File Formats

- **.tga File Extension (BIMAGE)**
- **.streamdb File Extension**
- **container.mask**
- **Sound Archives (.snd)**
- **soundmetadata.bin**
- **MapResources Files**

.tga File Extension (BIMAGE)

> The ".tga" files used in DOOM Eternal are not targa images. They are actually "BIM" (or .bimage) files.

> These files are stored in two parts. The first part consists of the `HEADER`, `MIPMAP`, and `NON_STREAMED_IMAGE` sections. This is embedded in DOOM Eternal's .resources files. The second part consists of the `STREAMED_IMAGE` section, which is embedded in Doom Eternal's .streamdb files.

Signature

DOOM Eternal ".tga" files can be identified by the first 3 bytes of the file header stored in a **.resources** file, which is always `0x42 0x49 0x4D` or `"BIM"`, short for "bimage" or "binary image". However, these files are usually stored compressed via Oodle Kraken, which may obscure the file signature.

To view these files, you can extract the .tga headers from a **.resources** file using the **EternalResourceExtractor tool**.

File Structure

A DOOM Eternal ".tga" file consists of 4 sections:

1. The `HEADER` section, which contains important details about the image size, format and encoding.
2. The `MIPMAP` section, which contains metadata for each individual image "mip"
3. The `NON_STREAMED_IMAGE` section, which contains non-streamed versions of the image. Usually, these are any "mips" that are smaller than about 32x32 pixels. Sometimes, though, it can be a full-sized image.
4. The `STREAMED_IMAGE` section, which is the portion of the image stored in .streamdb. The largest version of the image is normally stored in the .streamdb, along with several smaller mips.

> The `HEADER`, `MIPMAP`, and `NON_STREAMED_IMAGE` sections are stored as a single compressed file, embedded within a DOOM Eternal **.resources** file. The `STREAMED_IMAGE` is

> stored separately in a **.streamdb** file. Even though they are stored separately, they are considered one "file" because exporting them to a usable format such as .DDS, .PNG, etc, requires that we connect all the disparate pieces.

```
struct ETERNAL_TGA_FILE
{
    HEADER header;
    MIPMAP mipmap[];□□□□□□□□// Array length varies with header.MipCount;
    NON_STREAMED_IMAGE non_streamed_image[];□// Smaller mips of the image.
    STREAMED_IMAGE streamed_image[];□□□// Full-size + larger mips of the image.
};
```

Header Section

The `HEADER` struct is a 63-byte sequence:

```
struct HEADER
{□□
□char Signature[3];□□□□// "BIM"
    byte Version;□□□□□// 0x15
□int TextureType; □□□□// enum textureType_t
□int TextureMaterialKind;□□// enum textureMaterialKind_t
□int PixelWidth;□□□□□// image width in pixels
□int PixelHeight;□□□□// image height in pixels
□int Depth;
    int MipCount; □□□□□// determines # of MIPMAP structs to follow
□int64_t MipLevel;□□□□
□float unkFloat1; □□□□// usually 1.0, purpose unknown
□byte boolIsEnvironmentMap;□□// 1 if image is an environment map
□int TextureFormat;□□□□// enum textureFormat_t
□int Always7;□□□□□// literally always 7, purpose unknown
□int nullPadding;
□short AtlasPadding;
□byte boolIsStreamed;□□□// 1 if file is located in streamDB
□byte unkBool;
□byte boolNoMips;□□□□// 1 if image has no mips
□byte boolFFTBloom;□□□□// 1 if using FFT Bloom
□int StreamDBMipCount; □□□// usually same number of mips stored in streamdb
};
```

- `TextureType` is a member of the enum **`textureType_t`** - it marks the image as 2-dimensional, 3-dimensional, or cubic.
- `TextureMaterialKind` is a member of the enum **`textureMaterialKind_t`** - it tells the engine what type of material this texture is (albedo, normal, specular, etc).
- `TextureFormat` is a member of the enum **`textureFormat_t`** - it tells the engine how the image is encoded (image format, block compression type, etc).

Mipmap Section

The `MIPMAP` struct is a 36-byte sequence that comes immediately after the `HEADER`. This struct will be repeated `n` times, where `n` is equal to `HEADER.MipCount` above.

```
struct MIPMAP
{
□int64_t MipLevel;□□□□// Starts at 0, increment by 1 each time it repeats
□int MipPixelWidth;□□□□// Original PixelWidth reduced by 50% for each MipLevel
□int MipPixelHeight;□□□□// Original PixelHeight reduced by 50% for each MipLevel
□int UnknownFlagA;
□int DecompressedSize;□□□// Decompressed size in bytes
□int FlagIsCompressed;□□□// 1 if the texture is compressed
□int CompressedSize;□□□□// Compressed size in bytes
□int CumulativeSizeStreamDB;□□
};
```

- `CumulativeSizeStreamDB` is always zero for the first mipmap. For additional mipmaps, it is the sum of previous mipmaps compressed sizes.

Non-Streamed Images

The `NON_STREAMED_IMAGE` section begins immediately after the last `MIPMAP`. The **starting offset** of this section can be calculated as `offset = 63 + (36 * HEADER.MipCount)`.

Usually, any mips smaller than about 50x50 pixels in size will be stored in this section. They are packed together back-to-back, from largest mip to smallest (excluding any mips that are located in the .streamdb). The size in bytes and the pixel dimensions of each mip are given by the corresponding `MIPMAP` struct.

```
struct NON_STREAMED_IMAGE
{
```

```
□BYTE rawImageData[];□□□□□// Image format and encoding given in HEADER
};
```

> Some images are "non-streaming," which means they aren't present in the .streamdb files at all. Non-streaming images will always have a `HEADER.boolIsStreamed = 0` (false). In that case, the full-sized image and all mips will be stored here, and there will not be any `STREAMED_IMAGE` section.

Streamed Images

In most cases, the full-size versions of these ".tga" images are stored in .streamdb files in a **headerless** format, where they are accessed by the game engine as needed.

```
struct STREAMED_IMAGE
{
□BYTE rawImageData[];□□□□□// Image format & encoding given in HEADER
};
```

Unlike the `NON_STREAMED_IMAGE` section, these streamed images are not stored back-to-back in the **.streamdb**. Each streamed mip of the image has its own entry in the .streamdb index.

010 Editor Template

A 010 Editor template for use with Doom Eternal's TGA file headers can be found here:

https://github.com/brongo/eternal-010-templates/blob/main/templates/DoomEternalTGA.bt

.streamdb File Extension

> .streamdb stands for stream database. The .streamdb files contain the majority of game data in DOOM Eternal.

> As a general rule, any struct with `_t` appended to the end is an **actual struct used by the game engine**. Any other struct names have been created by the wiki author for organization/convenience purposes.

Signature

DOOM Eternal ".streamdb" files can be identified by the first 8 bytes of the file header, which is always: `0x50A5C2292EF3C761`.

Internally, the game engine references a 2nd type of stream database, which would be identified by a slightly different file signature: `0x4FA5C2292EF3C761` - however, this signature has not been observed in any files used in DOOM Eternal.

File Structure

The .streamdb file consists of 3 parts. An `INDEX` section, which is a list of all the files contained within, followed by a `PREFETCH` section, and finally the `DATA` section, which contains data referenced by the index.

```
struct STREAM_DB_FILE
{
INDEX index;
    PREFETCH prefetch;
    DATA data;
};
```

The `INDEX` contains a list of hashed IDs rather than plaintext names. All files contained within the .streamdb are stored in a **headerless** format, and are usually compressed via Oodle Kraken or Oodle Leviathan compression technology.

> Files embedded in the .streamdb are impossible to identify by looking at the .streamdb alone. Instead, they are referenced via data contained in **.resources** files.

Index Section

The .streamdb `INDEX` structure is of varying length. It consists of a 32-byte header, followed by a variable number of 16-byte entries. The overall structure is described as follows:

```
struct INDEX
{
□streamDatabaseHeader_t header;□□□□□// File signature + metadata
□streamDatabaseEntry2_t streamdbEntries[];□□// One 16-byte entry for each header.numEntries
};
```

The `streamDatabaseHeader_t` struct is a 32-byte sequence:

```
struct streamDatabaseHeader_t
{□□
□uint64 magic;□□□□□□□// 50 A5 C2 29 2E F3 C7 61
□uint32 headerLength;□
□uint32 pad0;□□□□□□□// null padding
□uint32 pad1;□□□□□□□// null padding
□uint32 pad2;□□□□□□□// null padding
□uint32 numEntries;□□□□□□// Total entries, not including prefetch IDs
□uint32 flags;□□□□□□□// Always 3
};
```

The `streamDatabaseEntry2_t` struct is a 16-byte sequence. It will be repeated `n` times, where `n = streamDatabaseHeader_t.numEntries`. Therefore, the total length of the `INDEX` section can be calculated as `length = 32 + (16 * streamDatabaseHeader_t.numEntries)`

```
struct streamDatabaseEntry2_t
{□□
□uint64 identity;□□□□□□// Shuffled version of .resources ID
□uint32 offset16; □□□□□□// Multiply by 16 for data offset within .streamdb
□uint32 length;□□□□□□□// Size of the file in .streamdb (usually compressed)
};
```

After the last entry, the `INDEX` section ends and the `PREFETCH` section begins.

Prefetch Section

The .streamdb `PREFETCH` structure is of varying length. It consists of a 16-byte header, followed (optionally) by either one or two 16-byte prefetchBlocks, and a number of 8-byte prefetchIDs.

The overall `PREFETCH` structure is described as follows:

```
struct PREFETCH
{
    streamDatabasePrefetchHeader_t prefetchHeader;□// Prefetch section header
    streamDatabasePrefetchBlock_t prefetchBlock[];□// (Optional) Between 0-2 prefetch "blocks"
□uint64 prefetchID[];□□□□□□□// (Optional) 0 or more prefetch file IDs.
};
```

The `streamDatabasePrefetchHeader_t` struct is a 16-byte sequence. It is always present in the .streamdb file, even if this .streamdb does not contain any prefetch entries.

```
struct streamDatabasePrefetchHeader_t
{
□uint32 numPrefetchBlocks;
□uint32 totalLength;□□□□□□// Total length of prefetch header, blocks, entries
};
```

If `streamDatabasePrefetchHeader_t.numPrefetchBlocks = 0`, then the PREFETCH section ends here. Otherwise, it is followed by the number of `streamDatabasePrefetchBlock_t` structs specified (which is always an integer between `0` and `2`).

```
struct streamDatabasePrefetchBlock_t
{
□uint64 name;□□□□□□□// Hash of "AI" or "FirstPerson"
□uint32 firstItemIndex;□□□□□// Offset relative to end of prefetch blocks
□uint32 numItems;□□□□□□// Num prefetch entries in this block
};
```

> The "name" is a hash of either the word `AI` or `FirstPerson`. A value of `5891933081285280768` is a hash of the word `AI`, and a value of `6801151928053439575` is a hash of the word `FirstPerson`.

Finally, the `PREFETCH` section ends with an array of `uint64 prefetchIDs[]` - each of these IDs will match a `streamDatabaseEntry2_t.identity` from the `INDEX` section above. The number of `prefetchID` is the specified by `streamDatabasePrefetchBlock_t.numItems`.

Data Section

The `DATA` section begins at the offset given in `streamDatabaseHeader_t.headerLength`. This section is simply a series of compressed files. The starting offset and the length (in bytes) of each file is given by a `streamDatabaseEntry2_t` in `INDEX` section.

There is often some null padding at the end of each compressed file. This is because the starting offset must be evenly divisible by 16 (because `streamDatabaseEntry2_t.offset16` is multiplied by `16` for the file offset - presumably to allow these offsets to be stored as uint32 rather than uint64).

The compressed files commonly begin with the bytes `8C 06` or `CC 06` which identifies Oodle Kraken compression.

010 Editor Template

A 010 Editor template for use with Doom Eternal's .streamdb files can be found here:

https://github.com/brongo/eternal-010-templates/blob/main/templates/DoomEternalStreamDB.bt

container.mask

About

In idTech7 and 8, `container.mask` is the single file located inside the `meta.resources` archive. It is a set of bit-masks, one for every resource archive in the game. Each file inside the archive has a bit in the mask, based on the order they appear inside the archive's file list. If a file's bit is `0` , then that file will not be loaded by the game.

This file is built to guarantee that only one version of a file will ever be loaded by the game simultaneously. When a game update adds new resource archives, they often contain updated versions of existing files. Only the latest copies of a file will have their bitmask files set to `1`. When combined with the `packagemapspec.json`, the container mask creates 2 layers of security for ensuring the correct copies of a file are loaded.

Edge Cases

When determining the highest-priority version of a file, it was previously thought you only needed to refer to the `packagemapspec.json` file. This assumption is wrong. Edge cases exist where the container mask disables the version of a file found in the higher-priority archive, and enables a version in a lower-priority archive instead. This may happen in instances where id modifies - then later reverts - a file over the course of several game updates. Therefore, you **must** use the container mask to determine which copy of a file is actually used by the game.

If multiple versions of a file exist, and **all** of them are disabled by the container mask, you cannot determine the latest version with absolute certainty. You could go by highest-priority archive or by file timestamps, but neither of those tools will be perfect.

Enabling Multiple Versions of a File

What if two different versions of a file are both enabled via the container mask? In the vanilla filesystem, this does not happen. However, when it does occur due to modding, the game will use the version from the highest-priority archive, according to the packagemapspec.json `files` array.

Format

The file format is fairly straightforward

```
struct bitmask {
□uint64_t□□hash;□□□// Hash used to identify the resource archive
    uint32_t        size;           // Size of the mask, in 64-bit integers
```

```
    uint64_t*       mask;           // Array of 64 bit integers representing the raw bitmask.
length == size
}

struct maskfile {
□uint32_t□□timestamp;     □// Not present in idTech8
    uint32_t□□num_bitmasks;  □// Number of bitmasks in the file
    bitmask*□□bitmasks;□□// Array of bitmasks, length == num_bitmasks
}
```

Hashes

As seen in the structures, the container.mask uses hashes to associate each archive with a bitmask. These are Farmhash64 hashes of each archive's metadata section. The hashed data begins after the archive's header, and ends at the `IDCL` magic that terminates the meta section. (The magic is included in the hashed data)

This code shows how to calculate a container mask hash for an archive.

Extra Bitmask Slots

As seen in the structures, the bitmasks are defined as 64-bit integers. This results in several implicit behaviors:

- The maximum number of files that can be stored in a resource archive is always a multiple of 64. Trying to store too many files (such that you exceed the size of the bitmask), will likely cause crashes or instability.
- If the number of files in a resource archive is not a multiple of 64, the bitmask will have some unused bits at it's end. These bits are either all `1` or all `0` - with their exact value determined by whether or not it's the highest-priority archive in it's patch group.

```
Example: common.resources, common_patch1.resources and common_patch2.resources

If the archive priority (based on packagemapspec.json) is:
- common_patch1.resources
- common_patch2.resources
- common.resources
```

```
Then:
- All extra bitmask bits in common_patch1 will have a value of 1
- All extra bits in every other archive's bitmask will have a value of 0
```

Container Mask (Audio Archives)

The `.snd` archives for storing audio files in idTech7 and 8 also have their own container mask. It is located in the `soundmetadata.bin` file. It's stored near the beginning of the file in DOOM Eternal. In DOOM The Dark Ages, it's at the end of the file.

Format

The audio container mask format is slightly more complex

```
struct sndBitmask {
□uint32_t  hash;      // Identifies the .snd container associated with this bitmask
    uint32_t  mask_size; // Size of the bitmask (in 32-bit integers)
  □uint32_t* bitmask;   // Array of integers representing the raw bitmask. Length == mask_size
}

// Encompasses a group of snd archives
// (i.e. SFX.snd, SFX_Patch_1.snd and SFX_Patch_2.snd all belong to one mask group)
struct sndMaskGroup {
□uint32_t □□group_name_length;
    char* □□□group_name;   □□// Group name string. Not null-terminated. Length ==
group_name_length
    uint32_t □□num_bitmasks; □□// Number of bitmasks in this group
    sndBitmask* □bitmasks;□  □□// Array of bitmasks. Length == num_bitmasks
}

struct sndMaskChunk {
□uint32_t      num_groups; // Number of mask groups in the chunk
    sndMaskGroup* groups;     // Array of mask groups. Length == num_groups
}
```

Hashes

The audio container mask also uses hashes to associate each .snd archive with it's bitmask.

AudioKinetic's case-insensitive FNV hashing algorithm is used. **Here is the algorithm.**

The hash is performed on the filename without it's extension. (Example: if the container name is `SFX_Patch_1.snd` then the string `SFX_Patch_1` is hashed.)

In DOOM The Dark Ages, certain snd archives have special, hard-coded hashes instead of regular FNV hashes. Specifically, `sound/soundbanks/pc/MUSIC.snd` has a hash of 0, and `sound/soundbanks/pc/SFX.snd` has a hash of 1. The handheld versions of these archives in `sound/soundbanks/hhpc` have regular FNV hashes.

Sound Archives (.snd)

Sound archives store DOOM: Eternal and DOOM: The Dark Ages audio files (samples).

File Format

```
// Eternal version of sndEntry
struct sndEntry_Eternal
{
	uint64_t farmhash;    // Farmhash64 of the entry's DECODED data
    uint32_t sampleID;    // ID used to to reference this sample in the sound banks
    uint32_t encodedSize; // Size of the entry's data as it's stored in the archive.
    uint64_t offset;      // Location of entry's data in the file. Relative to beginning of
file
    uint32_t decodedSize; // Size of the entry's data after being decoded
    uint16_t encoding;    // Either 2 or 3
    uint16_t metaSize;    // Size of this entry's meta section inside the meta blob
    uint32_t metaOffset;  // Location of this entry's meta section. Relative to global offset
0xC
}

// Dark Ages version of sndEntry
struct sndEntry_DarkAges
{
	uint64_t farmhash;    // Farmhash64 of the entry's data
    uint32_t sampleID;    // ID used to to reference this sample in the sound banks
    uint32_t encodedSize; // Size of the entry's data as it's stored in the archive.
    uint64_t offset;      // Location of entry's data in the file. Relative to beginning of
file
    uint32_t decodedSize; // Always equal to encodedSize
    uint32_t metaSize;    // Size of this entry's meta section inside the meta blob
    uint32_t metaOffset;  // Location of this entry's meta section. Relative to global offset
0xC
}

struct sndHeaderChunk
{
```

```
uint32_t  metaSize;   // Size of the header chunk's meta blob + 4
    char*     metaBlob;   // Header chunk's meta blob. Length == metaSize - 4
    uint32_t  numEntries; // Number of entries
sndEntry* entries;    // Array of entry information. Length == numEntries. Exact format
depends on the game
}

struct snd
{
uint32_t version;    // Always 6
    uint32_t headerSize; // Size of the header chunk
    sndHeaderChunk;      // Header chunk. Exact size given by headerSize
    char* dataChunk;     // Data chunk. Stores the complete audio samples. Length ==
remainder of file
}
```

About: Meta Blob

Audio samples in both games use wems as their audio file format. A sample's "meta" section consists of the start of the file, up to and including the length field of it's `data` chunk.

Code that demonstrates how to calculate the length of a sample's meta chunk can be found **here**

Notes: Encoding Setting

This section concerns the `encoding` property of Eternal's .snd entries. As mentioned in the definition, this value can either be 2 or 3.

When the value is 3, this means the sample's data is stored as a .wem. All DOOM: The Dark Ages audio also behaves this way.

When the value is 2, this means the sample's data is compressed and stored as an OPUS file. For these entries, the `encodedSize` is the size of the OPUS file. The `decodedSize` will match the size of the original WEM. Additionally, the sample's meta chunk, normally unused, gets read and parsed by the game. Both the decodedSize and meta chunk matter. If their data is incorrect, the sample may fail to play, get cut off early, or even crash the game when played.

Effectively, an `encoding` value of 2 implies an archive-level compression of the audio data. It would be extremely difficult to create sound mods that utilize this setting without calling external audio conversion tools while loading mods, or without having access to both the OPUS and original WEM file. Thankfully, we can simply flip the encoding value from 2 to 3 and store the modded sample as a WEM

without consequence. This is an overwhelmingly easier option, with the only "tradeoff" being slightly more storage space required for WEM data.

soundmetadata.bin

Binary AudioKinetic metadata file. This file is present in DOOM Eternal and DOOM: The Dark Ages. However, their contents are very different (despite maintaining the same basic chunking structure)

The following is the file format for Dark Ages

```
// All values are Little-Endian

template<typename TYPE>
struct list_t {
	uint32_t num;
	TYPE* data;
}

struct string_t {
	uint32_t length;
    char* data; // Not null-terminated
}

// Case-insensitive (to lowercase) FNV hash
typedef uint32_t hash_t;

struct NameHash {
  string_t name;
  hash_t hash; // Hash of the above string
}

// This file loves to flip-flop between putting
// a string after it's hash, or before it's hash
struct HashName {
  hash_t hash; // Hash of the following string
  string_t name;
}

struct SoundEvent {
  NameHash name;
  uint8_t languageID; // 0 corresponds with "SFX" and 1 with "ENGLISH(US)"
  string_t languageString;
```

```
}

// Sections 4 and 5 of the soundmetadata have
// the exact same format. Hence we can reuse
// the same structure definitions for convenience
struct Section_4_5_Item {
  HashName name;
  list_t<HashName> stringlist;
}

struct SampleList_PlaybackTimes {
  string_t languageString;
  float playbackTime; // Playback time of the sample in seconds
}

struct SampleList_Sample {
  uint32_t sampleID;
  string_t languageString; // Language name, NOT the hashed string corresponding to the
sampleID
}

// Can be used to identify which audio samples are used
// in which soundbanks
struct SampleList {
  NameHash name; // Name of a soundbank
  uint8_t byte; // Always 0
  uint32_t word; // Unknown
  uint8_t flag0; // 0 or 1
  uint8_t flag1; // 0 or 1
  uint32_t word2; // Unknown

  // IMPORTANT: This list (including it's length field)
  // is present ONLY IF flag0 == 0
  // Only present in non-SFX sample lists
  list_t<SampleList_PlaybackTimes> playbackTimes;

  list_t<SampleList_Sample> samples;
}

struct SoundMetaData_DarkAges {
```

```
  list_t<SoundEvent> SoundEvents;
  list_t<HashName> section2; // Exact purpose unknown
  list_t<NameHash> section3; // Exact purpose unknown
  list_t<Section_4_5_Item> SoundSwitches;
  list_t<Section_4_5_Item> SoundStates;
  list_t<SampleList> SampleLists;

  // See https://wiki.eternalmods.com/books/8-reverse-engineering-file-
formats/page/containermask
  // For an explanation of this structure.
  sndContainerMask ContainerMask
}
```

MapResources Files

MapResources files are located in the .resource archives. Each map in the game (including global maps like `common` and `warehouse`) have a MapResource file.

Their resource type string is `file` and their paths are `generated/buildgame/<MAP_PATH>.mapresources`

For example: `generated/buildgame/game/sp/m1_intro/m1_intro.mapresources` is the MapResources file for Village of Khalim. You can see the full list of maps at the bottom of the `packagemapspec.json`

Purpose

idTech7 and idTech8 have a 3-layer system that controls what files are loaded by the game:

1. The `packagemapspec.json` controls what .resources and .streamdb archives are used in each level.
2. The **container.mask** determines what files in the available .resources archives can be loaded, and which are disabled.
3. The MapResources Files determines what files a level will actually load from the pool of available files.

Structure

The MapResources files are massive lists of asset paths. There are some small differences in the format between idTech7 and idTech8

```
struct string_t {
	u32 length; // Little Endian
    char* data; // Not null-terminated
}

template<typename TYPE>
struct list_t {
	u32 num; // Big Endian
    TYPE* data;
```

```
}

struct entry_t {
	u32 typeindex;     // Big-Endian. Index into the typeStrings array
    string_t filename;
    u64 layermask[3];  // Little-Endian. See next section for explanation
}

struct MapResources {
	u32 timestamp; // This field is absent in idTech8 mapresources

    // List of all layers used in this level
    // For global maps (i.e. common and warehouse)
    // this list will always be empty.
    list_t<string_t> layers;

    u32 unknown; // Always 0

    // Resource type strings. These are the capitalized versions
    // (i.e. for 'generated/decls/aicomponentlist/.../' the
    // type string will be 'aiComponentList')
    list_t<string_t> typeStrings;

    list_t<entry_t> entries;

    // Names of maps.
    // This list does not exist in idTech8 mapresources
    list_t<string_t> maps;
}
```

Layer Mask

Each entry in the MapResources has a 24-byte "layer mask". Each layer in the file's layer list is assigned one bit in the mask. Layer index 0 is given bit 0, layer index 1 is given bit 1, etc.

The game uses this bitmask to conditionally load (and presumably unload) resources based on what layers are currently active in the level. Using a layer-specific resource in layers it's not assigned to is a potential cause of instability if the original layer is not always active.

The last bit in the layer mask (the most-significant bit of the third u64) is a special flag. If this is flag is set, the layer mask is ignored and the entry's data will always be loaded. Every entry in the global

mapresources (common.mapresources, warehouse.mapresources, etc.) has this bit set, since those have no layers. However, even in level-specific mapresources files, the majority of entries also have this flag set.

In Dark Ages, where the layer system is mostly unused, the only files consistently bound to specific layers are AI animation files. These are bound to the `spawn_target_layer` of each level. In practice, this should mean they're always loaded anyways.

DOOM Eternal makes extensive use of the layer system, with some levels having more than 64 layers. Roughly ~10% of the level-specific mapresource entries use the layermask. Layer-bound entries can include (but aren't limited to) things like:

- AI Animation files (like in Dark Ages)
- Models, material2 decls and images for layer-specific level geometry
- Havok shapes for layer-specific trigger volumes.

Force-Loading Assets

Unlike the packagemapspec and container mask, the MapResources files don't seem to be a strict filter on what files can be loaded. It is evident that idTech can force-load unlisted assets that a listed asset is dependent on. This will occur while parsing the listed assets. Unfortunately, this behavior is not predictable, and understanding of when it occurs is mostly speculation. It *seems* that certain classes of decls have force-loading enabled, while others don't. This would explain why adding new assets to the game via modding sometimes works without editing the mapresources, and sometimes doesn't. The key factor is where those new assets are referenced.

However, idTech8 seems to have stricter rules for force-loading assets. This is likely due to the pre-serialization of entities and entityDef files. Serialized files seem unable to be force-loaded, and don't have forced-loading enabled for their dependencies.

When an asset is force-loaded in this manner, the console will log a message like this:

> WARNING: idResourceStorageDiskStreamer::GetFile failed to find entry for 'generated/decls/damage/damage/dummy.decl' while loading damage:damage/dummy from edit.damageDecls.item from aiDamageDeclCollection:slayer/mace

This message was logged in Dark Ages, and can be interpreted as followed: "While loading an idDeclAIDamageDeclCollection" file, an unlisted damage decl with path "damage/dummy" was encountered in one of it's properties. This file was force-loaded despite not being in any mapresources entry list.