U5 Filesystem Reference Manual

Femtium Engineering Laboratories

2020-v1 Edition

1 U5FS

The U5FS filesystem exists in 2 versions. U5FSv0 (version 0) is the historic U5 filesystem, used by the original U5 architecture. It is now considered obsolete, and because of its limited feature set, its use in new projects is not recommended. U5FSv1 (version 1) is the modern version, and while it follows the design principles of v0, it is not backwards compatible.

The current Fenix kernel only supports v1 of U5FS.

A high-level overview of the properties of each filesystem:

Version 0

- Endianness: Always big-endian.
- Maximum block size: 2^{16} bytes.
- Maximum block count: 2^{16} bytes.
- Maximum filesystem size (using 4kb blocks): ~255 MB.
- Maximum file size: same as filesystem size.
- Supported file types: file, directory.
- Maximum hard links: 1 (no hard-link support).
- Directory name string encoding: zero-terminated strings with optional extra NULL byte. All names must be multiple-of-2 in size.
- Permissions: no.
- User/group ownership: no.
- Timestamps: no.

Version 1

- Endianness: Always big-endian.
- Maximum block size: 2^{32} bytes.
- Maximum block count: 2^{32} bytes.
- Maximum filesystem size (using 4kb blocks): ~16383 GB.
- Maximum file size: same as filesystem size.
- Supported file types: file, directory, symlink, character device, block device, fifo, socket (all standard unix file types).
- Maximum hard links: $2^{16} - 1$ per file.
- Directory name string encoding: zero-terminated UTF-8 strings.
- Permissions: 16-bit mask (standard unix permissions).
- User/group ownership: 32-bit user, 32-bit group.
- Timestamps: atime, mtime, ctime with nanosecond-precision.

2 High-level structure

There are 3 data structures found in all valid U5FS filesystems. These are: The superblock, the block allocation bitmap, and the root node.

The U5FS filesystem works by subdividing the storage space into a number of equally-sized blocks. These blocks are numbered using integers, starting from 0.

The superblock is a fixed-size structure (see section 10) that describes the properties of the filesystem. The superblock is always located at byte 0 in block 0, i.e. the very start of the filesystem. The superblock specifies the block size, block count, and root block number of the filesystem.

3 The allocation bitmap

The block allocation bitmap (or just "bitmap") is a set of consecutive blocks large enough to contain 1 bit for each block in the filesystem. Every block in the filesystem has a corresponding bit in the bitmap. If the block is in use, the corresponding bit will be set to 1. If it is available for allocation, the bit it set to 0.

Thus, when creating a new file, the bitmap can be scanned, and available blocks be found by looking for 0 bits. These bits must then be set to 1 if the corresponding block is going to be used.

The bitmap always starts at block 1, the first block after the superblock. The size of the bitmap, in blocks, is calculated as the smallest number of blocks that will contain 1 bit for each block in filesystem, from the superblock to the last block.

For example, a 420000-block long filesystem will need 420000 bits for the bitmap. If using 4096 byte blocks, we can fit $4096 * 8 = 32768$ bits per block. Thus, we need $\lceil \frac{420000}{32768} \rceil$ = $\lceil 12,817... \rceil$ = 13 blocks for the bitmap.

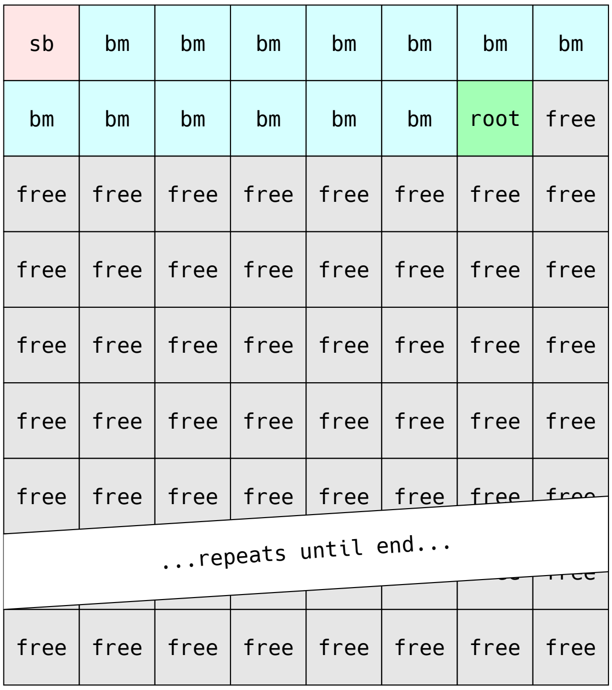
The superblock, all bitmap blocks, and the root node, will be marked in use in the bitmap, when a new U5FS is initialized.

4 The root node

The root node is traditionally placed in the first available block after the bitmap, but this is *not* a requirement. The superblock describes the location of the root node, and this is the location that must be used.

5 Example

In the diagram below, a conceptual illustration of a newly-initalized U5FS is shown. The first block is the superblock ("sb"), followed by the bitmap ("bm"), and then the root node. All other blocks are free, and available for use. In this example, the first 15 bits in the bitmap will be set to 1, representing the superblock + 13 bitmap blocks + root block that are in use. All other bits in the bitmap will be 0.



6 Directory structure

Every block in a U5FS filesystem will fall into one of the following categories:

- superblock (always block 0, and only block 0)
- bitmap (always starts at block 1)
- index node
- file data
- unused (free block, available for allocation)

We have already discussed the superblock and bitmap block types. Unused blocks are available for allocation, and represent the “free space” of the file system. File data blocks hold the data of a file. The only remaining block type is the index node.

Index nodes (better known by the UNIX name *inodes*), contain *metadata*, such as file names, file types, ownership information, timestamps, directory structures, etc.

Exactly one inode is guaranteed to exist - the root node. Its location can be looked up in the superblock. The root node is an inode that represents the root directory[1] of the filesystem.

Each type of filesystem object (file, directory, symlink, fifo, etc) has a corresponding type of inode, which contains the information about the object.[2]

The root node will always be of type `U5FS_DTYPE_DIR` (see section 13).

7 Directory inodes

All inode types contain basic attributes like ownership and permission information, timestamps, etc. Depending on the inode type, various other data follow this common data structure.

[1]In UNIX terms, “/”.
[2]See section 13 for a list of type constants.

The directory inode type contains a list of *directory entries*, which each describe a filesystem object contained in the directory. See section 12 for a description of this format.

If any of the objects in the root directory are themselves directories, the corresponding inodes of those directories can be looked up and read, to learn the contents of that directory.

In this way, it is possible to access any file in the filesystem by recursively iterating through directory entries, looking up new inodes, and iterating through the directory entries contained in those.

8 File inodes

Another common type of inode is the file inode. This inode contains a list of data blocks associated with the file. This list will contain $\lceil \frac{\text{file\_size}}{\text{block\_size}} \rceil$ block numbers. Each block contains block_size bytes of the file. To read the file contents, go through the blocks in the in the order they are listed in the file inode.

For example, a file of length 10000 bytes in a filesystem with 4096-byte blocks, will span 3 blocks. The last block will only use the first $10000 - 4096 * 2 = 1808$ bytes, since 10000 is not a multiple of the block size. The sum of all these "tails" of unused space is known as *slack space*. Generally, bigger blocks lead to more efficient file operations, but at the cost of increased losses due to slack space. On the other hand, a smaller block size means the file system can be more densely packed, but will require more operations on average. A commonly chosen block size is 4096 bytes, since this represents a good balance between efficient storage and efficient operation.[3]

Files do not have to be fully allocated. For example, a file might be of size 4MB, but only contain data towards the end of that range. To handle this scenario, all blocks indexes that are not allocated for the file point to block 0. Since block 0 is the superblock, it can never refer to a valid file data block.

9 Other inodes

Other inode types exist to handle types like fifo, sockets, and devices files. These contain only limited amounts of data, and are generally simpler to handle. Refer to section 11 for details.

10 Superblock

The U5FSv1 superblock has the following structure:

```
struct u5fs_superblock_v1:
    u32 magic          /* Must contain U5FS_SUPERBLOCK_MAGIC */
    u32 version        /* Must contain U5FS_VERSION_1 for v1 */
    u32 blocksize      /* Block size, in bytes */
    u32 blockcount     /* Total number of blocks, including superblock */
    u32 rootnode       /* Index of root node block */
```

The remaining space in block 0 is reserved, and must not be used.

11 Filesystem inode format

To make types simpler to read, we use a few helper types for simplicity:

```
struct cred:
    u32 uid      /* the user-id of the user that owns this file */
    u32 gid      /* the group-id of the group that owns this file */
```

[3]In fact, the current version of Fenix only supports the 4096 byte block size.

```
struct timespec:
    u32 time_sec  /* number of whole seconds since 00:00:00, January 1st, 1970 */
    u32 time_nsec /* fractional part of second, in nanoseconds */
```

The main u5fs inode type:

```
struct u5fs_inode:
    u32 indirection          /* pointer to 2nd-level block node, for large files */
    struct cred owner        /* uid/gid of owner */
    struct timespec atime    /* time of last file access */
    struct timespec mtime    /* time of last file modification */
    struct timespec ctime    /* time of last metadata change (chown/chmod/etc) */
    u16 perm                 /* permission bits */
    u16 links                /* number of references to this file */
    u32 size                 /* file size in bytes */
```

The “indirection” block is a pointer to a “next” block, for inodes too large to fit in a single block.[4]

In data layout form: (each row represents 32 bits)

<table>
<tr><td colspan="2">indirection_inode</td></tr>
<tr><td colspan="2">owner_uid</td></tr>
<tr><td colspan="2">owner_gid</td></tr>
<tr><td colspan="2">atime_secs</td></tr>
<tr><td colspan="2">atime_nsec</td></tr>
<tr><td colspan="2">mtime_secs</td></tr>
<tr><td colspan="2">mtime_nsec</td></tr>
<tr><td colspan="2">ctime_secs</td></tr>
<tr><td colspan="2">ctime_nsec</td></tr>
<tr><td>perm</td><td>links</td></tr>
<tr><td colspan="2">size</td></tr>
</table>

```
struct u5fs_file_inode:
    struct u5fs_inode header /* file inodes start with the common u5fs_inode fields. */
    u32 blocks[]             /* ..followed by a list of all data blocks that back the file. */
```

```
struct u5fs_dir_inode:
    struct u5fs_inode header /* dir inodes start with the common u5fs_inode fields. */
    u8 dirent[]              /* ..followed by a variable-length list of directory entries. */
                             /* (the .dirent data structure is header.size bytes long) */
                             /* see: struct u5fs_dentry */
```

```
struct u5fs_device_inode:
    struct u5fs_inode header /* dir inodes start with the common u5fs_inode fields */
    u16 padding              /* for historical reasons, these 2 bytes are unused */
    u16 major                /* for device nodes: major number */
    u16 minor                /* for device nodes: minor number */
```

[4]Fenix currently does not support indirected inodes. It is generally not required to implement this for most common use cases.

```
struct u5fs_symlink_inode:
    struct u5fs_inode header /* symlink inodes start with the common u5fs_inode fields */
    u16 size                 /* symlink length */
    u8 name[]                /* symlink name (NOT zero-terminated) */
```

12 Directory entry format

```
struct u5fs_dentry:
    u32 dnode   /* inode of file */
    u8  dtype   /* enum u5fs_dnode_type */
    u8  name[]  /* zero-terminated string */
```

13 Reference constants

```
const U5FS_SUPERBLOCK_MAGIC = 0x55354653 /* "U5FS" */
```

```
enum u5fs_version:
    U5FS_VERSION_0 = 0
    U5FS_VERSION_1 = 1
```

```
enum u5fs_dtype:
    U5FS_DTYPE_DIR  = 1
    U5FS_DTYPE_FILE = 2
    U5FS_DTYPE_CDEV = 3
    U5FS_DTYPE_BDEV = 4
    U5FS_DTYPE_LNK  = 5
    U5FS_DTYPE_PIPE = 6
    U5FS_DTYPE_SOCK = 7
```