOS prior to 10.13.3 had a known issue where this flag could be set incorrectly. Before reading this flag, confirm that the inode's object identifier is larger than the value stored in the `apfs_cloneinfo_id_epoch` field of `apfs_superblock_t`. In addition, to ensure that the volume hasn't been modified by an older OS version, confirm that the value of the `apfs_cloneinfo_xid` field and the `apfs_modified_by` field of `apfs_superblock_t` contain the same value.

`INODE_ACTIVE_FILE_TRIMMED`

The inode is an overprovisioning file that has been trimmed.

`INODE_ACTIVE_FILE_TRIMMED = 0x00000800`

This file type is used only on devices running iOS. By allocating space for the file, but never writing to that space, extra blocks are set aside for overprovisioning that's performed by the underlying NAND storage.

`INODE_PINNED_TO_MAIN`

The inode's file content is always on the main storage device.

`INODE_PINNED_TO_MAIN = 0x00001000`

This flag is only valid for Fusion systems. The main storage is a solid-state drive.

INODE_PINNED_TO_TIER2

The inode's file content is always on the secondary storage device.

```
INODE_PINNED_TO_TIER2 = 0x00002000
```

This flag is only valid for Fusion systems. The secondary storage is a hard drive.

INODE_HAS_RSRC_FORK

The inode has a resource fork.

```
INODE_HAS_RSRC_FORK = 0x00004000
```

If this flag is set, `INODE_NO_RSRC_FORK` must not be set. It's also valid for neither flag to be set, which implicitly indicates that the inode doesn't have a resource fork.

INODE_NO_RSRC_FORK

The inode doesn't have a resource fork.

```
INODE_NO_RSRC_FORK = 0x00008000
```

If this flag is set, `INODE_HAS_RSRC_FORK` must not be set. It's also valid for neither flag to be set, which implicitly indicates that the inode doesn't have a resource fork.

INODE_ALLOCATION_SPILLEDOVER

The inode's file content has some space allocated outside of the preferred storage tier for that file.

```
INODE_ALLOCATION_SPILLEDOVER = 0x00010000
```

See also `APFS_FS_SPILLEDOVER`.

INODE_FAST_PROMOTE

This inode is scheduled for promotion from slow storage to fast storage.

```
INODE_FAST_PROMOTE = 0x00020000
```

The promotion between tiers will happen the first time this inode is read.

INODE_HAS_UNCOMPRESSED_SIZE

This inode stores its uncompressed size in the inode.

```
INODE_HAS_UNCOMPRESSED_SIZE = 0x00040000
```

The uncompressed size is stored in the `uncompressed_size` field of `j_inode_val_t`.

Prior to macOS 10.15 and iOS 13.1, this flag was ignored and Apple's implementation always treated the `uncompressed_size` field as padding.

INODE_IS_PURGEABLE

This inode will be deleted at the next purge.

```
INODE_IS_PURGEABLE = 0x00080000
```

A purge is requested from user space by part of the operating system, and the process of deleting purgeable files is the responsibility of the operating system.

INODE_WANTS_TO_BE_PURGEABLE

This inode should become purgeable when its link count drops to one.

```
INODE_WANTS_TO_BE_PURGEABLE = 0x00100000
```

INODE_IS_SYNC_ROOT

This inode is the root of a sync hierarchy for `fileproviderd`.

```
INODE_IS_SYNC_ROOT = 0x00200000
```

Don’t add or remove this flag, but preserve the flag if it already exists.

To prevent data loss, Apple’s implementation coordinates with `fileproviderd` during operations such as renaming a file in a sync hierarchy, moving a file from inside a sync hierarchy out of that hierarchy, and moving a file from outside of a sync hierarchy into that hierarchy. Other implementations of the Apple File System should treat requests to perform these operations as errors.

INODE_SNAPSHOT_COW_EXEMPTION

This inode is exempt from copy-on-write behavior if the data is part of a snapshot.

```
INODE_SNAPSHOT_COW_EXEMPTION = 0x00400000
```

Don’t add or remove this flag, but preserve the flag if it already exists.

The number of files with this flag is tracked by the `APFS_COW_EXEMPT_COUNT_NAME` extended attribute.

INODE_INHERITED_INTERNAL_FLAGS

A bit mask of the flags that are inherited by the files and subdirectories in a directory.

```
INODE_INHERITED_INTERNAL_FLAGS = (INODE_MAINTAIN_DIR_STATS \
                               | INODE_SNAPSHOT_COW_EXEMPTION)
```

INODE_CLONED_INTERNAL_FLAGS

A bit mask of the flags that are preserved when cloning.

```
INODE_CLONED_INTERNAL_FLAGS = (INODE_HAS_RSRC_FORK
                            | INODE_NO_RSRC_FORK \
                            | INODE_HAS_FINDER_INFO \
                            | INODE_SNAPSHOT_COW_EXEMPTION)
```

APFS_VALID_INTERNAL_INODE_FLAGS

A bit mask of all valid flags.

```
#define APFS_VALID_INTERNAL_INODE_FLAGS (INODE_IS_APFS_PRIVATE \
                                       | INODE_MAINTAIN_DIR_STATS \
                                       | INODE_DIR_STATS_ORIGIN \
                                       | INODE_PROT_CLASS_EXPLICIT \
                                       | INODE_WAS_CLONED \
                                       | INODE_HAS_SECURITY_EA \
                                       | INODE_BEING_TRUNCATED \
                                       | INODE_HAS_FINDER_INFO \
                                       | INODE_IS_SPARSE \
                                       | INODE_WAS_EVER_CLONED \
                                       | INODE_ACTIVE_FILE_TRIMMED \
                                       | INODE_PINNED_TO_MAIN \
                                       | INODE_PINNED_TO_TIER2 \
                                       | INODE_HAS_RSRC_FORK \
                                       | INODE_NO_RSRC_FORK \
                                       | INODE_ALLOCATION_SPILLEDOVER \
                                       | INODE_FAST_PROMOTE \
                                       | INODE_HAS_UNCOMPRESSED_SIZE \
                                       | INODE_IS_PURGEABLE \
                                       | INODE_WANTS_TO_BE_PURGEABLE \
                                       | INODE_IS_SYNC_ROOT \
                                       | INODE_SNAPSHOT_COW_EXEMPTION)
```

APFS_INODE_PINNED_MASK

A bit mask of the flags that are related to pinning.

```
#define APFS_INODE_PINNED_MASK (INODE_PINNED_TO_MAIN | INODE_PINNED_TO_TIER2)
```

j_xattr_flags

The flags used in an extended attribute record to provide additional information.

```
typedef enum {
    XATTR_DATA_STREAM       = 0x00000001,
    XATTR_DATA_EMBEDDED     = 0x00000002,
    XATTR_FILE_SYSTEM_OWNED = 0x00000004,
    XATTR_RESERVED_8        = 0x00000008,
} j_xattr_flags;
```

XATTR_DATA_STREAM

The attribute data is stored in a data stream.

```
XATTR_DATA_STREAM = 0x00000001
```

If this flag is set, XATTR_DATA_EMBEDDED must not be set.

XATTR_DATA_EMBEDDED

The attribute data is stored directly in the record.

```
XATTR_DATA_EMBEDDED = 0x00000002
```

If this flag is set, the size of the value be smaller than XATTR_MAX_EMBEDDED_SIZE, and XATTR_DATA_STREAM must not be set.

XATTR_FILE_SYSTEM_OWNED

The extended attribute record is owned by the file system.

```
XATTR_FILE_SYSTEM_OWNED = 0x00000004
```

For example, this flag is used on symbolic links. The links have an extended attribute whose name is SYMLINK_EA_NAME, and this flag is set on that attribute.

XATTR_RESERVED_8

Reserved.

```
XATTR_RESERVED_8 = 0x00000008
```

Don't add this flag to an extended attribute record, but preserve the flag if it already exists.

dir_rec_flags

The flags used by directory records.

```
typedef enum {
    DREC_TYPE_MASK  = 0x000f,
    RESERVED_10     = 0x0010
} dir_rec_flags;
```

DREC_TYPE_MASK

The bit mask used to access the type.

```
DREC_TYPE_MASK = 0x000f
```

This bit mask is used with the `flags` field of j_drec_val_t.

RESERVED_10

Reserved.

```
RESERVED_10 = 0x0010
```

Don't set this flag. If you find a directory record with this flag set in production, file a bug against the Apple File System implementation.

Inode Numbers

Inodes whose number is always the same.

```
#define INVALID_INO_NUM            0

#define ROOT_DIR_PARENT            1
#define ROOT_DIR_INO_NUM           2
#define PRIV_DIR_INO_NUM           3
#define SNAP_DIR_INO_NUM           6
#define PURGEABLE_DIR_INO_NUM      7

#define MIN_USER_INO_NUM           16

#define UNIFIED_ID_SPACE_MARK      0x0800000000000000ULL
```

If the `APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE` flag is set on the volume, the system volume reserves each of the inode numbers listed above but with `UNIFIED_ID_SPACE_MARK` added to them. For example, the inode number `0x0800000000000002ULL` is equal to `ROOT_DIR_INO_NUM + UNIFIED_ID_SPACE_MARK`, meaning this inode number is reserved for the system volume's root directory.

INVALID_INO_NUM

An invalid inode number.

```
#define INVALID_INO_NUM 0
```

ROOT_DIR_PARENT

The inode number for the root directory's parent.

```
#define ROOT_DIR_PARENT 1
```

This is a sentinel value; there's no inode on disk with this inode number.

ROOT_DIR_INO_NUM

The inode number for the root directory of the volume.

```
#define ROOT_DIR_INO_NUM 2
```

PRIV_DIR_INO_NUM

The inode number for the private directory.

```
#define PRIV_DIR_INO_NUM 3
```

The private directory's filename is "private-dir". When creating a new volume, you must create a directory with this name and inode number.

This directory isn't reserved by Apple; implementations of the Apple File System can use it to store their own record-keeping information. However, to prevent implementations from interfering with each other, an implementation modifies files in the private directory only if the implementation created the files.

See also INODE_IS_APFS_PRIVATE.

SNAP_DIR_INO_NUM

The inode number for the directory where snapshot metadata is stored.

```
#define SNAP_DIR_INO_NUM 6
```

Snapshot inodes are stored in the snapshot metedata tree.

PURGEABLE_DIR_INO_NUM

The inode number used for storing references to purgeable files.

```
#define PURGEABLE_DIR_INO_NUM 7
```

This inode number and the directory records that use it are reserved. Other implementations of the Apple File System must not modify them.

There isn't an actual directory with this inode number.

Purgeable files have the `INODE_IS_PURGEABLE` flag set on the `internal_flags` field of `j_inode_val_t`.

MIN_USER_INO_NUM

The smallest inode number available for user content.

```
#define MIN_USER_INO_NUM 16
```

All inode numbers less than this value are reserved.

UNIFIED_ID_SPACE_MARK

The smallest inode number used by the system volume in a volume group.

```
#define UNIFIED_ID_SPACE_MARK 0x0800000000000000ULL
```

For more information, see `APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE`.

Extended Attributes Constants

Constants used with extended attributes.

```
#define XATTR_MAX_EMBEDDED_SIZE      3804
#define SYMLINK_EA_NAME              "com.apple.fs.symlink"
#define FIRMLINK_EA_NAME             "com.apple.fs.firmlink"
#define APFS_COW_EXEMPT_COUNT_NAME   "com.apple.fs.cow-exempt-file-count"
```

XATTR_MAX_EMBEDDED_SIZE

The largest size, in bytes, of an extended attribute whose value is stored directly in the record.

```
#define XATTR_MAX_EMBEDDED_SIZE 3804
```

For information about embedded values, see `j_xattr_val_t`.

`SYMLINK_EA_NAME`

The name of an extended attribute for a symbolic link whose value is the target file on the data volume.

```
#define SYMLINK_EA_NAME "com.apple.fs.symlink"
```

`FIRMLINK_EA_NAME`

The name of an extended attribute for a firm link whose value is the target file.

```
#define FIRMLINK_EA_NAME "com.apple.fs.firmlink"
```

`APFS_COW_EXEMPT_COUNT_NAME`

The number of files on the volume that don't use copy on write.

```
#define APFS_COW_EXEMPT_COUNT_NAME "com.apple.fs.cow-exempt-file-count"
```

Don't add this extended attribute or modify its value, but preserve the attribute if it already exists.

The inodes that are counted here have the `INODE_SNAPSHOT_COW_EXEMPTION` flag set. This number is used by Time Machine when making snapshots.

File-System Object Constants

No overview available.

```
#define OWNING_OBJ_ID_INVALID       ~0ULL
#define OWNING_OBJ_ID_UNKNOWN       ~1ULL

#define JOBJ_MAX_KEY_SIZE           832
#define JOBJ_MAX_VALUE_SIZE         3808

#define MIN_DOC_ID                  3
```

`MIN_DOC_ID`

The smallest document identifier available for user content.

```
#define MIN_DOC_ID 3
```

All document identifiers less than this value are reserved.

File Extent Constants

No overview available.

```
#define FEXT_CRYPTO_ID_IS_TWEAK     0x01
```

File Modes

The values used by the `mode` field of `j_inode_val_t` to indicate a file's mode.

```
typedef uint16_t    mode_t;

#define S_IFMT                      0170000

#define S_IFIFO                     0010000
#define S_IFCHR                     0020000
#define S_IFDIR                     0040000
#define S_IFBLK                     0060000
#define S_IFREG                     0100000
#define S_IFLNK                     0120000
#define S_IFSOCK                    0140000
#define S_IFWHT                     0160000
```

The names, values, and meanings of these constants are the same as the constants provided by `<sys/stat.h>`. These values are the same as the values defined in Directory Entry File Types, except for a bit shift.

mode_t

A file mode.

```
typedef uint16_t mode_t;
```

S_IFMT

The bit mask used to access the file type.

```
#define S_IFMT 0170000
```

S_IFIFO

A named pipe.

```
#define S_IFIFO 0010000
```

S_IFCHR

A character-special file.

```
#define S_IFCHR 0020000
```

S_IFDIR

A directory.

```
#define S_IFDIR 0040000
```

S_IFBLK

A block-special file.

```
#define S_IFBLK 0060000
```

S_IFREG

A regular file.

```
#define S_IFREG 0100000
```

S_IFLNK

A symbolic link.

```
#define S_IFLNK 0120000
```

S_IFSOCK

A socket.

```
#define S_IFSOCK 0140000
```

S_IFWHT

A whiteout.

```
#define S_IFWHT 0160000
```

Directory Entry File Types

Values used by the `flags` field of `j_drec_val_t` to indicate a directory entry's type.

```
#define DT_UNKNOWN                   0
#define DT_FIFO                      1
#define DT_CHR                       2
#define DT_DIR                       4
#define DT_BLK                       6
#define DT_REG                       8
#define DT_LNK                       10
#define DT_SOCK                      12
#define DT_WHT                       14
```

These values are the same as the values defined in File Modes, except for a bit shift.

DT_UNKNOWN

An unknown directory entry.

```
#define DT_UNKNOWN 0
```

DT_FIFO

A named pipe

```
#define DT_FIFO 1
```

DT_CHR

A character-special file.

```
#define DT_CHR 2
```

DT_DIR

A directory.

```
#define DT_DIR 4
```

DT_BLK

A block-special file.

```
#define DT_BLK 6
```

DT_REG

A regular file.

```
#define DT_REG 8
```

DT_LNK

A symbolic link.

```
#define DT_LNK 10
```

DT_SOCK

A socket.

```
#define DT_SOCK 12
```

DT_WHT

A whiteout.

```
#define DT_WHT 14
```

Data Streams

Short pieces of information like a file's name are stored inside the data structures that contain metadata. Data that's too large to store inline is stored separately, in a data stream. This includes the contents of files, and the value of some attributes.

j_phys_ext_key_t

The key half of a physical extent record.

```
struct j_phys_ext_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_phys_ext_key j_phys_ext_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the physical block address of the start of the extent. The type in the header is always APFS_TYPE_EXTENT.

j_phys_ext_val_t

The value half of a physical extent record.

```
struct j_phys_ext_val {
    uint64_t    len_and_kind;
    uint64_t    owning_obj_id;
    int32_t     refcnt;
} __attribute__((packed));
typedef struct j_phys_ext_val j_phys_ext_val_t;

#define PEXT_LEN_MASK               0x0fffffffffffffffULL
#define PEXT_KIND_MASK              0xf000000000000000ULL
#define PEXT_KIND_SHIFT             60
```

len_and_kind

A bit field that contains the length of the extent and its kind.

```
uint64_t len_and_kind;
```

The extent's length is a uint64_t value, accessed as len_and_kind & PEXT_LEN_MASK, and measured in blocks. The extent's kind is a j_obj_kinds value, accessed as (len_and_kind & PEXT_KIND_MASK) >> PEXT_KIND_SHIFT.

For a volume that has no snapshots, the kind is always APFS_KIND_NEW.

owning_obj_id

The identifier of the file system record that's using this extent.

```
uint64_t owning_obj_id;
```

If the owning record is an inode, this field contains the inode's private identifier (the `private_id` field of `j_inode_val_t`). If the owning record is an extended attribute, this field contains the extended attribute's record identifier (the identifier from the `hdr` field of `j_xattr_key_t`).

refcnt

The reference count.

```
int32_t refcnt;
```

The extent can be deleted when its reference count reaches zero.

PEXT_LEN_MASK

The bit mask used to access the extent length.

```
#define PEXT_LEN_MASK 0x0fffffffffffffffULL
```

PEXT_KIND_MASK

The bit mask used to access the extent kind.

```
#define PEXT_KIND_MASK 0xf000000000000000ULL
```

PEXT_KIND_SHIFT

The bit shift used to access the extent kind.

```
#define PEXT_KIND_SHIFT 60
```

j_file_extent_key_t

The key half of a file extent record.

```
struct j_file_extent_key {
    j_key_t     hdr;
    uint64_t    logical_addr;
} __attribute__((packed));
typedef struct j_file_extent_key j_file_extent_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always `APFS_TYPE_FILE_EXTENT`.

logical_addr

The offset within the file's data, in bytes, for the data stored in this extent.

```
uint64_t logical_addr;
```

j_file_extent_val_t

The value half of a file extent record.

```
struct j_file_extent_val {
    uint64_t    len_and_flags;
    uint64_t    phys_block_num;
    uint64_t    crypto_id;
} __attribute__((packed));
typedef struct j_file_extent_val j_file_extent_val_t;

#define J_FILE_EXTENT_LEN_MASK      0x00ffffffffffffffULL
#define J_FILE_EXTENT_FLAG_MASK     0xff00000000000000ULL
#define J_FILE_EXTENT_FLAG_SHIFT    56
```

len_and_flags

A bit field that contains the length of the extent and its flags.

```
uint64_t len_and_flags;
```

The extent's length is a `uint64_t` value, accessed as `len_and_kind & J_FILE_EXTENT_LEN_MASK`, and measured in bytes. The length must be a multiple of the block size defined by the `nx_block_size` field of `nx_superblock_t`. The extent's flags are accessed as `(len_and_kind & J_FILE_EXTENT_FLAG_MASK) >> J_FILE_EXTENT_FLAG_SHIFT`.

There are currently no flags defined.

phys_block_num

The physical block address that the extent starts at.

```
uint64_t phys_block_num;
```

crypto_id

The encryption key or the encryption tweak used in this extent.

```
uint64_t crypto_id;
```

If the `APFS_FS_ONEKEY` flag is set on the volume, this field contains the AES-XTS tweak value. Otherwise, this value matches the `obj_id` field of the `j_crypto_key_t` record that contains information about how this file extent is encrypted, including the per-file encryption key.

The default value for this field is the value of the `default_crypto_id` field of the `j_dstream_t` for the data stream that this extent is part of.

J_FILE_EXTENT_LEN_MASK

The bit mask used to access the extent length.

```
#define J_FILE_EXTENT_LEN_MASK 0x00ffffffffffffffULL
```

J_FILE_EXTENT_FLAG_MASK

The bit mask used to access the flags.

```
#define J_FILE_EXTENT_FLAG_MASK 0xff00000000000000ULL
```

J_FILE_EXTENT_FLAG_SHIFT

The bit shift used to access the flags.

```
#define J_FILE_EXTENT_FLAG_SHIFT 56
```

j_dstream_id_key_t

The key half of a directory-information record.

```
struct j_dstream_id_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_dstream_id_key j_dstream_id_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always APFS_TYPE_DSTREAM_ID.

j_dstream_id_val_t

The value half of a data stream record.

```
struct j_dstream_id_val {
    uint32_t    refcnt;
} __attribute__((packed));
typedef struct j_dstream_id_val j_dstream_id_val_t;
```

refcnt

The reference count.

```
uint32_t refcnt;
```

The data stream record can be deleted when its reference count reaches zero.

j_xattr_dstream_t

A data stream for extended attributes.

```
struct j_xattr_dstream {
    uint64_t    xattr_obj_id;
    j_dstream_t dstream;
};
typedef struct j_xattr_dstream j_xattr_dstream_t;
```

To access the data in the stream, read the object identifier and then find the corresponding extents.

xattr_obj_id

The identifier for the data stream.

```
uint64_t xattr_obj_id;
```

This field contains the record identifier of the data stream that owns this record.

dstream

Information about the data stream.

```
j_dstream_t dstream;
```

j_dstream_t

Information about a data stream.

```
struct j_dstream {
    uint64_t    size;
    uint64_t    alloced_size;
    uint64_t    default_crypto_id;
    uint64_t    total_bytes_written;
    uint64_t    total_bytes_read;
} __attribute__((aligned(8),packed));
typedef struct j_dstream j_dstream_t;
```

This structure is used inside j_xattr_dstream_t.

size

The size, in bytes, of the data.

```
uint64_t size;
```

alloced_size

The total space allocated for the data stream, including any unused space.

```
uint64_t alloced_size;
```

default_crypto_id

The default encryption key or encryption tweak used in this data stream.

```
uint64_t default_crypto_id;
```

This value matches the `obj_id` field in the `j_key_t` key that corresponds to a `j_crypto_val_t` value. For a volume that uses software encryption, the value of this field is always `CRYPTO_SW_ID`.

This value is used as the default value by file extents (`j_file_extent_val_t`) that make up this data stream.

total_bytes_written

The total number of bytes that have been written to this data stream.

```
uint64_t total_bytes_written;
```

The value of this field increases every time a write operation occurs. This value is allowed to overflow and restart from zero.

total_bytes_read

The total number of bytes that have been read from this data stream.

```
uint64_t total_bytes_read;
```

The value of this field increases every time a read operation occurs. This value is allowed to overflow and restart from zero.

Extended Fields

Directory entries and inodes use extended fields to store a dynamically extensible set of member fields.

To determine whether a directory entry or an inode has any extended fields, find the table of contents entry for the file-system record, and then compare the recorded size to the size of the structure. For example:

```
kvloc_t toc_entry = /* assume this exists */
if (toc_entry.v.len == sizeof(j_drec_val_t)) {
    // no extended fields
} else {
    // at least one extended field
}
```

Both `j_drec_val_t` and `j_inode_val_t` have an `xfields` field that contains several kinds of data, stored one after another, ordered as follows:

1. An instance of `xf_blob_t`, which tells you how many extended fields there are, and how many bytes they take up on disk.
2. An array of instances of `x_field_t`, one for each extended field, which tells you the field's type and size.
3. An array of extended-field data, aligned to eight-byte boundaries.

The arrays of extended-field metadata (`x_field_t`) and extended-field data are stored in the same order. The extended-field data's type depends on the field. For a list of field types, see Extended-Field Types.

xf_blob_t

A collection of extended attributes.

```
struct xf_blob {
    uint16_t    xf_num_exts;
    uint16_t    xf_used_data;
    uint8_t     xf_data[];
};
typedef struct xf_blob xf_blob_t;
```

Directory entries (`j_drec_val_t`) and inodes (`j_inode_val_t`) use this data type to store their extended fields.

xf_num_exts

The number of extended attributes.

```
uint16_t xf_num_exts;
```

xf_used_data

The amount of space, in bytes, used to store the extended attributes.

```
uint16_t xf_used_data;
```

This total includes both the space used to store metadata, as instances of `x_field_t`, and values.

xf_data[]

The extended fields.

```
uint8_t xf_data[];
```

This field contains an array of instances of x_field_t, followed by the extended field data.

x_field_t

An extended field's metadata.

```
struct x_field {
    uint8_t     x_type;
    uint8_t     x_flags;
    uint16_t    x_size;
};
typedef struct x_field x_field_t;
```

This type is used by xf_blob_t to store an array of extended fields. Within the array, each extended field must have a unique type.

The extended field's data is stored outside of this structure, as part of the space set aside by the directory entry or inode.

x_type

The extended field's data type.

```
uint8_t x_type;
```

For possible values, see Extended-Field Types.

x_flags

The extended field's flags.

```
uint8_t x_flags;
```

For the values used in this bit field, see Extended-Field Flags.

x_size

The size, in bytes, of the data stored in the extended field.

```
uint16_t x_size;
```

Extended-Field Types

Values used by the x_type field of x_field_t to indicate an extended field's type.

```
#define DREC_EXT_TYPE_SIBLING_ID        1

#define INO_EXT_TYPE_SNAP_XID           1
```

```
#define INO_EXT_TYPE_DELTA_TREE_OID     2
#define INO_EXT_TYPE_DOCUMENT_ID        3
#define INO_EXT_TYPE_NAME               4
#define INO_EXT_TYPE_PREV_FSIZE         5
#define INO_EXT_TYPE_RESERVED_6         6
#define INO_EXT_TYPE_FINDER_INFO        7
#define INO_EXT_TYPE_DSTREAM            8
#define INO_EXT_TYPE_RESERVED_9         9
#define INO_EXT_TYPE_DIR_STATS_KEY      10
#define INO_EXT_TYPE_FS_UUID            11
#define INO_EXT_TYPE_RESERVED_12        12
#define INO_EXT_TYPE_SPARSE_BYTES       13
#define INO_EXT_TYPE_RDEV               14
#define INO_EXT_TYPE_PURGEABLE_FLAGS    15
#define INO_EXT_TYPE_ORIG_SYNC_ROOT_ID  16
```

DREC_EXT_TYPE_SIBLING_ID

The sibling identifier for a directory record (`uint64_t`).

```
#define DREC_EXT_TYPE_SIBLING_ID 1
```

The corresponding sibling-link record has the same identifier in the `sibling_id` field of `j_sibling_key_t`.

This extended field is used only for hard links.

INO_EXT_TYPE_SNAP_XID

The transaction identifier for a snapshot (`xid_t`).

```
#define INO_EXT_TYPE_SNAP_XID 1
```

INO_EXT_TYPE_DELTA_TREE_OID

The virtual object identifier of the file-system tree that corresponds to a snapshot's extent delta list (`oid_t`).

```
#define INO_EXT_TYPE_DELTA_TREE_OID 2
```

The tree object's subtype is always `OBJECT_TYPE_FSTREE`.

INO_EXT_TYPE_DOCUMENT_ID

The file's document identifier (`uint32_t`).

```
#define INO_EXT_TYPE_DOCUMENT_ID 3
```

The document identifier lets applications keep track of the document during operations like atomic save, where one folder replaces another. The document identifier remains associated with the full path, not just with the inode that's currently at that path. Implementations of Apple File System must preserve the document identifier when the inode at that path is replaced.

Both documents that are stored as a bundle and documents that are stored as a single file can have a document identifier assigned.

Valid document identifiers are greater than `MIN_DOC_ID` and less than `UINT32_MAX - 1`. For the next document identifier that will be assigned, see the `apfs_next_doc_id` field of `apfs_superblock_t`.

INO_EXT_TYPE_NAME

The name of the file, represented as a null-terminated UTF-8 string.

```
#define INO_EXT_TYPE_NAME 4
```

This extended field is used only for hard links: The name stored in the inode is the name of the primary link to the file, and the name of the hard link is stored in this extended field.

INO_EXT_TYPE_PREV_FSIZE

The file's previous size (`uint64_t`).

```
#define INO_EXT_TYPE_PREV_FSIZE 5
```

This extended field is used for recovering after a crash. If it's set on an inode, truncate the file back to the size contained in this field.

INO_EXT_TYPE_RESERVED_6

Reserved.

```
#define INO_EXT_TYPE_RESERVED_6 6
```

Don't create extended fields of this type in your own code. Preserve the value of any extended fields of this type.

INO_EXT_TYPE_FINDER_INFO

Opaque data stored and used by Finder (32 bytes).

```
#define INO_EXT_TYPE_FINDER_INFO 7
```

INO_EXT_TYPE_DSTREAM

A data stream (`j_dstream_t`).

```
#define INO_EXT_TYPE_DSTREAM 8
```

INO_EXT_TYPE_RESERVED_9

Reserved.

```
#define INO_EXT_TYPE_RESERVED_9 9
```

Don't create extended fields of this type. When you modify an existing volume, preserve the contents of any extended fields of this type.

INO_EXT_TYPE_DIR_STATS_KEY

Statistics about a directory (`j_dir_stats_val_t`).

```
#define INO_EXT_TYPE_DIR_STATS_KEY 10
```

 |

INO_EXT_TYPE_FS_UUID

The UUID of a file system that's automatically mounted in this directory (uuid_t).

```
#define INO_EXT_TYPE_FS_UUID 11
```

This value matches the value of the apfs_vol_uuid field of apfs_superblock_t.

INO_EXT_TYPE_RESERVED_12

Reserved.

```
#define INO_EXT_TYPE_RESERVED_12 12
```

Don't create extended fields of this type. If you find an object of this type in production, file a bug against the Apple File System implementation.

INO_EXT_TYPE_SPARSE_BYTES

The number of sparse bytes in the data stream (uint64_t).

```
#define INO_EXT_TYPE_SPARSE_BYTES 13
```

INO_EXT_TYPE_RDEV

The device identifier for a block- or character-special device (uint32_t).

```
#define INO_EXT_TYPE_RDEV 14
```

This extended field stores the same information as the st_rdev field of the stat structure defined in <sys/stat.h>.

INO_EXT_TYPE_PURGEABLE_FLAGS

Information about a purgeable file.

```
#define INO_EXT_TYPE_PURGEABLE_FLAGS 15
```

The value of this extended field is reserved. Don't create new extended fields of this type. When duplicating a file or directory, omit this extended field from the new copy.

Purgeable files have the INODE_IS_PURGEABLE flag set on the internal_flags field of j_inode_val_t.

INO_EXT_TYPE_ORIG_SYNC_ROOT_ID

The inode number of the sync-root hierarchy that this file originally belonged to.

```
#define INO_EXT_TYPE_ORIG_SYNC_ROOT_ID 16
```

The specified inode always has the INODE_IS_SYNC_ROOT flag set.

Extended-Field Flags

The flags used by an extended field's metadata.

```
#define XF_DATA_DEPENDENT           0x0001
#define XF_DO_NOT_COPY              0x0002
#define XF_RESERVED_4               0x0004
#define XF_CHILDREN_INHERIT         0x0008
#define XF_USER_FIELD               0x0010
#define XF_SYSTEM_FIELD             0x0020
#define XF_RESERVED_40              0x0040
#define XF_RESERVED_80              0x0080
```

These flags are used by the `x_flags` field of `x_field_t`.

XF_DATA_DEPENDENT

The data in this extended field depends on the file's data.

```
#define XF_DATA_DEPENDENT 0x0001
```

When the file data changes, this extended field must be updated to match the new data. If it's not possible to update the field — for example, because the Apple File System implementation doesn't recognize the field's type — the field must be removed.

XF_DO_NOT_COPY

When copying this file, omit this extended field from the copy.

```
#define XF_DO_NOT_COPY 0x0002
```

XF_RESERVED_4

Reserved.

```
#define XF_RESERVED_4 0x0004
```

Don't set this flag, but preserve it if it's already set.

XF_CHILDREN_INHERIT

When creating a new entry in this directory, copy this extended field to the new directory entry.

```
#define XF_CHILDREN_INHERIT 0x0008
```

XF_USER_FIELD

This extended field was added by a user-space program.

```
#define XF_USER_FIELD 0x0010
```

XF_SYSTEM_FIELD

This extended field was added by the kernel, by the implementation of Apple File System, or by another system component.

```
#define XF_SYSTEM_FIELD 0x0020
```

Extended fields with this flag set can’t be removed or modified by a program running in user space.

XF_RESERVED_40

Reserved.

```
#define XF_RESERVED_40 0x0040
```

Don’t set this flag, but preserve it if it’s already set.

XF_RESERVED_80

Reserved.

```
#define XF_RESERVED_80 0x0080
```

Don’t set this flag, but preserve it if it’s already set.

Siblings

Hard links that all refer to the same inode are called *siblings*. Each sibling has its own identifier that's used instead of the shared inode number when siblings need to be distinguished. For example, some Carbon APIs in macOS use sibling identifiers.

The sibling whose identifier is the lowest number is called the *primary link*. The other siblings copy various properties of the primary link, as discussed in `j_inode_val_t`.

You use sibling links and sibling maps to convert between sibling identifiers and inode numbers. Sibling-link records let you find all the hard links whose target is a given inode. Sibling-map records let you find the target inode of a given hard link.

j_sibling_key_t

The key half of a sibling-link record.

```
struct j_sibling_key {
    j_key_t     hdr;
    uint64_t    sibling_id;
} __attribute__((packed));
typedef struct j_sibling_key j_sibling_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier, that is, its inode number. The type in the header is always `APFS_TYPE_SIBLING_LINK`.

sibling_id

The sibling's unique identifier.

```
uint64_t sibling_id;
```

This value matches the object identifier for the sibling map record (`j_sibling_key_t`).

j_sibling_val_t

The value half of a sibling-link record.

```
struct j_sibling_val {
    uint64_t    parent_id;
    uint16_t    name_len;
    uint8_t     name[0];
} __attribute__((packed));
typedef struct j_sibling_val j_sibling_val_t;
```

parent_id

The object identifier for the inode that's the parent directory.

```
uint64_t parent_id;
```

name_len

The length of the name, including the final null character (U+0000).

```
uint16_t name_len;
```

name

The name, represented as a null-terminated UTF-8 string.

```
uint8_t name[0];
```

j_sibling_map_key_t

The key half of a sibling-map record.

```
struct j_sibling_map_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_sibling_map_key j_sibling_map_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the sibling's unique identifier, which matches the `sibling_id` field of `j_sibling_key_t`. The type in the header is always `APFS_TYPE_SIBLING_MAP`.

j_sibling_map_val_t

The value half of a sibling-map record.

```
struct j_sibling_map_val {
    uint64_t  file_id;
} __attribute__((packed));
typedef struct j_sibling_map_val j_sibling_map_val_t;
```

file_id

The inode number of the underlying file.

```
uint64_t file_id;
```

Snapshot Metadata

Snapshots let you get a stable, read-only copy of the filesystem at a given point in time — for example, while updating a backup of the entire drive. Snapshots are designed to be fast and inexpensive to create; however, deleting a snapshot involves more work.

j_snap_metadata_key_t

The key half of a record containing metadata about a snapshot.

```
struct j_snap_metadata_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_snap_metadata_key j_snap_metadata_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the snapshot's transaction identifier. The type in the header is always APFS_TYPE_SNAP_METADATA.

j_snap_metadata_val_t

The value half of a record containing metadata about a snapshot.

```
struct j_snap_metadata_val {
    oid_t       extentref_tree_oid;
    oid_t       sblock_oid;
    uint64_t    create_time;
    uint64_t    change_time;
    uint64_t    inum;
    uint32_t    extentref_tree_type;
    uint32_t    flags;
    uint16_t    name_len;
    uint8_t     name[0];
} __attribute__((packed));
typedef struct j_snap_metadata_val j_snap_metadata_val_t;
```

extentref_tree_oid

The physical object identifier of the B-tree that stores extents information.

```
oid_t extentref_tree_oid;
```

sblock_oid

The physical object identifier of the volume superblock.

```
oid_t sblock_oid;
```

`create_time`

The time that this snapshot was created.

```
uint64_t create_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

`change_time`

The time that this snapshot was last modified.

```
uint64_t change_time;
```

This timestamp is represented as the number of nanoseconds since January 1, 1970 at 0:00 UTC, disregarding leap seconds.

`inum`

No overview available.

```
uint64_t inum;
```

`extentref_tree_type`

The type of the B-tree that stores extents information.

```
uint32_t extentref_tree_type;
```

`flags`

A bit field that contains additional information about a snapshot metadata record.

```
uint32_t flags;
```

For the values used in this bit field, see `snap_meta_flags`.

`name_len`

The length of the snapshot's name, including the final null character (U+0000).

```
uint16_t name_len;
```

`name`

The snapshot's name, represented as a null-terminated UTF-8 string.

```
uint8_t name[0];
```

j_snap_name_key_t

The key half of a snapshot name record.

```
struct j_snap_name_key {
    j_key_t     hdr;
    uint16_t    name_len;
    uint8_t     name[0];
} __attribute__((packed));
typedef struct j_snap_name_key j_snap_name_key_t;
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is always `~0ULL`. The type in the header is always `APFS_TYPE_SNAP_NAME`.

name_len

The length of the extended attribute's name, including the final null character (U+0000).

```
uint16_t name_len;
```

name

The extended attribute's name, represented as a null-terminated UTF-8 string.

```
uint8_t name[0];
```

j_snap_name_val_t

The value half of a snapshot name record.

```
struct j_snap_name_val {
    xid_t       snap_xid;
} __attribute__((packed));
typedef struct j_snap_name_val j_snap_name_val_t;
```

snap_xid

The last transaction identifier included in the snapshot.

```
xid_t snap_xid;
```

snap_meta_flags

No overview available.

```
typedef enum {
    SNAP_META_PENDING_DATALESS  = 0x00000001,
    SNAP_META_MERGE_IN_PROGRESS = 0x00000002,
```

```
} snap_meta_flags;
```

snap_meta_ext_obj_phys_t

Additional metadata about snapshots.

```
struct snap_meta_ext_obj_phys {
    obj_phys_t      smeop_o;
    snap_meta_ext_t smeop_sme;
}
typedef struct snap_meta_ext_obj_phys_t;
```

smeop_o

No overview available.

```
obj_phys_t smeop_o;
```

smeop_sme

No overview available.

```
snap_meta_ext_t smeop_sme;
```

snap_meta_ext_t

No overview available.

```
typedef struct snap_meta_ext {
    uint32_t    sme_version;

    uint32_t    sme_flags;
    xid_t       sme_snap_xid;
    uuid_t      sme_uuid;

    uint64_t    sme_token;

} __attribute__((packed))
typedef struct snap_meta_ext snap_meta_ext_t;
```

sme_version

The version of this structure.

```
uint32_t sme_version;
```

sme_flags

```
uint32_t sme_flags;
```

sme_snap_xid

The snapshot's transaction identifier.

```
xid_t sme_snap_xid;
```

sme_uuid

The snapshot's UUID.

```
uuid_t sme_uuid;
```

sme_token

Opaque metadata.

```
uint64_t sme_token;
```

B-Trees

The B-trees used in Apple File System are implemented using the `btree_node_phys_t` structure to represent a node. The same structure is used for all nodes in a tree. Within a node, storage is divided into several areas:

- Information about the node
- The table of contents, which lists the location of keys and values
- Storage for the keys
- Storage for the values
- Information about the entire tree

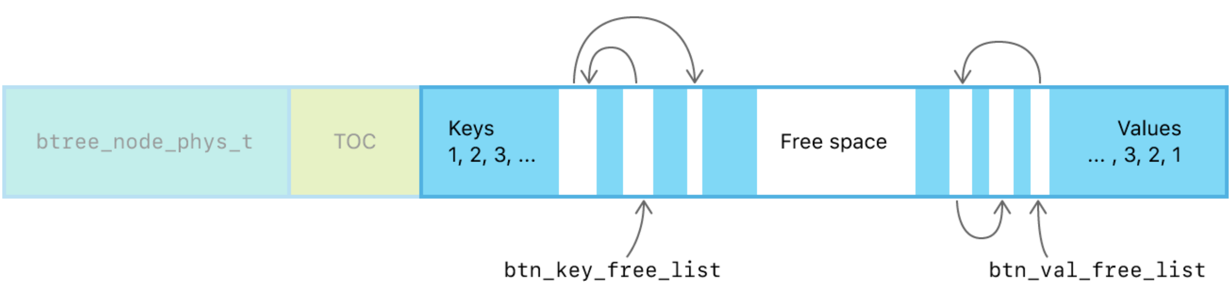
The figure below shows the storage areas of a typical root node.

The instance of `btree_node_phys_t` stores information about this B-tree node, like its flags and the location of its keys, and is always located at the beginning of the block. For a root node, an instance of `btree_info_t` is located at the end of the block, and contains information like the sizes of keys and values, the total number of keys in the tree, and the number of nodes in the tree. Nonroot nodes omit `btree_info_t`. The rest of the block (the `btn_data` field of `btree_node_phys_t`) is organized dynamically.

Compared to other B-tree implementations, this data structure has some unique characteristics. Traversal is always done from the root node because nodes don't have parent or sibling pointers. All values are stored in leaf nodes, which makes these B+ trees, and the values in nonleaf nodes are object identifiers of child nodes. The keys, values, or both can be of variable size; if the keys and values of a node are both fixed in size, some optimizations for the table of contents are possible.

Keys and Values

The keys and values are stored starting at opposite ends of the B-tree node's storage area, with free space that's available for new keys or values in the available portion of the storage area between them. The key and value areas grow toward each other into their shared free space. Free space within the key area and within the value area is organized using a free list. For example, free space appears outside the shared free space when an entry is removed from a B-tree. The figure below shows free space for keys and values in a typical nonroot node.



The locations of keys and values are stored as offsets, which uses less on-disk space than storing the full location. The offset to a key is counted from the beginning of the key area to the beginning of the key. The offset to a value is counted from the end of the value area to the beginning of the value.

Keys and value are normally aligned to eight-byte boundaries when stored. The length recorded for a key or value in the table of contents omits any padding needed for alignment. If the BTREE_KV_NONALIGNED flag is set, keys and values are stored without padding.

If the BTREE_ALLOW_GHOSTS flag is set on the B-tree, the tree can contain keys that have no value.

Table of Contents

The table of contents stores the location of each key and value that form a key-value pair.

If the BTNODE_FIXED_KV_SIZE flag is set, the table of contents stores only the offsets for keys and values. Otherwise, it stores both their offsets and lengths.

Free space within the table of contents is located at the end. If there's no free space remaining, but a new entry is needed, the table of contents area must be expanded. The entire key area is shifted to make space available, using some of the shared free space for key space, and some space from the beginning of the key space for the table of contents. Because the offset to a key is counted relative to the beginning of the key area, moving the entire key area doesn't invalidate any of these offsets. Likewise, when the table of contents has too much unused space, it shrinks, and the key area is shifted into the space from the table of contents. Apple's implementation uses BTREE_TOC_ENTRY_INCREMENT and BTREE_TOC_ENTRY_MAX_UNUSED to determine when to expand or shrink the table of contents.

> **Note**
>
> When the BTNODE_FIXED_KV_SIZE flag is set, Apple's implementation allocates enough space for the table of contents to avoid the need to expand it later. This is possible because the maximum number of entries is known, as well as the size of an entry. However, if the BTREE_ALLOW_GHOSTS flag is also set, the table of contents might still need to expand.

Key Comparison

The entries in the table of contents are sorted by key. The comparison function used for sorting depends on the key's type. Object map B-trees are sorted by object identifier and then by transaction identifier. Free queue B-trees are sorted by transaction identifier and then by physical address. File-system records are sorted according to the rules listed in File-System Objects.

btree_node_phys_t

A B-tree node.

```
struct btree_node_phys {
    obj_phys_t  btn_o;
    uint16_t    btn_flags;
    uint16_t    btn_level;
    uint32_t    btn_nkeys;
    nloc_t      btn_table_space;
    nloc_t      btn_free_space;
    nloc_t      btn_key_free_list;
    nloc_t      btn_val_free_list;
    uint64_t    btn_data[];
```

```
};
typedef struct btree_node_phys btree_node_phys_t;
```

The locations of the key and value areas aren't stored explicitly. The key area begins after the end of the table of contents and ends before the start of the shared free space. The value area begins after the end of shared free space and ends at the end of the B-tree node (for nonroot nodes) or before the instance of `btree_info_t` that's at the end of a root node.

btn_o

The object's header.

```
obj_phys_t btn_o;
```

btn_flags

The B-tree node's flags.

```
uint16_t btn_flags;
```

For the values used in this bit field, see B-Tree Node Flags.

btn_level

The number of child levels below this node.

```
uint16_t btn_level;
```

For example, the value of this field is zero for a leaf node and one for the immediate parent of a leaf node. Likewise, the height of a tree is one plus the value of this field on the tree's root node.

btn_nkeys

The number of keys stored in this node.

```
uint32_t btn_nkeys;
```

btn_table_space

The location of the table of contents.

```
nloc_t btn_table_space;
```

The offset for the table of contents is counted from the beginning of the node's `btn_data` field to the beginning of the table of contents.

If the `BTNODE_FIXED_KV_SIZE` flag is set, the table of contents is an array of instances of `kvoff_t`; otherwise, it's an array of instances of `kvloc_t`.

btn_free_space

The location of the shared free space for keys and values.

```
nloc_t btn_free_space;
```

The location's offset is counted from the beginning of the key area to the beginning of the free space.

btn_key_free_list

A linked list that tracks free key space.

```
nloc_t btn_key_free_list;
```

The offset from the beginning of the key area to the first available space for a key is stored in the `off` field, and the total amount of free key space is stored in the `len` field. Each free space stores an instance of `nloc_t` whose `len` field indicates the size of that free space and whose `off` field contains the location of the next free space.

btn_val_free_list

A linked list that tracks free value space.

```
nloc_t btn_val_free_list;
```

The offset from the end of the value area to the first available space for a value is stored in the `off` field, and the total amount of free value space is stored in the `len` field. Each free space stores an instance of `nloc_t` whose `len` field indicates the size of that free space and whose `off` field contains the location of the next free space.

btn_data

The node's storage area.

```
uint64_t btn_data[];
```

This area contains the table of contents, keys, free space, and values. A root node also has as an instance of `btree_info_t` at the end of its storage area. For more information, see B-trees.

btree_info_fixed_t

Static information about a B-tree.

```
struct btree_info_fixed {
    uint32_t    bt_flags;
    uint32_t    bt_node_size;
    uint32_t    bt_key_size;
    uint32_t    bt_val_size;
};
typedef struct btree_info_fixed btree_info_fixed_t;
```

This information doesn't change over time as the B-tree is modified. It's stored separately from the rest of the information in `btree_info_t`, which does change, to enable this information to be cached more easily.

bt_flags

The B-tree's flags.

```
uint32_t bt_flags;
```

For the values used in this bit field, see B-Tree Flags.

bt_node_size

The on-disk size, in bytes, of a node in this B-tree.

```
uint32_t bt_node_size;
```

Leaf nodes, nonleaf nodes, and the root node are all the same size.

bt_key_size

The size of a key, or zero if the keys have variable size.

```
uint32_t bt_key_size;
```

If this field has a value of zero, the `btn_flags` field of instances of `btree_node_phys_t` in this tree must not include `BTNODE_FIXED_KV_SIZE`.

bt_val_size

The size of a value, or zero if the values have variable size.

```
uint32_t bt_val_size;
```

If this field has a value of zero, the `btn_flags` field of instances of `btree_node_phys_t` for leaf nodes in this tree must not include `BTNODE_FIXED_KV_SIZE`. Nonleaf nodes in a tree with variable-size values include `BTNODE_FIXED_KV_SIZE`, because the values stored in those nodes are the object identifiers of their child nodes, and object identifiers have a fixed size.

btree_info_t

Information about a B-tree.

```
struct btree_info {
    btree_info_fixed_t  bt_fixed;
    uint32_t            bt_longest_key;
    uint32_t            bt_longest_val;
    uint64_t            bt_key_count;
    uint64_t            bt_node_count;
};
typedef struct btree_info btree_info_t;
```

This information appears only in a root node, stored at the end of the node.

btree_info_fixed_t

Information about the B-tree that doesn't change over time.

```
btree_info_fixed_t bt_fixed;
```

bt_longest_key

The length, in bytes, of the longest key that has ever been stored in the B-tree.

```
uint32_t bt_longest_key;
```

bt_longest_val

The length, in bytes, of the longest value that has ever been stored in the B-tree.

```
uint32_t bt_longest_val;
```

bt_key_count

The number of keys stored in the B-tree.

```
uint64_t bt_key_count;
```

bt_node_count

The number of nodes stored in the B-tree.

```
uint64_t bt_node_count;
```

btn_index_node_val_t

The value used by hashed B-trees for nonleaf nodes.

```
struct btn_index_node_val {
    oid_t       binv_child_oid;
    uint8_t     binv_child_hash[BTREE_NODE_HASH_SIZE_MAX];
};
typedef struct btn_index_node_val btn_index_node_val_t;

#define BTREE_NODE_HASH_SIZE_MAX 64
```

For nonhashed B-trees, instead of using this structure, the values are instances of `oid_t`. Because this structure's `oid_t` field comes first, code that's expecting only the object identifier of the child node as the B-tree value is still able to read the hashed B-tree by ignoring the hashes.

binv_child_oid

The object identifier of the child node.

```
oid_t binv_child_oid;
```

binv_child_hash

The hash of the child node.

```
uint8_t binv_child_hash[BTREE_NODE_HASH_SIZE_MAX];
```

The hash algorithm used by this tree determines the length of the hash. See the `im_hash_type` field of `integrity_meta_phys_t`, and the `hash_size` field of `j_file_data_hash_val_t`.

To compute the hash, use the entire child node object as the input for the hash algorithm specified for this tree. If the output from that hash algorithm is smaller than the `BTREE_NODE_HASH_SIZE_MAX` bytes, treat the remaining bytes as padding — set them to zero when you create a new node, and preserve their value when you modify an existing node.

BTREE_NODE_HASH_SIZE_MAX

The maximum length of a hash that can be stored in this structure.

```
#define BTREE_NODE_HASH_SIZE_MAX 64
```

This value is the same as APFS_HASH_MAX_SIZE.

nloc_t

A location within a B-tree node.

```
struct nloc {
    uint16_t    off;
    uint16_t    len;
};
typedef struct nloc nloc_t;

#define BTOFF_INVALID               0xffff
```

off

The offset, in bytes.

```
uint16_t off;
```

Depending on the data type that contains this location, the offset is either implicitly positive or negative, and is counted starting at different points in the B-tree node.

len

The length, in bytes.

```
uint16_t len;
```

BTOFF_INVALID

An invalid offset.

```
#define BTOFF_INVALID 0xffff
```

This value is stored in the `off` field of nloc_t to indicate that there's no offset. For example, the last entry in a free list has no entry after it, so it uses this value for its `off` field.

kvloc_t

The location, within a B-tree node, of a key and value.

```
struct kvloc {
    nloc_t      k;
    nloc_t      v;
};
typedef struct kvloc kvloc_t;
```

The B-tree node's table of contents uses this structure when the keys and values are not both fixed in size.

`nloc_t`

The location of the key.

```
nloc_t k;
```

`nloc_t`

The location of the value.

```
nloc_t v;
```

kvoff_t

The location, within a B-tree node, of a fixed-size key and value.

```
struct kvoff {
    uint16_t    k;
    uint16_t    v;
};
typedef struct kvoff kvoff_t;
```

The B-tree node's table of contents uses this structure when the keys and values are both fixed in size. The meaning of the offsets stored in this structure's `k` and `v` fields is the same as the meaning of the `off` field in an instance of `nloc_t`. This structure doesn't have a field that's equivalent to the `len` field of `nloc_t` — the key and value lengths are always the same, and omitting them from the table of contents saves space.

`k`

The offset of the key.

```
uint16_t k;
```

`v`

The offset of the value.

```
uint16_t v;
```

B-Tree Flags

The flags used to describe configuration options for a B-tree.

```
#define BTREE_UINT64_KEYS           0x00000001
#define BTREE_SEQUENTIAL_INSERT     0x00000002
#define BTREE_ALLOW_GHOSTS          0x00000004
#define BTREE_EPHEMERAL             0x00000008
#define BTREE_PHYSICAL              0x00000010
#define BTREE_NONPERSISTENT         0x00000020
#define BTREE_KV_NONALIGNED         0x00000040
#define BTREE_HASHED                0x00000080
```

```
#define BTREE_NOHEADER              0x00000100
```

BTREE_UINT64_KEYS

Code that works with the B-tree should enable optimizations to make comparison of keys fast.

```
#define BTREE_UINT64_KEYS 0x00000001
```

This is a hint used by Apple's implementation.

BTREE_SEQUENTIAL_INSERT

Code that works with the B-tree should enable optimizations to keep the B-tree compact during sequential insertion of entries.

```
#define BTREE_SEQUENTIAL_INSERT 0x00000002
```

This is a hint used by Apple's implementation.

Normally, nodes are split in half when they become almost full. With this flag set, a new node is added to provide the needed space, instead of splitting a node that's almost full. This yields a tree with nodes that are almost full instead of nodes that are about half full.

BTREE_ALLOW_GHOSTS

The table of contents is allowed to contain keys that have no corresponding value.

```
#define BTREE_ALLOW_GHOSTS 0x00000004
```

In the table of contents, a ghost is indicated by a value whose location offset is BTOFF_INVALID.

The meaning of a ghost depends on context — it can indicate a key that has been deleted and should be ignored, or a key whose value is implicit from context. For example, in the space manager's free queue, a ghost indicates a free extent that's one block long.

Using ghosts to store an implicit value allows more entries to be stored in some circumstances because no space in the value area is used by the ghost.

BTREE_EPHEMERAL

The nodes in the B-tree use ephemeral object identifiers to link to child nodes.

```
#define BTREE_EPHEMERAL 0x00000008
```

If this flag is set, BTREE_PHYSICAL must not be set. If neither flag is set, nodes in the B-tree use virtual object identifiers to link to their child nodes.

BTREE_PHYSICAL

The nodes in the B-tree use physical object identifiers to link to child nodes.

```
#define BTREE_PHYSICAL 0x00000010
```

If this flag is set, BTREE_EPHEMERAL must not be set. If neither flag is set, nodes in the B-tree use virtual object identifiers to link to their child nodes.

BTREE_NONPERSISTENT

The B-tree isn't persisted across unmounting.

```
#define BTREE_NONPERSISTENT 0x00000020
```

This flag is valid only when BTREE_EPHEMERAL is also set, and only on in-memory B-trees.

BTREE_KV_NONALIGNED

The keys and values in the B-tree aren't required to be aligned to eight-byte boundaries.

```
#define BTREE_KV_NONALIGNED 0x00000040
```

Aligning to eight-byte boundaries avoids unaligned reads on 64-bit platforms, which improves performance, but wastes space on disk for structures whose size isn't a multiple of eight bytes.

BTREE_HASHED

The nonleaf nodes of this B-tree store a hash of their child nodes.

```
#define BTREE_HASHED 0x00000080
```

If this flag is set, all nodes of this B-tree have the BTNODE_HASHED flag set.

The hash is stored in the `binv_child_hash` field of `btn_index_node_val_t`.

BTREE_NOHEADER

The nodes of this B-tree are stored without object headers.

```
#define BTREE_NOHEADER 0x00000100
```

If this flag is set, all nodes of this B-tree have the BTNODE_NOHEADER flag set.

B-Tree Table of Contents Constants

Constants used in managing the size of the table of contents in a B-tree node.

```
#define BTREE_TOC_ENTRY_INCREMENT    8
#define BTREE_TOC_ENTRY_MAX_UNUSED  (2 * BTREE_TOC_ENTRY_INCREMENT)
```

These values are used by Apple's implementation; other implementations can choose different values. If you don't use these values, profile your implementation to determine the performance impact of your chosen values.

BTREE_TOC_ENTRY_INCREMENT

The number of entries that are added or removed when changing the size of the table of contents.

```
#define BTREE_TOC_ENTRY_INCREMENT 8
```

BTREE_TOC_ENTRY_MAX_UNUSED

The maximum allowed number of unused entries in the table of contents.

```
#define BTREE_TOC_ENTRY_MAX_UNUSED  (2 * BTREE_TOC_ENTRY_INCREMENT)
```

B-Tree Node Flags

The flags used with a B-tree node.

```
#define BTNODE_ROOT                 0x0001
#define BTNODE_LEAF                 0x0002

#define BTNODE_FIXED_KV_SIZE        0x0004
#define BTNODE_HASHED               0x0008
#define BTNODE_NOHEADER             0x0010

#define BTNODE_CHECK_KOFF_INVAL     0x8000
```

BTNODE_ROOT

The B-tree node is a root node.

```
#define BTNODE_ROOT 0x0001
```

If this flag is set, the node's object type is OBJECT_TYPE_BTREE. If this is the tree's only node, both BTNODE_ROOT and BTNODE_LEAF are set. Otherwise, the BTNODE_LEAF flag must not be set.

BTNODE_LEAF

The B-tree node is a leaf node.

```
#define BTNODE_LEAF 0x0002
```

If this is the tree's only node, the node object's type is OBJECT_TYPE_BTREE, and both BTNODE_ROOT and BTNODE_LEAF are set. Otherwise, the node's object type is OBJECT_TYPE_BTREE_NODE, and the BTNODE_ROOT flag must not be set.

BTNODE_FIXED_KV_SIZE

The B-tree node has keys and values of a fixed size, and the table of contents omits their lengths.

```
#define BTNODE_FIXED_KV_SIZE 0x0004
```

If the keys and values both have a fixed size, this flag must be set.

Within the same B-tree, it's valid to have a mix of nodes that have this flag set and nodes that don't. For example, consider a B-tree with fixed-sized keys and variable-sized values. Leaf nodes in that tree don't have this flag set because of the variable-sized values. However, nonleaf nodes in in the same tree *do* have this flag set. The values stored in nonleaf nodes are object identifiers, which *are* fixed-sized values; therefore, this flag can be applied to nonleaf nodes of any tree with fixed-size keys.

BTNODE_HASHED

The B-tree node contains child hashes.

```
#define BTNODE_HASHED 0x0008
```

This flag is valid only on B-trees that have the BTREE_HASHED flag. You can this flag on a leaf node, for consistency with the nonleaf nodes in the same tree, but it doesn't mean anything there and is ignored.

If this flag isn't set, the `binv_child_hash` field of `btn_index_node_val_t` is unused.

BTNODE_NOHEADER

The B-tree node is stored without an object header.

```
#define BTNODE_NOHEADER 0x0010
```

This flag is valid only on B-trees that have the `BTREE_NOHEADER` flag.

If this flag is set, the `btn_o` field of this instance of `btree_node_phys_t` is always zero.

BTNODE_CHECK_KOFF_INVAL

The B-tree node is in a transient state.

```
#define BTNODE_CHECK_KOFF_INVAL 0x8000
```

Objects with this flag never appear on disk. If you find an object of this type in production, file a bug against the Apple File System implementation.

This flag isn't reserved by Apple; non-Apple implementations of Apple File System can set it on B-tree nodes in memory.

B-Tree Node Constants

Constants used to determine the size of a B-tree node.

```
#define BTREE_NODE_SIZE_DEFAULT      4096
#define BTREE_NODE_MIN_ENTRY_COUNT   4
```

A node is almost always one logical block in size. Smaller nodes waste space, and larger nodes can experience allocation issues when space is fragmented. For example, a five-block node requires five adjacent blocks to all be free, but on a fragmented disk such a large free space might not exist.

BTREE_NODE_SIZE_DEFAULT

The default size, in bytes, of a B-tree node.

```
#define BTREE_NODE_SIZE_DEFAULT 4096
```

BTREE_NODE_MIN_ENTRY_COUNT

The minimum number of entries that must be able to fit in a nonleaf B-tree node.

```
#define BTREE_NODE_MIN_ENTRY_COUNT 4
```

To satisfy this requirement, reduce the size of the keys stored in the node. The maximum key size is computed as follows:

```
uint32_t btree_key_max_size(uint32_t nodesize) {
    uint32_t dataspace, esize, count, kvspace;

    dataspace = nodesize - offsetof(btree_node_phys_t, btn_data)
            - sizeof(btree_info_t);
```

```
    esize = sizeof(kvloc_t);
    count = BTREE_TOC_ENTRY_INCREMENT;
    kvspace = dataspace - (count * esize);
    return ((kvspace / BTREE_NODE_MIN_ENTRY_COUNT) - sizeof(oid_t));
}
```

> **Note**
>
> This requirement comes from logic in Apple's implementation that performs proactive splitting of B-tree nodes.

Encryption

Apple File System supports encryption in the data structures used for containers, volumes, and files. When a volume is encrypted, both its file-system tree and the contents of files in that volume are encrypted.

Depending on the device's capabilities, Apple File System uses either hardware or software encryption, which impacts encryption process and the meaning of several data structures. Hardware encryption is used for internal storage on devices that support it, including macOS (with T2 security chip) and iOS devices. Software encryption is used for external storage, and for internal storage on devices that don't support hardware encryption. When hardware encryption is in use, only the kernel can interact with internal storage.

> **Important**
>
> This document describes only software encryption.

The keys used to access file data are stored on disk in a wrapped state. You access these keys through a chain of key-unwrapping operations. The *volume encryption key* (VEK) is the default key used to access encrypted content on the volume. The *key encryption key* (KEK) is used to unwrap the VEK. The KEK is unwrapped in one of several ways:

- **User password.** The user enters their password, which is used to unwrap the KEK.
- **Personal recovery key.** This key is generated when the drive is formatted and is saved by the user on a paper printout. The string on that printout is used to unwrap the KEK.
- **Institutional recovery key.** This key is enabled by the user in Settings and allows the corresponding corporate master key to unwrap the KEK.
- **iCloud recovery key.** This key is used by customers working with Apple Support, and isn't described in this document.

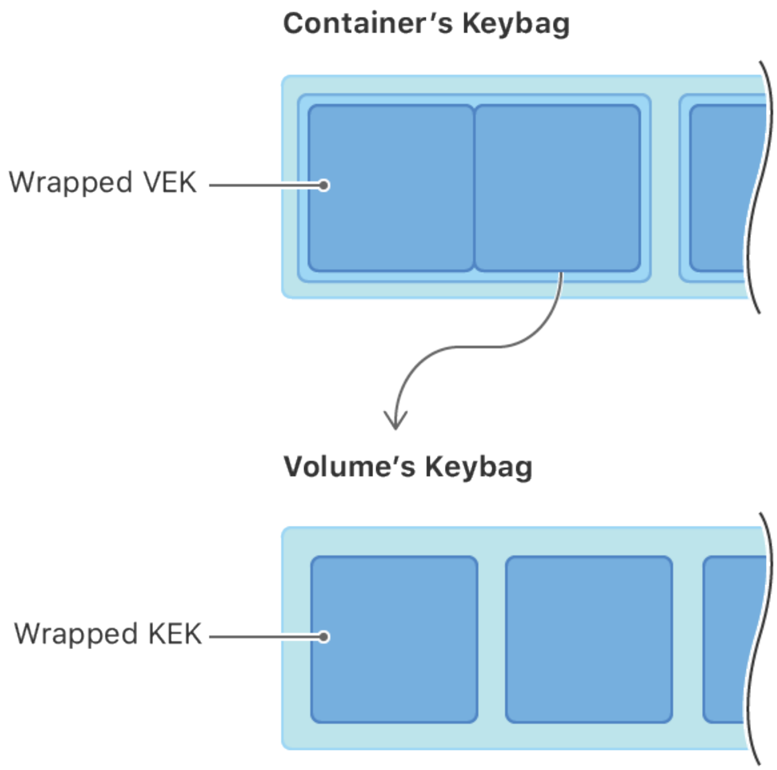
For example, to access a file given the user's password on a volume that uses per-volume encryption, the chain of key unwrapping and data decryption consists of the following high-level operations:

1. Unwrap the KEK using the user's password.
2. Unwrap the VEK using the KEK.
3. Decrypt the file-system B-tree using the VEK.
4. Decrypt the file data using the VEK.

The detailed steps are described in Accessing Encrypted Objects below.

Keybag

On macOS devices, both the container and the volume have a keybag (an instance of `kb_locker_t`). The container's keybag is stored at the location indicated by the `nx_keylocker` field of `nx_superblock_t`. For each volume, the container's keybag stores the volume's wrapped VEK and the location of the volume's keybag. The volume's keybag contains several copies of the volume's KEK, wrapped using user passwords and recovery keys.



Keybags are encrypted using the UUID of the container or volume, which makes it possible to quickly and securely destroy the contents of an encrypted volume by changing or deleting the UUID. For a volume, destroying the UUID by securely erasing a volume superblock makes the corresponding keybag unreadable, which in turn makes the encrypted content of that volume inaccessible. For a container superblock, you need to destroy all of the copies of that block in the checkpoint descriptor area and the copy at block zero.

Accessing Encrypted Objects

Before accessing an encrypted object, confirm that the `APFS_FS_ONEKEY` flag is set on the volume. Volumes that use per-file encryption require hardware encryption, and the steps below describe only software encryption.

To obtain the unwrapped VEK for a volume, do the following:

1. Locate the container's keybag using the `nx_keylocker` field of `nx_superblock_t`.
2. Unwrap the container's keybag using the container's UUID, according to the algorithm described in RFC 3394.
3. Find an entry in the container's keybag whose UUID matches the volume's UUID and whose tag is `KB_TAG_VOLUME_KEY`. The key data for that entry is the wrapped VEK for this volume.
4. Find an entry in the container's keybag whose UUID matches the volume's UUID and whose tag is `KB_TAG_VOLUME_UNLOCK_RECORDS`. The key data for that entry is the location of the volume's keybag.
5. Unwrap the volume's keybag using the volume's UUID according to the algorithm described in RFC 3394.
6. Find an entry in the volume's keybag whose UUID matches the user's Open Directory UUID and whose tag is `KB_TAG_VOLUME_UNLOCK_RECORDS`. The key data for that entry is the wrapped KEK for this volume.
7. Unwrap the KEK using the user's password, and then unwrap the VEK using the KEK, both according to the algorithm described in RFC 3394.

The volume's keybag might contain a passphrase hint for the user (KB_TAG_VOLUME_PASSPHRASE_HINT), which you can display when prompting for the password. It also might contain an entry for a personal recovery key, using APFS_FV_PERSONAL_RECOVERY_KEY_UUID as the UUID. You follow the same process for a personal recovery key as you do for a password: Unwrap the KEK with the user-entered string, and then use the unwrapped KEK to unwrap the VEK, both according to the algorithm described in RFC 3394.

To decrypt a file, do the following:

1. Decrypt the blocks where the volume's root file-system tree is stored, using the VEK as an AES-XTS key. The file-system tree is accessed using the `apfs_root_tree_oid` field of apfs_superblock_t.
2. Find the file extent record (APFS_TYPE_FILE_EXTENT) for the encrypted file.
3. Find the encryption state record (APFS_TYPE_CRYPTO_STATE) whose identifier matches the `crypto_id` field of j_file_extent_val_t.
4. Decrypt the blocks where the file's data is stored, using the VEK as an AES-XTS key and the value of `crypto_id` as the tweak.

j_crypto_key_t

The key half of a per-file encryption state record.

```
struct j_crypto_key {
    j_key_t     hdr;
} __attribute__((packed));
typedef struct j_crypto_key j_crypto_key_t;
```

Several encryption state objects always have the same identifier, as listed in Encryption Identifiers.

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always APFS_TYPE_CRYPTO_STATE.

j_crypto_val_t

The value half of a per-file encryption state record.

```
struct j_crypto_val {
    uint32_t                refcnt;
    wrapped_crypto_state_t  state;
} __attribute__((aligned(4),packed));
typedef struct j_crypto_val j_crypto_val_t;
```

refcnt

The reference count.

```
int32_t refcnt;
```

The encryption state record can be deleted when its reference count reaches zero.

state

The encryption state information.

```
wrapped_crypto_state_t state;
```

If this encryption state record is used by the file-system tree rather than by a file, this field is an instance of wrapped_meta_crypto_state_t and the key used is always the volume encryption key (VEK).

wrapped_crypto_state_t

A wrapped key used for per-file encryption.

```
struct wrapped_crypto_state {
    uint16_t            major_version;
    uint16_t            minor_version;
    crypto_flags_t      cpflags;
    cp_key_class_t      persistent_class;
    cp_key_os_version_t key_os_version;
    cp_key_revision_t   key_revision;
    uint16_t            key_len;
    uint8_t             persistent_key[0];
} __attribute__((aligned(2), packed));
typedef struct wrapped_crypto_state wrapped_crypto_state_t;

#define CP_MAX_WRAPPEDKEYSIZE       128
```

This structure is used inside of j_crypto_val_t.

major_version

The major version for this structure's layout.

```
uint16_t major_version;
```

The current value of this field is five. If backward-incompatible changes are made to this data structure in the future, the major version number will be incremented.

This structure is equivalent to a structure used by iOS for per-file encryption on HFS-Plus; versions four and earlier were used by previous versions of that structure.

minor_version

The major version for this structure's layout.

```
uint16_t minor_version;
```

The current value of this field is zero. If backward-compatible changes are made to this data structure in the future, the minor version number will be incremented.

cpflags

The encryption state's flags.

```
crypto_flags_t cpflags;
```

There are currently none defined.

persistent_class

The protection class associated with the key.

```
cp_key_class_t persistent_class;
```

For possible values and the bit mask that must be used, see Protection Classes.

key_os_version

The version of the OS that created this structure.

```
cp_key_os_version_t key_os_version;
```

This field is used as part of key rolling. For information about how the major version number, minor version number, and build number are packed into 32 bits, see `cp_key_os_version_t`.

key_revision

The version of the key.

```
cp_key_revision_t key_revision;
```

Set this field to one when creating a new instance, and increment it by one when rolling to a new key.

key_len

The size, in bytes, of the wrapped key data.

```
uint16_t key_len;
```

The maximum value of this field is `CP_MAX_WRAPPEDKEYSIZE`.

persistent_key

The wrapped key data.

```
uint8_t persistent_key[0];
```

CP_MAX_WRAPPEDKEYSIZE

The size, in bytes, of the largest possible key.

```
#define CP_MAX_WRAPPEDKEYSIZE 128
```

wrapped_meta_crypto_state_t

Information about how the volume encryption key (VEK) is used to encrypt a file.

```
struct wrapped_meta_crypto_state {
    uint16_t            major_version;
    uint16_t            minor_version;
    crypto_flags_t      cpflags;
    cp_key_class_t      persistent_class;
    cp_key_os_version_t key_os_version;
    cp_key_revision_t   key_revision;
    uint16_t            unused;
} __attribute__((aligned(2), packed));
typedef struct wrapped_meta_crypto_state wrapped_meta_crypto_state_t;
```

This structure is used inside of `j_crypto_val_t`. The fields in this structure are the same as `wrapped_crypto_state_t`, except this structure doesn't contain a wrapped key.

major_version

The major version for this structure's layout.

```
uint16_t major_version;
```

The value of this field is always five. This structure is equivalent to a structure used by iOS for per-file encryption on HFS-Plus; versions four and earlier were used by previous versions of that structure.

minor_version

The major version for this structure's layout.

```
uint16_t minor_version;
```

The value of this field is always zero.

cpflags

The encryption state's flags.

```
crypto_flags_t cpflags;
```

There are currently none defined.

persistent_class

The protection class associated with the key.

```
cp_key_class_t persistent_class;
```

For possible values, see Protection Classes.

key_os_version

The version of the OS that created this structure.

```
cp_key_os_version_t key_os_version;
```

For information about how the major version number, minor version number, and build number are packed into 32 bits, see cp_key_os_version_t.

key_revision

The version of the key.

```
cp_key_revision_t key_revision;
```

Set this field to one when creating a new instance.

unused

Reserved.

```
uint16_t unused;
```

Populate this field with zero when you create a new instance of this structure, and preserve its value when you modify an existing instance.

Encryption Types

Data types used in encryption-related structures.

```
typedef uint32_t    cp_key_class_t;
typedef uint32_t    cp_key_os_version_t;
typedef uint16_t    cp_key_revision_t;
typedef uint32_t    crypto_flags_t;
```

cp_key_class_t

A protection class.

```
typedef uint32_t cp_key_class_t;
```

For possible values, see Protection Classes.

cp_key_os_version_t

An OS version and build number.

```
typedef uint32_t cp_key_os_version_t;
```

This type stores an OS version and build number as follows:

- Two bytes for the major version number as an unsigned integer
- Two bytes for the minor version letter as an ASCII character
- Four bytes for the build number as an unsigned integer

For example, to store the build number 18A391:

1. Store the number 18 (`0x12`) in the highest two bytes, yielding `0x12000000`.
2. Store the character A (`0x41`) in the next two bytes, yielding `0x12410000`.
3. Store the number 391 (`0x0187`) in the lowest four bytes, yielding `0x12410187`.

cp_key_revision_t

A version number for an encryption key.

```
typedef uint16_t cp_key_revision_t;
```

crypto_flags_t

Flags used by an encryption state.

```
typedef uint32_t crypto_flags_t;
```

These flags are used by the `cpflags` field of `wrapped_crypto_state_t` and `wrapped_meta_crypto_state_t`. There are currently none defined.

Protection Classes

Constants that indicate the data protection class of a file.

```
#define PROTECTION_CLASS_DIR_NONE   0
#define PROTECTION_CLASS_A          1
#define PROTECTION_CLASS_B          2
#define PROTECTION_CLASS_C          3
#define PROTECTION_CLASS_D          4
#define PROTECTION_CLASS_F          6
#define PROTECTION_CLASS_M          14

#define CP_EFFECTIVE_CLASSMASK      0x0000001f
```

These values are used by the `persistent_class` field of `wrapped_meta_crypto_state_t`.

For more information about protection classes, see iOS Security Guide and FileProtectionType.

PROTECTION_CLASS_DIR_NONE

Directory default.

```
#define PROTECTION_CLASS_DIR_NONE 0
```

This protection class is used only on devices running iOS.

Files with this protection class use their containing directory's default protection class, which is set by the `default_protection_class` field of `j_inode_val_t`.

PROTECTION_CLASS_A

Complete protection.

```
#define PROTECTION_CLASS_A 1
```

This value corresponds to `FileProtectionType.complete`.

PROTECTION_CLASS_B

Protected unless open.

```
#define PROTECTION_CLASS_B 2
```

This value corresponds to `FileProtectionType.completeUnlessOpen`.

PROTECTION_CLASS_C

Protected until first user authentication.

```
#define PROTECTION_CLASS_C 3
```

This value corresponds to `FileProtectionType.completeUntilFirstUserAuthentication`.

PROTECTION_CLASS_D

No protection.

```
#define PROTECTION_CLASS_D 4
```

This value corresponds to `FileProtectionType.none`.

PROTECTION_CLASS_F

No protection with nonpersistent key.

```
#define PROTECTION_CLASS_F 6
```

The behavior of this protection class is the same as Class D, except the key isn't stored in any persistent way. This protection class is suitable for temporary files that aren't needed after rebooting the device, such as a virtual machine's swap file.

PROTECTION_CLASS_M

No overview available.

```
#define PROTECTION_CLASS_M 14
```

CP_EFFECTIVE_CLASSMASK

The bit mask used to access the protection class.

```
#define CP_EFFECTIVE_CLASSMASK 0x0000001f
```

All other bits are reserved. Populate those bits with zero when you create a wrapped key, and preserve their value when you modify an existing wrapped key.

Encryption Identifiers

Encryption state objects whose identifier is always the same.

```
#define CRYPTO_SW_ID                4
#define CRYPTO_RESERVED_5           5

#define APFS_UNASSIGNED_CRYPTO_ID   (~0ULL)
```

CRYPTO_SW_ID

The identifier of a placeholder encryption state used when software encryption is in use.

```
#define CRYPTO_SW_ID 4
```

There is no associated encryption key for this encryption state. All the fields of the corresponding `j_crypto_val_t` structure have a value of zero.

CRYPTO_RESERVED_5

Reserved.

```
#define CRYPTO_RESERVED_5 5
```

Don't create an encryption state object with this identifier. If you find an object with this identifier in production, file a bug against the Apple File System implementation.

APFS_UNASSIGNED_CRYPTO_ID

The identifier of a placeholder encryption state used when cloning files.

```
#define APFS_UNASSIGNED_CRYPTO_ID (~0ULL)
```

As a performance optimization when cloning a file, Apple's implementation sets this placeholder value on the clone and continues to use the original file's encryption state for both that file and its clone. If the clone is modified, a new encryption state object is created for the clone. Creating a new encryption state object is relatively expensive, and usually takes longer than the cloning process.

kb_locker_t

A keybag.

```
struct kb_locker {
    uint16_t        kl_version;
    uint16_t        kl_nkeys;
    uint32_t        kl_nbytes;
    uint8_t         padding[8];
    keybag_entry_t  kl_entries[];
};
typedef struct kb_locker kb_locker_t;

#define APFS_KEYBAG_VERSION           2
```

A keybag stores wrapped encryption keys and information that's needed to unwrap them. The container and each volume have their own keybag.

The container's keybag stores wrapped VEKs and the location of each volume's keybag. A volume's keybag stores wrapped KEKs.

kl_version

The keybag version.

```
uint16_t kl_version;
```

The value of this field is APFS_KEYBAG_VERSION.

kl_nkeys

The number of entries in the keybag.

```
uint16_t kl_nkeys;
```

kl_nbytes

The size, in bytes, of the data stored in the kl_entries field.

```
uint32_t kl_nbytes;
```

padding

Reserved.

```
uint8_t padding[8];
```

Populate this field with zero when you create a new keybag, and preserve its value when you modify an existing keybag.

This field is padding.

kl_entries

The entries.

```
keybag_entry_t kl_entries[];
```

APFS_KEYBAG_VERSION

The first version of the keybag.

```
#define APFS_KEYBAG_VERSION 2
```

Version one was used during prototyping of Apple File System, and uses an incompatible, undocumented layout. If you find a keybag in production whose version is less than two, file a bug against the Apple File System implementation.

keybag_entry_t

An entry in a keybag.

```
struct keybag_entry {
    uuid_t      ke_uuid;
    uint16_t    ke_tag;
    uint16_t    ke_keylen;
    uint8_t     padding[4];
    uint8_t     ke_keydata[];
};
typedef struct keybag_entry keybag_entry_t;

#define APFS_VOL_KEYBAG_ENTRY_MAX_SIZE      512
#define APFS_FV_PERSONAL_RECOVERY_KEY_UUID  "EBC6C064-0000-11AA-AA11-00306543ECAC"
```

ke_uuid

In a container's keybag, the UUID of a volume; in a volume's keybag, the UUID of a user.

```
uuid_t ke_uuid;
```

ke_tag

A description of the kind of data stored in this keybag entry.

```
uint16_t ke_tag;
```

For possible values, see Keybag Tags.

ke_keylen

The length, in bytes, of the keybag entry's data.

```
uint16_t ke_keylen;
```

The value of this field must be less than `APFS_VOL_KEYBAG_ENTRY_MAX_SIZE`.

padding

Reserved.

```
uint8_t padding[4];
```

Populate this field with zero when you create a new keybag entry, and preserve its value when you modify an existing entry.

This field is padding.

ke_keydata

The keybag entry's data.

```
uint8_t ke_keydata[];
```

The data stored this field depends on the tag and whether this is an entry in a container or volume's keybag, as described in Keybag Tags.

APFS_VOL_KEYBAG_ENTRY_MAX_SIZE

The largest size, in bytes, of a keybag entry.

```
#define APFS_VOL_KEYBAG_ENTRY_MAX_SIZE 512
```

APFS_FV_PERSONAL_RECOVERY_KEY_UUID

The user UUID used by a keybag record that contains a personal recovery key.

```
#define APFS_FV_PERSONAL_RECOVERY_KEY_UUID "EBC6C064-0000-11AA-AA11-00306543ECAC"
```

The personal recovery key is generated during the initial volume-encryption process, and it's stored by the user as a paper printout. You use it the same way you use a user's password to unwrap the corresponding KEK.

media_keybag_t

A keybag, wrapped up as a container-layer object.

```
struct media_keybag {
    obj_phys_t  mk_obj;
    kb_locker_t mk_locker;
};
typedef struct media_keybag media_keybag_t;
```

mk_obj

The object's header.

```
obj_phys_t mk_obj;
```

mk_locker

The keybag data.

```
kb_locker_t mk_locker;
```

Keybag Tags

A description of what kind of information is stored by a keybag entry.

```
enum {
    KB_TAG_UNKNOWN                = 0,
    KB_TAG_RESERVED_1             = 1,

    KB_TAG_VOLUME_KEY             = 2,
    KB_TAG_VOLUME_UNLOCK_RECORDS  = 3,
    KB_TAG_VOLUME_PASSPHRASE_HINT = 4,

    KB_TAG_WRAPPING_M_KEY         = 5,
```

```
    KB_TAG_VOLUME_M_KEY             = 6,

    KB_TAG_RESERVED_F8              = 0xF8
};
```

KB_TAG_UNKNOWN

Reserved.

`KB_TAG_UNKNOWN = 0`

This tag never appears on disk. If you find a keybag entry with this tag in production, file a bug against the Apple File System implementation.

This value isn't reserved by Apple; non-Apple implementations of Apple File System can use it in memory. For example, Apple's implementation uses this value as a wildcard that matches any tag.

KB_TAG_RESERVED_1

Reserved.

`KB_TAG_RESERVED_1 = 1`

Don't create keybag entries with this tag, but preserve any existing entries.

KB_TAG_VOLUME_KEY

The key data stores a wrapped VEK.

`KB_TAG_VOLUME_KEY = 2`

This tag is valid only in a container's keybag.

KB_TAG_VOLUME_UNLOCK_RECORDS

In a container's keybag, the key data stores the location of the volume's keybag; in a volume keybag, the key data stores a wrapped KEK.

`KB_TAG_VOLUME_UNLOCK_RECORDS = 3`

This tag is used only on devices running macOS.

The volume's keybag location is stored as an instance of `prange_t`; the data at that location is an instance of `kb_locker_t`.

KB_TAG_VOLUME_PASSPHRASE_HINT

The key data stores a user's password hint as plain text.

`KB_TAG_VOLUME_PASSPHRASE_HINT = 4`

This tag is valid only in a volume's keybag, and it's used only on devices running macOS.

KB_TAG_WRAPPING_M_KEY

The key data stores a key that's used to wrap a media key.

```
KB_TAG_WRAPPING_M_KEY = 5
```

This tag is used only on devices running iOS.

KB_TAG_VOLUME_M_KEY

The key data stores a key that's used to wrap media keys on this volume.

```
KB_TAG_VOLUME_M_KEY = 6
```

This tag is used only on devices running iOS.

KB_TAG_RESERVED_F8

Reserved.

```
KB_TAG_RESERVED_F8 = 0xF8
```

Don't create keybag entries with this tag, but preserve any existing entries.

Sealed Volumes

Sealed volumes contain a hash of their file system, which can be compared to their current content to determine whether the volume has been modified after it was sealed, or compared to a known value to determine whether the volume contains the expected content. On a sealed volume, all of the following must be true:

- The volume's role is `APFS_VOL_ROLE_SYSTEM`.
- The `APFS_INCOMPAT_SEALED_VOLUME` flag is set on the volume.
- The `apfs_integrity_meta_oid` field of `apfs_superblock_t` has a nonzero value.
- The `apfs_fext_tree_oid` field of `apfs_superblock_t` has a nonzero value.
- The `BTREE_HASHED` and `BTREE_NOHEADER` flags are set on the B-tree object that stores the volume's file system.

The B-tree that stores the volume's file system also stores a hash of its contents. A hashed B-tree differs from an nonhashed B-tree as follows:

- The `BTREE_HASHED` flag is set on the root node.
- The `BTNODE_HASHED` flag is set on the nonroot nodes.
- The values stored in nonleaf B-trees are instances of `btn_index_node_val_t`, containing the object identifier of the child node and the hash of the child node.

Conceptually, the hashed B-trees used by sealed volumes are similar to Merkle trees. However, unlike Merkle trees, these hashed B-trees store data as well as a hash of that data.

integrity_meta_phys_t

Integrity metadata for a sealed volume.

```
struct integrity_meta_phys {
    obj_phys_t          im_o;
    uint32_t            im_version;
    uint32_t            im_flags;
    apfs_hash_type_t    im_hash_type;
    uint32_t            im_root_hash_offset;
    xid_t               im_broken_xid;
    uint64_t            im_reserved[9];
} __attribute__((packed));
typedef struct integrity_meta_phys integrity_meta_phys_t;
```

im_o

The object's header.

```
obj_phys_t im_o;
```

im_version

The version of this data structure.

```
uint32_t im_version;
```

The value of this field must be one of the constants listed in Integrity Metadata Version Constants.

im_flags

The flags used to describe configuration options.

```
uint32_t im_flags;
```

For the values used in this bit field, see Integrity Metadata Flags.

This field appears in version 1 and later of this data structure.

im_hash_type

The hash algorithm being used.

```
apfs_hash_type_t im_hash_type;
```

This field appears in version 1 and later of this data structure.

im_root_hash_offset

The offset, in bytes, of the root hash relative to the start of this integrity metadata object.

```
uint32_t im_root_hash_offset;
```

This field appears in version 1 and later of this data structure.

im_broken_xid

The identifier of the transaction that unsealed the volume.

```
xid_t im_broken_xid;
```

When a sealed volume is modified, breaking its seal, that transaction identifier is recorded in this field and the `APFS_SEAL_BROKEN` flag is set. Otherwise, the value of this field is zero.

This field appears in version 1 and later of this data structure.

im_reserved

Reserved.

```
uint64_t im_reserved[9];
```

This field appears in version 2 and later of this data structure.

Integrity Metadata Version Constants

Version numbers for the integrity metadata structure.

```
enum {
    INTEGRITY_META_VERSION_INVALID  = 0,
    INTEGRITY_META_VERSION_1        = 1,
    INTEGRITY_META_VERSION_2        = 2,
    INTEGRITY_META_VERSION_HIGHEST  = INTEGRITY_META_VERSION_2
```

```
};
```

These constants are used as the value of the im_version field of the integrity_meta_phys_t structure.

INTEGRITY_META_VERSION_INVALID

An invalid version.

```
INTEGRITY_META_VERSION_INVALID = 0
```

INTEGRITY_META_VERSION_1

The first version of the structure.

```
INTEGRITY_META_VERSION_1 = 1
```

INTEGRITY_META_VERSION_1

The second version of the structure.

```
INTEGRITY_META_VERSION_2 = 2
```

INTEGRITY_META_VERSION_HIGHEST

The highest valid version number.

```
INTEGRITY_META_VERSION_HIGHEST = INTEGRITY_META_VERSION_2
```

Integrity Metadata Flags

Flags used by integrity metadata.

```
#define APFS_SEAL_BROKEN            (1U << 0)
```

These flags are used by the im_flags field of integrity_meta_phys_t.

APFS_SEAL_BROKEN

The volume was modified after being sealed, breaking its seal.

```
#define APFS_SEAL_BROKEN (1U << 0)
```

If this flag is set, the im_broken_xid field of integrity_meta_phys_t contains the transaction identifier for the modification that broke the seal.

apfs_hash_type_t

Constants used to identify hash algorithms.

```
typedef enum {
    APFS_HASH_INVALID               = 0,
    APFS_HASH_SHA256                = 0x1,
    APFS_HASH_SHA512_256            = 0x2,
    APFS_HASH_SHA384                = 0x3,
    APFS_HASH_SHA512                = 0x4,
```

```
    APFS_HASH_MIN                   = APFS_HASH_SHA256,
    APFS_HASH_MAX                   = APFS_HASH_SHA512,

    APFS_HASH_DEFAULT               = APFS_HASH_SHA256,
} apfs_hash_type_t;

#define APFS_HASH_CCSHA256_SIZE     32
#define APFS_HASH_CCSHA512_256_SIZE 32
#define APFS_HASH_CCSHA384_SIZE     48
#define APFS_HASH_CCSHA512_SIZE     64

#define APFS_HASH_MAX_SIZE          64
```

These constants are used as the value of the `im_hash_type` field of the `integrity_meta_phys_t` structure. The corresponding hash size is used as the value of the `hash_size` field of the `j_file_data_hash_val_t` structure.

APFS_HASH_INVALID

An invalid hash algorithm.

`APFS_HASH_INVALID = 0`

APFS_HASH_SHA256

The SHA-256 variant of Secure Hash Algorithm 2.

`APFS_HASH_SHA256 = 0x1`

APFS_HASH_SHA512_256

The SHA-512/256 variant of Secure Hash Algorithm 2.

`APFS_HASH_SHA512_256 = 0x2,`

APFS_HASH_SHA384

The SHA-384 variant of Secure Hash Algorithm 2.

`APFS_HASH_SHA384 = 0x3`

APFS_HASH_SHA512

The SHA-512 variant of Secure Hash Algorithm 2.

`APFS_HASH_SHA512 = 0x4`

APFS_HASH_MIN

The smallest valid value for identifying a hash algorithm.

`APFS_HASH_MIN = APFS_HASH_SHA256`

APFS_HASH_MAX

The largest valid value for identifying a hash algorithm.

```
APFS_HASH_MAX = APFS_HASH_SHA512
```

APFS_HASH_DEFAULT

The default hash algorithm.

```
APFS_HASH_DEFAULT = APFS_HASH_SHA256
```

APFS_HASH_CCSHA256_SIZE

The size of a SHA-256 hash.

```
#define APFS_HASH_CCSHA256_SIZE 32
```

APFS_HASH_CCSHA512_256_SIZE

The size of a SHA-512/256 hash.

```
#define APFS_HASH_CCSHA512_256_SIZE 32
```

APFS_HASH_CCSHA384_SIZE

The size of a SHA-384 hash.

```
#define APFS_HASH_CCSHA384_SIZE 48
```

APFS_HASH_CCSHA512_SIZE

The size of a SHA-512 hash.

```
#define APFS_HASH_CCSHA512_SIZE 64
```

APFS_HASH_MAX_SIZE

The maximum valid hash size.

```
#define APFS_HASH_MAX_SIZE 64
```

This value is the same as BTREE_NODE_HASH_SIZE_MAX.

fext_tree_key_t

The key half of a record from a file extent tree.

```
struct fext_tree_key {
    uint64_t private_id;
    uint64_t logical_addr;
} __attribute__((packed));
typedef struct fext_tree_key fext_tree_key_t;
```

private_id

The object identifier of the file.

```
uint64_t private_id;
```

This value corresponds the object identifier portion of the `obj_id_and_type` field of `j_key_t`.

logical_addr

The offset within the file's data, in bytes, for the data stored in this extent.

```
uint64_t logical_addr;
```

fext_tree_val_t

The value half of a record from a file extent tree.

```
struct fext_tree_val {
    uint64_t len_and_flags;
    uint64_t phys_block_num;
} __attribute__((packed));
typedef struct fext_tree_val fext_tree_val_t;
```

len_and_flags

A bit field that contains the length of the extent and its flags.

```
uint64_t len_and_flags;
```

The extent's length is a `uint64_t` value, accessed as `len_and_kind & J_FILE_EXTENT_LEN_MASK`, and measured in bytes. The length must be a multiple of the block size defined by the `nx_block_size` field of `nx_superblock_t`. The extent's flags are accessed as `(len_and_kind & J_FILE_EXTENT_FLAG_MASK) >> J_FILE_EXTENT_FLAG_SHIFT`.

There are currently no flags defined.

phys_block_num

The physical block address that the extent starts at.

```
uint64_t phys_block_num;
```

j_file_info_key_t

The key half of a file-info record.

```
struct j_file_info_key {
    j_key_t     hdr;
    uint64_t    info_and_lba;
} __attribute__((packed));
typedef struct j_key_t j_file_info_key_t;
```

```
#define J_FILE_INFO_LBA_MASK        0x00ffffffffffffffULL
#define J_FILE_INFO_TYPE_MASK       0xff00000000000000ULL
#define J_FILE_INFO_TYPE_SHIFT      56
```

hdr

The record's header.

```
j_key_t hdr;
```

The object identifier in the header is the file-system object's identifier. The type in the header is always `APFS_TYPE_FILE_INFO`.

info_and_lba

A bit field that contains the address and other information.

```
uint64_t info_and_lba;
```

The address is a `paddr_t` value accessed as `info_and_lba & J_FILE_INFO_LBA_MASK`. The type is a `j_obj_file_info_type` value accessed as `(info_and_lba & J_FILE_INFO_TYPE_MASK) >> J_FILE_INFO_TYPE_SHIFT`.

J_FILE_INFO_LBA_MASK

The bit mask used to access file-info addresses.

```
#define J_FILE_INFO_LBA_MASK 0x00ffffffffffffffULL
```

J_FILE_INFO_TYPE_MASK

The bit mask used to access file-info types.

```
#define J_FILE_INFO_TYPE_MASK 0xff00000000000000ULL
```

J_FILE_INFO_TYPE_SHIFT

The bit shift used to access file-info types.

```
#define J_FILE_INFO_TYPE_SHIFT 56
```

j_file_info_val_t

The value half of a file-info record.

```
struct j_file_info_val {
    union {
        j_file_data_hash_val_t  dhash;
    };
} __attribute__((packed));
typedef struct j_file_data_hash_val_t j_file_info_val_t;
```

Use the type stored in the `j_file_info_key_t` half of this record to determine which of the union's fields to use.

dhash

A hash of the file data.

```
j_file_data_hash_val_t dhash;
```

Use this field of the union if the type stored in the `info_and_lba` field of `j_file_info_val_t` is `APFS_FILE_INFO_DATA_HASH`.

j_obj_file_info_type

The type of a file-info record.

```
typedef enum {
    APFS_FILE_INFO_DATA_HASH = 1,
} j_obj_file_info_type;
```

These values are used by the `info_and_lba` field of `j_file_info_key_t`, to indicate how to interpret the data in the corresponding `j_file_info_val_t`.

APFS_FILE_INFO_DATA_HASH

The file-info record contains a hash of file data.

```
APFS_FILE_INFO_DATA_HASH = 1
```

j_file_data_hash_val_t

A hash of file data.

```
struct j_file_data_hash_val {
    uint16_t    hashed_len;
    uint8_t     hash_size;
    uint8_t     hash[0];
} __attribute__((packed));
typedef struct j_file_data_hash_val j_file_data_hash_val_t;
```

hashed_len

The length, in blocks, of the data segment that was hashed.

```
uint16_t hashed_len;
```

hash_size

The length, in bytes, of the hash data.

```
uint8_t hash_size;
```

The value of this field must match the constant that corresponds to the hash algorithm specified in the `im_hash_type` field of `integrity_meta_phys_t`. For a list of algorithms and hash sizes, see `apfs_hash_type_t`.

hash

The hash data.

```
uint8_t hash[0];
```

Space Manager

The space manager allocates and frees blocks where objects and file data can be stored. There's exactly one instance of this structure in a container.

chunk_info_t

No overview available.

```
struct chunk_info {
    uint64_t    ci_xid;
    uint64_t    ci_addr;
    uint32_t    ci_block_count;
    uint32_t    ci_free_count;
    paddr_t     ci_bitmap_addr;
};
typedef struct chunk_info chunk_info_t;
```

chunk_info_block

A block that contains an array of chunk-info structures.

```
struct chunk_info_block {
    obj_phys_t      cib_o;
    uint32_t        cib_index;
    uint32_t        cib_chunk_info_count;
    chunk_info_t    cib_chunk_info[];
};
typedef struct chunk_info_block chunk_info_block_t;
```

No overview available.

cib_addr_block

A block that contains an array of chunk-info block addresses.

```
struct cib_addr_block {
    obj_phys_t  cab_o;
    uint32_t    cab_index;
    uint32_t    cab_cib_count;
    paddr_t     cab_cib_addr[];
};
typedef struct cib_addr_block cib_addr_block_t;
```

No overview available.

spaceman_free_queue_entry_t

No overview available.

```
struct spaceman_free_queue_entry {
    spaceman_free_queue_key_t   sfqe_key;
    spaceman_free_queue_val_t   sfqe_count;
};
typedef struct spaceman_free_queue_entry spaceman_free_queue_entry_t;

typedef uint64_t spaceman_free_queue_val_t;
```

spaceman_free_queue_key_t

No overview available.

```
struct spaceman_free_queue_key {
    xid_t       sfqk_xid;
    paddr_t     sfqk_paddr;
};
typedef struct spaceman_free_queue_key spaceman_free_queue_key_t;
```

spaceman_free_queue_t

No overview available.

```
struct spaceman_free_queue {
    uint64_t    sfq_count;
    oid_t       sfq_tree_oid;
    xid_t       sfq_oldest_xid;
    uint16_t    sfq_tree_node_limit;
    uint16_t    sfq_pad16;
    uint32_t    sfq_pad32;
    uint64_t    sfq_reserved;
};
typedef struct spaceman_free_queue spaceman_free_queue_t;
```

spaceman_device_t

No overview available.

```
struct spaceman_device {
    uint64_t    sm_block_count;
    uint64_t    sm_chunk_count;
    uint32_t    sm_cib_count;
    uint32_t    sm_cab_count;
    uint64_t    sm_free_count;
    uint32_t    sm_addr_offset;
    uint32_t    sm_reserved;
    uint64_t    sm_reserved2;
};
typedef struct spaceman_device spaceman_device_t;
```

spaceman_allocation_zone_boundaries_t

No overview available.

```
struct spaceman_allocation_zone_boundaries {
    uint64_t saz_zone_start;
    uint64_t saz_zone_end;
};
typedef struct spaceman_allocation_zone_boundaries
        spaceman_allocation_zone_boundaries_t;
```

spaceman_allocation_zone_info_phys_t

No overview available.

```
struct spaceman_allocation_zone_info_phys {
    spaceman_allocation_zone_boundaries_t   saz_current_boundaries;
    spaceman_allocation_zone_boundaries_t
            saz_previous_boundaries[SM_ALLOCZONE_NUM_PREVIOUS_BOUNDARIES];
    uint16_t saz_zone_id;
    uint16_t saz_previous_boundary_index;
    uint32_t saz_reserved;
};
typedef struct spaceman_allocation_zone_info_phys
        spaceman_allocation_zone_info_phys_t;

#define SM_ALLOCZONE_INVALID_END_BOUNDARY       0
#define SM_ALLOCZONE_NUM_PREVIOUS_BOUNDARIES    7
```

spaceman_datazone_info_phys_t

No overview available.

```
struct spaceman_datazone_info_phys {
    spaceman_allocation_zone_info_phys_t
            sdz_allocation_zones[SD_COUNT][SM_DATAZONE_ALLOCZONE_COUNT];
};
typedef struct spaceman_datazone_info_phys spaceman_datazone_info_phys_t;

#define SM_DATAZONE_ALLOCZONE_COUNT 8
```

spaceman_phys_t

No overview available.

```
struct spaceman_phys {
    obj_phys_t                      sm_o;
    uint32_t                        sm_block_size;
    uint32_t                        sm_blocks_per_chunk;
    uint32_t                        sm_chunks_per_cib;
```

 |

```
    uint32_t                        sm_cibs_per_cab;
    spaceman_device_t               sm_dev[SD_COUNT];
    uint32_t                        sm_flags;
    uint32_t                        sm_ip_bm_tx_multiplier;
    uint64_t                        sm_ip_block_count;
    uint32_t                        sm_ip_bm_size_in_blocks;
    uint32_t                        sm_ip_bm_block_count;
    paddr_t                         sm_ip_bm_base;
    paddr_t                         sm_ip_base;
    uint64_t                        sm_fs_reserve_block_count;
    uint64_t                        sm_fs_reserve_alloc_count;
    spaceman_free_queue_t           sm_fq[SFQ_COUNT];
    uint16_t                        sm_ip_bm_free_head;
    uint16_t                        sm_ip_bm_free_tail;
    uint32_t                        sm_ip_bm_xid_offset;
    uint32_t                        sm_ip_bitmap_offset;
    uint32_t                        sm_ip_bm_free_next_offset;
    uint32_t                        sm_version;
    uint32_t                        sm_struct_size;
    spaceman_datazone_info_phys_t   sm_datazone;
};
typedef struct spaceman_phys spaceman_phys_t;

#define SM_FLAG_VERSIONED               0x00000001
```

sfq

No overview available.

```
enum sfq {
    SFQ_IP = 0,
    SFQ_MAIN = 1,
    SFQ_TIER2 = 2,
    SFQ_COUNT = 3
};
```

smdev

No overview available.

```
enum smdev {
    SD_MAIN = 0,
    SD_TIER2 = 1,
    SD_COUNT = 2
};
```

Chunk Info Block Constants

No overview available.

```
#define CI_COUNT_MASK              0x000fffff
#define CI_COUNT_RESERVED_MASK     0xfff00000
```

Internal-Pool Bitmap

No overview available.

```
#define SPACEMAN_IP_BM_TX_MULTIPLIER    16
#define SPACEMAN_IP_BM_INDEX_INVALID    0xffff
#define SPACEMAN_IP_BM_BLOCK_COUNT_MAX  0xfffe
```

Reaper

The reaper is a mechanism that allows large objects to be deleted over a period spanning multiple transactions. There's exactly one instance of this structure in a container.

nx_reaper_phys_t

No overview available.

```
struct nx_reaper_phys {
    obj_phys_t  nr_o;
    uint64_t    nr_next_reap_id;
    uint64_t    nr_completed_id;
    oid_t       nr_head;
    oid_t       nr_tail;
    uint32_t    nr_flags;
    uint32_t    nr_rlcount;
    uint32_t    nr_type;
    uint32_t    nr_size;
    oid_t       nr_fs_oid;
    oid_t       nr_oid;
    xid_t       nr_xid;
    uint32_t    nr_nrle_flags;
    uint32_t    nr_state_buffer_size;
    uint8_t  nr_state_buffer[];
};
typedef struct nx_reaper_phys nx_reaper_phys_t;
```

nx_reap_list_phys_t

No overview available.

```
struct nx_reap_list_phys {
    obj_phys_t  nrl_o;
    oid_t       nrl_next;
    uint32_t    nrl_flags;
    uint32_t    nrl_max;
    uint32_t    nrl_count;
    uint32_t    nrl_first;
    uint32_t    nrl_last;
    uint32_t    nrl_free;
    nx_reap_list_entry_t        nrl_entries[];
};
typedef struct nx_reap_list_phys nx_reap_list_phys_t;
```

nx_reap_list_entry_t

No overview available.

```
struct nx_reap_list_entry {
    uint32_t    nrle_next;
    uint32_t    nrle_flags;
    uint32_t    nrle_type;
    uint32_t    nrle_size;
    oid_t       nrle_fs_oid;
    oid_t       nrle_oid;
    xid_t       nrle_xid;
};
typedef struct nx_reap_list_entry nx_reap_list_entry_t;
```

Volume Reaper States

No overview available.

```
enum {
    APFS_REAP_PHASE_START           = 0,
    APFS_REAP_PHASE_SNAPSHOTS       = 1,
    APFS_REAP_PHASE_ACTIVE_FS       = 2,
    APFS_REAP_PHASE_DESTROY_OMAP    = 3,
    APFS_REAP_PHASE_DONE            = 4
};
```

Reaper Flags

The flags used for general information about a reaper.

```
#define NR_BHM_FLAG                 0x00000001
#define NR_CONTINUE                 0x00000002
```

These flags are used by the `nr_flags` field of `nx_reaper_phys_t`.

`NR_BHM_FLAG`

Reserved.

```
#define NR_BHM_FLAG 0x00000001
```

This flag must always be set.

`NR_CONTINUE`

The current object is being reaped.

```
#define NR_CONTINUE 0x00000002
```

Reaper List Entry Flags

No overview available.

```
#define NRLE_VALID                  0x00000001
#define NRLE_REAP_ID_RECORD         0x00000002
```

```
#define NRLE_CALL                   0x00000004
#define NRLE_COMPLETION             0x00000008
#define NRLE_CLEANUP                0x00000010
```

Reaper List Flags

No overview available.

```
#define NRL_INDEX_INVALID           0xffffffff
```

omap_reap_state_t

State used when reaping an object map.

```
struct omap_reap_state {
    uint32_t    omr_phase;
    omap_key_t  omr_ok;
};
typedef struct omap_reap_state omap_reap_state_t;
```

The reaper uses the state that's stored in this structure to resume after an interruption.

omr_phase

The current reaping phase.

```
uint32_t omr_phase;
```

For the values used in this field, see Object Map Reaper Phases.

omr_ok

The key of the most recently freed entry in the object map.

```
omap_key_t omr_ok;
```

This field allows the reaper to resume after the last entry it processed.

omap_cleanup_state_t

State used when reaping to clean up deleted snapshots.

```
struct omap_cleanup_state {
    uint32_t    omc_cleaning;
    uint32_t    omc_omsflags;
    xid_t       omc_sxidprev;
    xid_t       omc_sxidstart;
    xid_t       omc_sxidend;
    xid_t       omc_sxidnext;
    omap_key_t  omc_curkey;
};
typedef struct omap_cleanup_state omap_cleanup_state_t;
```

omc_cleaning

A flag that indicates whether the structure has valid data in it.

```
uint32_t omc_cleaning;
```

If the value of this field is zero, the structure has been allocated and zeroed, but doesn't yet contain valid data. Otherwise, the structure is valid.

omc_omsflags

The flags for the snapshot being deleted.

```
uint32_t omc_omsflags;
```

The value for this field is the same as the value of the snapshot's `omap_snapshot_t.oms_flags` field.

omc_sxidprev

The transaction identifier of the snapshot prior to the snapshots being deleted.

```
xid_t omc_sxidprev;
```

omc_sxidstart

The transaction identifier of the first snapshot being deleted.

```
xid_t omc_sxidstart;
```

omc_sxidend

The transaction identifier of the last snapshot being deleted.

```
xid_t omc_sxidend;
```

omc_sxidnext

The transaction identifier of the snapshot after the snapshots being deleted.

```
xid_t omc_sxidnext;
```

omc_curkey

The key of the next object mapping to consider for deletion.

```
omap_key_t omc_curkey;
```

apfs_reap_state_t

No overview available.

```
struct apfs_reap_state {
    uint64_t    last_pbn;
    xid_t       cur_snap_xid;
    uint32_t    phase;
```

```
} __attribute__((packed));
typedef struct apfs_reap_state apfs_reap_state_t;
```

Encryption Rolling

No overview available.

er_state_phys_t

No overview available.

```
struct er_state_phys {
    er_state_phys_header_t  ersb_header;
    uint64_t                ersb_flags;
    uint64_t                ersb_snap_xid;
    uint64_t                ersb_current_fext_obj_id;
    uint64_t                ersb_file_offset;
    uint64_t                ersb_progress;
    uint64_t                ersb_total_blk_to_encrypt;
    oid_t                   ersb_blockmap_oid;
    uint64_t                ersb_tidemark_obj_id;
    uint64_t                ersb_recovery_extents_count;
    oid_t                   ersb_recovery_list_oid;
    uint64_t                ersb_recovery_length;
};
typedef struct er_state_phys er_state_phys_t;

struct er_state_phys_v1 {
    er_state_phys_header_t  ersb_header;
    uint64_t                ersb_flags;
    uint64_t                ersb_snap_xid;
    uint64_t                ersb_current_fext_obj_id;
    uint64_t                ersb_file_offset;
    uint64_t                ersb_fext_pbn;
    uint64_t                ersb_paddr;
    uint64_t                ersb_progress;
    uint64_t                ersb_total_blk_to_encrypt;
    uint64_t                ersb_blockmap_oid;
    uint32_t                ersb_checksum_count;
    uint32_t                ersb_reserved;
    uint64_t                ersb_fext_cid;
    uint8_t                 ersb_checksum[0];
};
typedef struct er_state_phys er_state_phys_v1_t;

struct er_state_phys_header {
    obj_phys_t  ersb_o;
    uint32_t    ersb_magic;
    uint32_t    ersb_version;
};
```

```
typedef struct er_state_phys_header er_state_phys_header_t;
```

er_phase_t

No overview available.

```
enum er_phase_enum {
    ER_PHASE_OMAP_ROLL = 1,
    ER_PHASE_DATA_ROLL = 2,
    ER_PHASE_SNAP_ROLL = 3,
};
typedef enum er_phase_enum er_phase_t;
```

er_recovery_block_phys_t

No overview available.

```
struct er_recovery_block_phys {
    obj_phys_t      erb_o;
    uint64_t        erb_offset;
    oid_t           erb_next_oid;
    uint8_t         erb_data[0];
};
typedef struct er_recovery_block_phys er_recovery_block_phys_t;
```

gbitmap_block_phys_t

No overview available.

```
struct gbitmap_block_phys {
    obj_phys_t  bmb_o;
    uint64_t    bmb_field[0];
};
typedef struct gbitmap_block_phys gbitmap_block_phys_t;
```

gbitmap_phys_t

No overview available.

```
struct gbitmap_phys {
    obj_phys_t  bm_o;
    oid_t       bm_tree_oid;
    uint64_t    bm_bit_count;
    uint64_t    bm_flags;
};
typedef struct gbitmap_phys gbitmap_phys_t;
```

Encryption-Rolling Checksum Block Sizes

No overview available.

```
enum {
    ER_512B_BLOCKSIZE  = 0,
    ER_2KiB_BLOCKSIZE  = 1,
    ER_4KiB_BLOCKSIZE  = 2,
    ER_8KiB_BLOCKSIZE  = 3,
    ER_16KiB_BLOCKSIZE = 4,
    ER_32KiB_BLOCKSIZE = 5,
    ER_64KiB_BLOCKSIZE = 6,
};
```

Encryption Rolling Flags

No overview available.

```
#define ERSB_FLAG_ENCRYPTING          0x00000001
#define ERSB_FLAG_DECRYPTING          0x00000002
#define ERSB_FLAG_KEYROLLING          0x00000004
#define ERSB_FLAG_PAUSED              0x00000008
#define ERSB_FLAG_FAILED              0x00000010
#define ERSB_FLAG_CID_IS_TWEAK        0x00000020
#define ERSB_FLAG_FREE_1              0x00000040
#define ERSB_FLAG_FREE_2              0x00000080

#define ERSB_FLAG_CM_BLOCK_SIZE_MASK  0x00000F00
#define ERSB_FLAG_CM_BLOCK_SIZE_SHIFT 8

#define ERSB_FLAG_ER_PHASE_MASK       0x00003000
#define ERSB_FLAG_ER_PHASE_SHIFT      12
#define ERSB_FLAG_FROM_ONEKEY         0x00004000
```

Encryption-Rolling Constants

No overview available.

```
#define ER_CHECKSUM_LENGTH          8
#define ER_MAGIC                    'FLAB'
#define ER_VERSION                  1

#define ER_MAX_CHECKSUM_COUNT_SHIFT 16
#define ER_CUR_CHECKSUM_COUNT_MASK  0x0000FFFF
```

Fusion

No overview available.

fusion_wbc_phys_t

No overview available.

```
typedef struct {
    obj_phys_t  fwp_objHdr;
    uint64_t    fwp_version;
    oid_t       fwp_listHeadOid;
    oid_t       fwp_listTailOid;
    uint64_t    fwp_stableHeadOffset;
    uint64_t    fwp_stableTailOffset;
    uint32_t    fwp_listBlocksCount;
    uint32_t    fwp_reserved;
    uint64_t    fwp_usedByRC;
    prange_t    fwp_rcStash;
} fusion_wbc_phys_t;
```

fusion_wbc_list_entry_t

No overview available.

```
typedef struct {
    paddr_t     fwle_wbcLba;
    paddr_t     fwle_targetLba;
    uint64_t    fwle_length;
} fusion_wbc_list_entry_t;
```

fusion_wbc_list_phys_t

No overview available.

```
typedef struct {
    obj_phys_t  fwlp_objHdr;
    uint64_t    fwlp_version;
    uint64_t    fwlp_tailOffset;
    uint32_t    fwlp_indexBegin;
    uint32_t    fwlp_indexEnd;
    uint32_t    fwlp_indexMax;
    uint32_t    fwlp_reserved;
    fusion_wbc_list_entry_t fwlp_listEntries[];
} fusion_wbc_list_phys_t;
```

This mapping keeps track of data from the hard drive that's cached on the solid-state drive. For *read* caching, the same data is stored on both the hard drive and the solid-state drive. For *write* caching, the data is stored on the solid-state drive, but space for the data has been allocated on the hard drive, and the data will eventually be copied to that space.

Address Markers

No overview available.

```
#define FUSION_TIER2_DEVICE_BYTE_ADDR 0x4000000000000000ULL

#define FUSION_TIER2_DEVICE_BLOCK_ADDR(_blksize) \
            (FUSION_TIER2_DEVICE_BYTE_ADDR >> __builtin_ctzl(_blksize))

#define FUSION_BLKNO(_fusion_tier2, _blkno, _blksize) \
            ((_fusion_tier2) \
            ? (FUSION_TIER2_DEVICE_BLOCK_ADDR(_blksize) | (_blkno)) \
            : (_blkno))
```

`fusion_mt_key_t`

No overview available.

```
typedef paddr_t     fusion_mt_key_t;
```

`fusion_mt_val_t`

No overview available.

```
typedef struct {
    paddr_t     fmv_lba;
    uint32_t    fmv_length;
    uint32_t    fmv_flags;
} fusion_mt_val_t;
```

Fusion Middle-Tree Flags

No overview available.

```
#define FUSION_MT_DIRTY             (1 << 0)
#define FUSION_MT_TENANT            (1 << 1)
#define FUSION_MT_ALLFLAGS          (FUSION_MT_DIRTY | FUSION_MT_TENANT)
```

Symbol Index

Revision History

2020-06-22

Added the Sealed Volumes chapter.

Added the following symbols:

APFS_COW_EXEMPT_COUNT_NAME
APFS_FS_RESERVED_100
APFS_FS_RESERVED_80
APFS_INCOMPAT_SEALED_VOLUME
apfs_superblock_t.apfs_fext_tree_oid
apfs_superblock_t.apfs_fext_tree_type
apfs_superblock_t.apfs_integrity_meta_oid
apfs_superblock_t.reserved_oid
apfs_superblock_t.reserved_type
APFS_TYPE_FILE_INFO
APFS_UNASSIGNED_CRYPTO_ID
APFS_VOL_ROLE_PRELOGIN
BTNODE_HASHED
BTNODE_NOHEADER
btn_index_node_val_t
BTREE_HASHED
BTREE_NOHEADER
INODE_SNAPSHOT_COW_EXEMPTION
INO_EXT_TYPE_ORIG_SYNC_ROOT_ID
nx_superblock_t.nx_mkb_locker
OBJECT_TYPE_FEXT_TREE
OBJECT_TYPE_INTEGRITY_META
OBJECT_TYPE_MEDIA_KEYBAG
OBJECT_TYPE_RESERVED_20
spaceman_free_queue_entry_t

2020-05-15

Added the following symbols:

APFS_FEATURE_STRICTATIME
APFS_FEATURE_VOLGRP_SYSTEM_INO_SPACE
APFS_INCOMPAT_INCOMPLETE_RESTORE
apfs_superblock_t.apfs_cloneinfo_id_epoch
apfs_superblock_t.apfs_cloneinfo_xid
apfs_superblock_t.apfs_snap_meta_ext_oid
apfs_superblock_t.apfs_volume_group_id
APFS_VOL_ROLE_BACKUP
APFS_VOL_ROLE_ENTERPRISE
APFS_VOL_ROLE_HARDWARE
APFS_VOL_ROLE_RESERVED_10
APFS_VOL_ROLE_RESERVED_7
APFS_VOL_ROLE_RESERVED_8
APFS_VOL_ROLE_UPDATE
APFS_VOL_ROLE_XART
FIRMLINK_EA_NAME
INODE_FAST_PROMOTE
INODE_HAS_UNCOMPRESSED_SIZE
INODE_IS_PURGEABLE
INODE_IS_SYNC_ROOT
INODE_WANTS_TO_BE_PURGEABLE
INO_EXT_TYPE_PURGEABLE_FLAGS
j_inode_val.uncompressed_size
KB_TAG_VOLUME_M_KEY
KB_TAG_WRAPPING_M_KEY
nx_superblock_t.nx_newest_mounted_version
OBJECT_TYPE_ER_RECOVERY_BLOCK
OBJECT_TYPE_SNAP_META_EXT
OMAP_VALID_FLAGS
PROTECTION_CLASS_M
PURGEABLE_DIR_INO_NUM
snap_meta_ext_obj_phys_t
snap_meta_ext_t
SYSTEM_OBJ_ID_MARK
UNIFIED_ID_SPACE_MARK

2019-02-07

Corrected the discussion of object identifiers in j_snap_metadata_val_t. The extentref_tree_oid and sblock_oid fields contain a physical object identifier, not a virtual object identifier.

2019-01-24

Added information about software encryption on macOS in the Encryption chapter.

2018-09-17

New document that describes the data structures used for read-only access to Apple File System on unencrypted, non-Fusion storage.

Copyright and Notices

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
USA
408-996-1010