AARU

AaruFormat Specification

Author: Nat Portillo
Contact: claunia@claunia.com

Version: 2.0 FINAL DRAFT
Date: 2025-07-31

AaruFormat Specification

Version history

Date	Version	Branch	Author	Modifications
08 May 2022	1.0	Official	Natalia Portillo	Initial version
18 May 2022	2.0d1	Draft	Natalia Portillo	Update version. Add stub for new deduplication table. Add stub for new media type.
04 Sep 2022	2.0d2	Draft	Rebecca Wallander	Add flux data definitions.
15 Sep 2022	2.0d3	Draft	Natalia Portillo	Fix some typos. Add index continuation block. Add track layout block.
15 Sep 2022	2.0d4	Draft	Natalia Portillo	Define deduplication table version 2. Define twin sector table.
16 Sep 2022	2.0d5	Draft	Natalia Portillo	Define snapshot block.
16 Sep 2022	2.0d6	Draft	Natalia Portillo	Define parent block.
16 Sep 2022	2.0d7	Draft	Natalia Portillo	Rework flux block to use deduplication tables.
16 Sep 2022	2.0d7	Draft	Natalia Portillo	Define bitstream block. Add annex explaining the meaning and relationship between user, bitstream and flux data.
31 July 2025	2.0df	Draft	Natalia Portillo	Final draft. Move specification to Asciidoc. Deprecate Compact Disc lead-in, first track pregap, lead-out, and floppy disk lead-out data types. They are stored as user data now (with negative and overflow sectors as appropiate).

Table of Contents

Chapter 1. Introduction

This document is the detailed specification of AaruFormat.

1.1. Audience

This specification is directed to emulator developers, software preservators, archives, museums and collectors, that want to have a common file format where to store, archive and manage, dumps and copies of any type of computer storage.

1.2. Scope

The scope of this specification is to define an open, free and universal file format able to store and describe any kind of digital or analog storage media for computer systems, in a clear and extensible way that allows for new media to be easily added, along with any kind of metadata describing them, plus verification and recovery data.

Currently the idea is for it to be able to store punch cards, disks (magnetic, optical, magnetoptical) and tapes (analog and digital tapes), decoded or as audio tones and as magnetic or optical fluxes, with any kind of copy protection or absence of it.

Because of its design goals, the format here described may not be the best for reproduction or emulation, but it pretends to be the best for archival and preservation.

There are other formats pretending to achieve some of these goals, and precisely that's why this format is designed. To be a single, universal, extensible, standard, eliminating the need to use a different format for each type of storage.

Chapter 2. Definitions

2.1. Types

All binary types used in this specification are stored as little-endian values on the file. This specification follows the C syntax to denote hexadecimal values, and requires the reader have some knowledge on programming.

2.2. Endianness

Unless otherwise specified, all fields in this specification are considered to be in *Little-Endian* format, that is the hexadecimal number `0x12345678` is stored in disk as the following sequence of bytes: `0x78 0x56 0x34 0x12`.

2.3. Header identifiers

Header identifiers are 4 `ASCII` characters stored as a sequence of bytes inside a single 32 bits pack. They are shown in this specification enclosed in single quotes. For example, the header identifier `AARU` should be stored on disk as `0x41 0x41 0x52 0x55`.

2.4. Integers

Integer values are designated in this specification if unsigned (U) and no letter for signed, continuing with `int`, the number of bits able to be stored in them, and finishing with `_t`.

That so, the signed integers should be: `int8_t`, `int16_t`, `int32_t`, `int64_t` and `int128_t`.

And the unsigned integers should be: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` , `uint128_t`.

2.5. Strings

All strings are stored as a sequence of bytes, in Unicode's `UTF-16` little endian encoding and terminated and filled with `NULL` (`0x00`) bytes.

`String8` values mean the string is stored in Unicode's `UTF-8` encoding and terminated and filled with `NULL` (`0x00`) bytes.

`StringA` values mean the string is stored in `ASCII` encoding and terminated and filled with `NULL` (`0x00`) bytes.

2.6. Timestamp

All timestamps used in this specification are stored as a signed 64bit integer (`int64_t`) counting the number of nanoseconds in the UTC timezone after/before the epoch of 1st January 1601 at 00:00 of the Gregorian Calendar. This epoch is chosen because it is when the leap-year scheme was adopted.

2.7. Media tag

A media tag is a piece of data that is physically present in the media but it’s not part of the user data. It can be the table of contents, some manufacturing information, sector replacement tables, etc.

2.8. Sector tag

A sector tag is a piece of data that is physically present in the media, once per each sector, but it’s not part of the user data. It can be addressing information, error detection or correction information, encryption metadata, etc.

2.9. NULL

NULLs are `0x00` bytes.

Chapter 3. Master Header Structure

The AaruHeaderV2 is the fundamental header present at the beginning of every AaruFormat file. It defines the image's versioning, metadata, layout offset, feature compatibility, and structural alignment. All subsequent parsing and interpretation of the file depends on the contents of this header.

3.1. Structure Definition

```
#define HEADER_APP_NAME_LEN 64
#define GUID_SIZE 16
/**Header, at start of file */
typedef struct AaruHeaderV2
{
    /**Header identifier, see AARU_MAGIC */
    uint64_t  identifier;
    /**UTF-16LE name of the application that created the image */
    uint8_t   application[HEADER_APP_NAME_LEN];
    /**Image format major version. A new major version means a possibly incompatible
change of format */
    uint8_t   imageMajorVersion;
    /**Image format minor version. A new minor version indicates a compatible change of
format */
    uint8_t   imageMinorVersion;
    /**Major version of the application that created the image */
    uint8_t   applicationMajorVersion;
    /**Minor version of the application that created the image */
    uint8_t   applicationMinorVersion;
    /**Type of media contained on image */
    uint32_t  mediaType;
    /**Offset to index */
    uint64_t  indexOffset;
    /**Windows filetime (100 nanoseconds since 1601/01/01 00:00:00 UTC) of image
creation time */
    int64_t   creationTime;
    /**Windows filetime (100 nanoseconds since 1601/01/01 00:00:00 UTC) of image last
written time */
    int64_t   lastWrittenTime;
    /**Unique identifier that allows children images to recognize and find this image
*/
    uint8_t   guid[GUID_SIZE];
    /**Block alignment shift. All blocks in the image are aligned at 2 <<
blockAlignmentShift bytes */
    uint8_t   blockAlignmentShift;
    /**Data shift. All data blocks in the image contain 2 << dataShift items at most */
    uint8_t   dataShift;
    /**Table shift. All deduplication tables in the image use this shift to calculate
the position of an item */
    uint8_t   tableShift;
    /**Features used in this image that if unsupported are still compatible for reading
and writing implementations */
    uint64_t  featureCompatible;
    /**Features used in this image that if unsupported are still compatible for reading
implementations but not for writing */
```

```
    uint64_t  featureCompatibleRo;
    /**Features used in this image that if unsupported prevent reading or writing the
image */
    uint64_t  featureIncompatible;
} AaruHeaderV2;
```

3.2. Field Descriptions

Name	Type	Description
identifier	uint64_t	Header identifier constant. Must match the predefined `AARU_MAGIC` value to validate the format.
application	uint8_t[HEADER_APP_NAME_LEN]	UTF-16LE encoded name of the application responsible for creating the image. Length is defined by `HEADER_APP_NAME_LEN`.
imageMajorVersion	uint8_t	Major version of the AaruFormat structure. A bump indicates potential breaking changes.
imageMinorVersion	uint8_t	Minor version of the format, for backward-compatible structural updates.
applicationMajorVersion	uint8_t	Major version of the creating application.
applicationMinorVersion	uint8_t	Minor version of the application.
mediaType	uint32_t	Media type identifier denoting the nature of the captured content (e.g., floppy, optical disc, tape).
indexOffset	uint64_t	Absolute file offset to the beginning of the index structure, used to locate blocks throughout the image.
creationTime	int64_t	Timestamp (Windows filetime) representing when the image was first created.
lastWrittenTime	int64_t	Timestamp (Windows filetime) of the last modification made to the image.
guid	uint8_t[GUID_SIZE]	Globally Unique Identifier (GUID) that allows linking of related image derivatives and child snapshots. Length is defined by `GUID_SIZE`.
blockAlignmentShift	uint8_t	Determines block alignment boundaries using the formula 2 << blockAlignmentShift.
featureCompatible	uint64_t	Bitmask of features that, even if not implemented, still allow reading and writing the image.

Name	Type	Description
featureCompatibleRo	uint64_t	Bitmask of features that allow read-only processing of the image if unsupported.
featureIncompatible	uint64_t	Bitmask of features that must be supported to read or write the image at all.

Chapter 4. The Blocks

The blocks in AaruFormat serve as the building components of the image, containing both the data and metadata extracted from the media it represents.

4.1. Index Block (`INDX`) *DEPRECATED*

The index block stores references to all blocks present in the file. It is composed of a header, followed by a sequence of entries, the count of which is defined within the header.

Multiple index blocks may exist within a file to represent previous states or historical versions; however, only the final index block must be referenced by the main file header.

Deprecation Notice: This block is deprecated and **MUST NOT** be used in new image files.

4.1.1. Structure Definition

```c
#define INDEX_MAGIC 0x58444E49
/**Header for the index, followed by entries */
typedef struct IndexHeader
{
    /**Identifier, <see cref="BlockType.Index" /> */
    uint32_t identifier;
    /**How many entries follow this header */
    uint16_t entries;
    /**CRC64-ECMA of the index */
    uint64_t crc64;
} IndexHeader;
```

4.1.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The index block identifier, always `INDX`
uint16_t	2 bytes	entries	The number of entries following this header
uint64_t	8 bytes	crc64	CRC64-ECMA checksum of the entries following this header

4.1.3. Index entries

```c
/**Index entry */
typedef struct IndexEntry
{
    /**Type of item pointed by this entry */
    uint32_t blockType;
    /**Type of data contained by the block pointed by this entry */
    uint32_t dataType;
    /**Offset in file where item is stored */
    uint64_t offset;
} IndexEntry;
```

4.1.4. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	blockType	The type of block this entry points to.
uint32_t	4 bytes	dataType	The type of data the block pointed by this entry contains.
uint64_t	8 bytes	offset	The offset in bytes from the start of the file where the block pointed by this entry starts.

4.2. Index Block version 2 (IDX2)

The index block stores references to all blocks present in the file. It is composed of a header, followed by a sequence of entries, the count of which is defined within the header.

Multiple index blocks may exist within a file to represent previous states or historical versions; however, only the final index block must be referenced by the main file header.

4.2.1. Structure Definition

```c
#define INDEX2_MAGIC 0x32584449
/**Header for the index, followed by entries */
typedef struct IndexHeader2
{
    /**Identifier, <see cref="BlockType.Index" /> */
    uint32_t identifier;
    /**How many entries follow this header */
    uint64_t entries;
    /**CRC64-ECMA of the index */
    uint64_t crc64;
} IndexHeader;
```

4.2.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The index block identifier, always `IDX2`
uint64_t	8 bytes	entries	The number of entries following this header
uint64_t	8 bytes	crc64	CRC64-ECMA checksum of the entries following this header

4.2.3. Index entries

```c
/**Index entry */
typedef struct IndexEntry
{
    /**Type of item pointed by this entry */
    uint32_t blockType;
    /**Type of data contained by the block pointed by this entry */
    uint32_t dataType;
    /**Offset in file where item is stored */
    uint64_t offset;
} IndexEntry;
```

4.2.4. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	blockType	The type of block this entry points to.
uint32_t	4 bytes	dataType	The type of data the block pointed by this entry contains.
uint64_t	8 bytes	offset	The offset in bytes from the start of the file where the block pointed by this entry starts.

4.3. Index Block Continuation (`IDXC`)

The index block continuation follows the same structure and semantics as the index block version 2, with the exception that it includes a pointer to the preceding index block or index block continuation.

At most, a single index block continuation may appear within any index block structure.

The purpose of this block is to enable incremental indexing prior to finalizing the image file. This allows new blocks to be indexed as they are written, facilitating partial recovery in the event of application failure.

The block is immediately followed by index entries formatted identically to those defined in index block version 2.

4.3.1. Structure Definition

```
/* Undefined */
```

4.3.2. Field Descriptions

Type	Size	Name	Description
uint32	4 bytes	identifier	The index block identifier, always `IDXC`
uint64	8 bytes	entries	The number of entries following this header
uint64	8 bytes	crc64	CRC64-ECMA checksum of the entries following this header
uint64	8 bytes	previous	Pointer in image file to previous index block

4.4. Data Block (`DBLK`)

A data block encapsulates media-derived content and is composed of a header followed by either compressed or uncompressed data.

The contents of a data block may represent user data—such as media sectors—or auxiliary data elements, including media or sector-specific tags.

When a data block includes multiple items (e.g., sectors or sector tags), the `sectorSize` field specifies the size, in bytes, of each individual item. Conversely, if the block contains a single item (e.g., media tags), `sectorSize` must be set to 0.

4.4.1. Structure Definition

```c
#define DATABLOCK_MAGIC 0x4B4C4244
/**Block header, precedes block data */
typedef struct BlockHeader
{
    /**Identifier, <see cref="BlockType.DataBlock" /> */
    uint32_t identifier;
    /**Type of data contained by this block */
    uint32_t type;
    /**Compression algorithm used to compress the block */
    uint16_t compression;
    /**Size in uint8_ts of each sector contained in this block */
    uint32_t sectorSize;
    /**Compressed length for the block */
    uint32_t cmpLength;
    /**Uncompressed length for the block */
    uint32_t length;
    /**CRC64-ECMA of the compressed block */
    uint64_t cmpCrc64;
    /**CRC64-ECMA of the uncompressed block */
    uint64_t crc64;
} BlockHeader;
```

4.4.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The data block identifier, always `DBLK`
uint16_t	2 bytes	type	The data type contained in this block. See Annex B.
uint16_t	2 bytes	compression	The compression algorithm used in the data. See Annex C.
uint32_t	4 bytes	sectorSize	The size in bytes of the sectors contained in this data block if applicable.
uint32_t	4 bytes	cmpLength	The size in bytes of the compressed data that follows this header.

Type	Size	Name	Description
uint32_t	4 bytes	length	The size in bytes of the data block when decompressed.
uint64_t	8 bytes	cmpCrc64	The CRC64-ECMA checksum of the compressed data that follows this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the decompressed data.

4.5. Deduplication Table (DDT*) *DEPRECATED*

The deduplication table is a sequential array of pointers, with each entry corresponding to a sector on the storage media. These pointers map sector data to logical content blocks, enabling efficient elimination of duplicate data. Every image must include at least one deduplication table of type `UserData`.

Deprecation Notice: This block is deprecated and **MUST NOT** be used in new image files.

4.5.1. Structure Definition

```
#define DDT_MAGIC 0X2A544444
/**Header for a deduplication table. Table follows it */
typedef struct DdtHeader
{
    /**Identifier, <see cref="BlockType.DeDuplicationTable" /> */
    uint32_t identifier;
    /**Type of data pointed by this DDT */
    uint32_t type;
    /**Compression algorithm used to compress the DDT */
    uint16_t compression;
    /**Each entry is ((uint8_t offset in file) &lt;&lt; shift) + (sector offset in
block) */
    uint8_t  shift;
    /**How many entries are in the table */
    uint64_t entries;
    /**Compressed length for the DDT */
    uint64_t cmpLength;
    /**Uncompressed length for the DDT */
    uint64_t length;
    /**CRC64-ECMA of the compressed DDT */
    uint64_t cmpCrc64;
    /**CRC64-ECMA of the uncompressed DDT */
    uint64_t crc64;
} DdtHeader;
```

4.5.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The deduplication table identifier, always `DDT*`
uint16_t	2 bytes	type	The data type pointed by this table. See Annex B.
uint16_t	2 bytes	compression	The compression algorithm used in the table. See Annex C.
uint8_t	1 byte	shift	The shift used to calculate the position of a sector in a data block pointed by this table.
uint64_t	8 bytes	entries	How many pointers follow this header.
uint32_t	4 bytes	cmpLength	The size in bytes of the compressed table that follows this header.

Type	Size	Name	Description
uint32_t	4 bytes	length	The size in bytes of the table block when decompressed.
uint64_t	8 bytes	cmpCrc64	The CRC64-ECMA checksum of the compressed table that follows this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the decompressed table.

4.5.3. Deduplication Table Entries

Each entry in the deduplication table references a specific data block and a particular item within that block.

Mapping Logic

- Entry 0 corresponds to data associated with LBA 0 of the media; subsequent entries map sequentially.
- The pointer value for an entry is computed using the formula:

  ```
  pointer = (byte_offset_of_block << shift) + item_index_in_block
  ```

  For example, a raw pointer value of `0x8003` in a table with a `shift` of 5 resolves as follows:

- Byte offset: `0x400` → `1024`
- Item index: `0x3` → `3`
- Therefore, the pointer targets item 3 within the data block located at byte offset `1024` in the file.

4.5.4. Special Case – Corrected Sector Tables

Deduplication tables of type `CdSectorPrefixCorrected` and `CdSectorSuffixCorrected` split the entry value using bitmasking:

- Pointer component: `entry & 0x00FFFFFF`
- Flags component: `entry & 0xFF000000`

Flags

Flag	Value	Description
None	`0x00000000`	The suffix or prefix cannot be regenerated as is stored in the pointed data block.
NotDumped	`0x10000000`	The sector has not been dumped. Ignore the pointer.
Correct	`0x20000000`	The suffix (only for MODE 1 sectors) or prefix is correct and can be regenerated. Ignore the pointer.

Flag	Value	Description
Mode2Form1Ok	`0x30000000`	The suffix for MODE 2 sectors is correct, can be regenerated, and corresponds to a MODE 2 Form 1 sector.
Mode2Form2Ok	`0x40000000`	The suffix for MODE 2 sectors is correct, can be regenerated, and corresponds to a MODE 2 Form 2 sector with a valid CRC.
Mode2Form2NoCrc	`0x50000000`	The suffix for MODE 2 sectors is correct, can be regenerated, and corresponds to a MODE 2 Form 2 sector with an empty CRC.

4.6. Deduplication Table (`DDT2`)

The deduplication table is a multi-level table of pointers to LBAs contained in the image. It starts with the following header.

```
/* Undefined */
```

4.6.1. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The deduplication table identifier, always `DDT2` or `DDTS`. The first level of a table is always `DDT2` and its presence is mandatory. Subtables will have `DDTS`
uint16_t	2 bytes	type	The data type pointed by this table. See Annex B.
uint16_t	2 bytes	compression	The compression algorithm used in the table. See Annex C.
uint8_t	1 byte	levels	How many levels of subtables are present. 1 means this is the only level.
uint8_t	1 byte	tableLevel	What level does this table correspond to
uint64_t	8 bytes	previousLevel	Pointer to absolute byte offset in the image file where the previous table level resides
uint16_t	2 bytes	negative	The negative displacement of LBA numbers. For media that can have negative LBAs, this establishes the number to substract to the table entry number
uint64_t	8 bytes	start	The first LBA contained in this table. It must be 0 for 'DDT2' blocks and can be other number for subtables 'DDTS'
uint8_t	1 byte	alignment	Shift of alignment of all blocks in the image. This must be the same in all deduplication tables and subtables.
uint8_t	1 byte	shift	The shift used to calculate the position of a sector in a data block pointed by this table, or how many sectors are pointed by the next level.
uint8_t	1 byte	sizeType	Size type (see table below)
uint64_t	8 bytes	entries	How many pointers follow this header.
uint32_t	4 bytes	cmpLength	The size in bytes of the compressed table that follows this header.
uint32_t	4 bytes	length	The size in bytes of the table block when decompressed.

Type	Size	Name	Description
uint64_t	8 bytes	cmpCrc64	The CRC64-ECMA checksum of the compressed table that follows this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the decompressed table.

The size type defines the following type of entries:

Type	Value	Description
Mini	0	Each entry uses two bytes, with the leftmost byte (mask 0xFF00) used for flags, and the rightmost byte used as a pointer to the sector or next level.
Small	1	Each entry uses three bytes, with the leftmost byte used for flags and the next two bytes used as a pointer to the sector or next level.
Medium	2	Each entry uses four bytes, with the leftmost byte (mask 0xFF000000) used for flags and the next three bytes used as a pointer to the sector or next level.
Big	3	Each entry uses five bytes, with the leftmost byte used for flags and the next three bytes used as a pointer to the sector or next level.

4.6.2. Sector Pointer Resolution and Table Levels

When `levels` is equal to 1—indicating a single-level deduplication table—each entry in the table corresponds directly to a media sector. The pointer value is resolved using the following procedure:

- Right-shift the raw pointer value by the `shift` value.
- Multiply the result by the `alignment` to compute the absolute byte offset of the target data block.
- The remainder of the original pointer value modulo `(1 << shift)` yields the item index within the block.

Each data block stores a fixed number of bytes per sector, allowing compact and efficient sector addressing.

For example: Given a pointer value of `0x8003`, a `shift` of 5, and an `alignment` of 9: - `0x8003 >> 5 = 0x400 = 1024` - `1024 * 9 = 9216` - The sector index within the block is `0x8003 & 0x1F = 3`

Thus, the sector is located at byte offset `9216`, and it is the 3rd item in the block.

Multi-Level Tables

When `levels > 1`, the interpretation of pointer entries changes substantially. Although typical usage involves no more than two levels, implementations **MUST** be capable of handling an arbitrary number of levels to ensure forward compatibility.

At each level—except the final—the table entry functions as an address to the next-level table. The range of LBAs covered by each entry is calculated as:

```
range = entry_index * (1 << shift)^(levels - 1)
```

For example, with a `shift` value of 9 and two levels: - Entry `0` spans LBAs `0–511` - Entry `1` spans LBAs `512–1023`

With three levels: - Entry `0` at level 0 spans LBAs `0–262143` - Entry `0` at level 1 within that region spans LBAs `0–511`, and so on recursively.

Resolution Example

To locate sector `1012` using a two-level table with `shift = 9` and `alignment = 9`:

1. **Level 0**:
    - Sector `1012` falls within entry `1` (covers `512–1023`)
    - Entry `1` contains the value `0x12000`
    - Multiply by `alignment` → `0x12000 * 9 = 0x225000 = 37,748,736`
    - Read the next-level table at byte offset `37,748,736`, marked with the identifier `DDTS`
2. **Level 1**:
    - The relevant entry is `500` (`1012 - 512 = 500`)
    - Entry `500` contains `0x35006`
    - Right-shift `0x35006 >> 9 = 0x6A = 106`
    - Multiply by `alignment`: `106 * 9 = 954`
    - Sector resides at byte offset `217,088` and is the 6th item in the block (`0x35006 & 0x1FF = 6`)

Deduplication table flags

Flag	Value	Description
NotDumped	`0x00`	The sector(s) have not been dumped
Dumped	`0x01`	The sector(s) have been dumped without errors
Errored	`0x02`	The sector(s) returned an error on dumping
Mode1Correct	`0x03`	The sector is MODE 1 and the suffix or prefix is correct and can be regenerated. Must only appear on deduplications tables with types CdSectorPrefixCorrected or CdSectorSuffixCorrected
Mode2Form1Ok	`0x04`	The suffix for MODE 2 sectors is correct, can be regenerated, and corresponds to a MODE 2 Form 1 sector. Must only appear on deduplications tables with type CdSectorSuffixCorrected
Mode2Form2Ok	`0x05`	The suffix for MODE 2 sectors is correct, can be regenerated, and corresponds to a MODE 2 Form 2 sector with a valid CRC. Must only appear on deduplications tables with type CdSectorSuffixCorrected

Flag	Value	Description
Mode2Form2NoCrc	`0x06`	The suffix for MODE 2 sectors is correct, can be regenerated, and corresponds to a MODE 2 Form 2 sector with an empty CRC. Must only appear on deduplications tables with type CdSectorSuffixCorrected
Twin	`0x07`	The pointer contains a “twin” sector table (see below)
Unrecorded	`0x08`	The sector was unrecorded and each re-read returns random data

When flags are present in a table that has sublevels it applies to all the sectors that shall be present in the subtable, unless the flag specify something else.

4.7. Twin Sector Table (`TWTB`)

This table enumerates hardware sectors that share an identical sector number. Such sectors are referred to as "twin sectors," although the grouping may consist of more than two instances. The associated pointer is resolved following the same logic applied in a last-level deduplication table.

```
/* Undefined */
```

4.7.1. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The twin sector table identifier, always `TWTB`
uint8_t	1 byte	alignment	Shift of alignment of all blocks in the image. This must be the same in all deduplication tables and subtables.
uint8_t	1 byte	shift	The shift used to calculate the position of a sector in a data block pointed by this table, or how many sectors are pointed by the next level.
uint64_t	8 bytes	entries	How many pointers follow this header.
uint32_t	4 bytes	length	The size in bytes of the table block.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the decompressed table.

4.7.2. Twin sector entries

Type	Size	Name	Description
uint32_t	8 bytes	pointer	Pointer to the sector.

4.7.3. Pointer-Based Data Block Resolution

To determine the corresponding data block:

1. Right-shift the pointer value using the specified `shift` parameter.
2. Multiply the result by the `alignment` value.
3. The remainder from this operation indicates the sector's offset within the target data block.

Each data block contains a fixed number of bytes per sector, which remains constant across blocks. This invariant size allows for more efficient storage of pointer values.

Example

Given the following parameters:

- Pointer Value: `0x8003`
- Shift Value: `5`
- Alignment: `9`

The data block is located at byte offset `524288`. The sector referenced by the pointer is the **third entry** within this block.

4.8. Geometry Block (`GEOM`)

The geometry block encapsulates metadata that defines the disk’s geometry, primarily to support transformations between CHS (Cylinder-Head-Sector) and LBA (Logical Block Addressing) addressing schemes.

Note that the stored geometry may not reflect the media’s actual physical layout. Instead, it typically represents the translation parameters active at the time the drive image was acquired.

```c
#define GEOM_MAGIC 0x4D4F4547
/**Geometry block, contains physical geometry information */
typedef struct GeometryBlockHeader
{
    /**Identifier, <see cref="BlockType.GeometryBlock" /> */
    uint32_t identifier;
    uint32_t cylinders;
    uint32_t heads;
    uint32_t sectorsPerTrack;
} GeometryBlockHeader;
```

4.8.1. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The geometry table identifier, always `GEOM`
uint32_t	4 bytes	cylinders	The number of cylinders.
uint32_t	4 bytes	heads	The number of heads.
uint32_t	4 bytes	sectorsPerTrack	The number of sectors per track.

4.9. Metadata Block (META)

The metadata block contains descriptive information related to the media source, which is not part of the original media data itself. Typical fields may include the manufacturer name, device model, acquisition sequence identifiers, and other contextual attributes.

All string values within this block are encoded as little-endian UTF-16 and terminated with a null character.

```
#define META_MAGIC 0x4154454D
/**Metadata block, contains metadata */
typedef struct MetadataBlockHeader
{
    /**Identifier, <see cref="BlockType.MetadataBlock" /> */
    uint32_t identifier;
    /**Size in uint8_ts of this whole metadata block */
    uint32_t blockSize;
    /**Sequence of media set this media beint64_ts to */
    int32_t  mediaSequence;
    /**Total number of media on the media set this media beint64_ts to */
    int32_t  lastMediaSequence;
    /**Offset to start of creator string from start of this block */
    uint32_t creatorOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t creatorLength;
    /**Offset to start of creator string from start of this block */
    uint32_t commentsOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t commentsLength;
    /**Offset to start of creator string from start of this block */
    uint32_t mediaTitleOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t mediaTitleLength;
    /**Offset to start of creator string from start of this block */
    uint32_t mediaManufacturerOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t mediaManufacturerLength;
    /**Offset to start of creator string from start of this block */
    uint32_t mediaModelOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t mediaModelLength;
    /**Offset to start of creator string from start of this block */
    uint32_t mediaSerialNumberOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t mediaSerialNumberLength;
    /**Offset to start of creator string from start of this block */
    uint32_t mediaBarcodeOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t mediaBarcodeLength;
    /**Offset to start of creator string from start of this block */
    uint32_t mediaPartNumberOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t mediaPartNumberLength;
    /**Offset to start of creator string from start of this block */
    uint32_t driveManufacturerOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t driveManufacturerLength;
    /**Offset to start of creator string from start of this block */
    uint32_t driveModelOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t driveModelLength;
    /**Offset to start of creator string from start of this block */
    uint32_t driveSerialNumberOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t driveSerialNumberLength;
    /**Offset to start of creator string from start of this block */
    uint32_t driveFirmwareRevisionOffset;
    /**Length in uint8_ts of the null-terminated UTF-16LE creator string */
    uint32_t driveFirmwareRevisionLength;
} MetadataBlockHeader;
```

4.9.1. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The metadata table identifier, always `META`
uint32_t	4 bytes	blockSize	The size of this block including all of its data.
int32_t	4 bytes	mediaSequence	The number of heads.
int32_t	4 bytes	lastMediaSequence	The number of sectors per track.
uint32_t	4 bytes	creatorOffset	Offset to start of creator string from start of this block.
uint32_t	4 bytes	creatorLength	Length in bytes of the creator string.
uint32_t	4 bytes	commentsOffset	Offset to start of comments string from start of this block.
uint32_t	4 bytes	commentsLength	Length in bytes of the comments string.
uint32_t	4 bytes	mediaTitleOffset	Offset to start of media title string from start of this block.
uint32_t	4 bytes	mediaTitleLength	Length in bytes of the media title string.
uint32_t	4 bytes	mediaManufacturerOffset	Offset to start of media manufacturer string from start of this block.
uint32_t	4 bytes	mediaManufacturerLength	Length in bytes of the media manufacturer string.
uint32_t	4 bytes	mediaModelOffset	Offset to start of media model string from start of this block.

Type	Size	Name	Description
uint32_t	4 bytes	mediaModelLength	Length in bytes of the media model string.
uint32_t	4 bytes	mediaSerialNumberOffset	Offset to start of media serial number string from start of this block.
uint32_t	4 bytes	mediaSerialNumberLength	Length in bytes of the media serial number string.
uint32_t	4 bytes	mediaBarcodeOffset	Offset to start of media barcode string from start of this block.
uint32_t	4 bytes	mediaBarcodeLength	Length in bytes of the media barcode string.
uint32_t	4 bytes	mediaPartNumberOffset	Offset to start of media part number string from start of this block.
uint32_t	4 bytes	mediaPartNumberLength	Length in bytes of the media part number string.
uint32_t	4 bytes	driveManufacturerOffset	Offset to start of drive manufacturer string from start of this block.
uint32_t	4 bytes	driveManufacturerLength	Length in bytes of the drive manufacturer string.
uint32_t	4 bytes	driveModelOffset	Offset to start of drive model string from start of this block.
uint32_t	4 bytes	driveModelLength	Length in bytes of the drive model string.
uint32_t	4 bytes	driveSerialNumberOffset	Offset to start of drive serial number string from start of this block.
uint32_t	4 bytes	driveSerialNumberLength	Length in bytes of the drive serial number string.
uint32_t	4 bytes	driveFirmwareRevisionOffset	Offset to start of drive firmware revision string from start of this block.
uint32_t	4 bytes	driveFirmwareRevisionLength	Length in bytes of the drive firmware revision string.

4.10. Tracks Block (TRKS)

The tracks block holds a structured list of track entries, typically aligned with the layout specified in the table of contents or a similar indexing schema. This format is common in optical media such as CDs, DVDs, and related disc-based formats.

4.10.1. Structure Definition

```
#define TRACKS_MAGIC 0x534B5254
/**Contains list of optical disc tracks */
typedef struct TracksHeader
{
    /**Identifier, <see cref="BlockType.TracksBlock" /> */
    uint32_t identifier;
    /**How many entries follow this header */
    uint16_t entries;
    /**CRC64-ECMA of the block */
    uint64_t crc64;
} TracksHeader;
```

4.10.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The tracks block identifier, always TRKS
uint16_t	2 bytes	entries	The number of entries following this header
uint64_t	8 bytes	crc64	CRC64-ECMA checksum of the entries following this header

4.10.3. Track entries

```
/**Optical disc track */
typedef struct TrackEntry
{
    /**Track sequence */
    uint8_t sequence;
    /**Track type */
    uint8_t type;
    /**Track starting LBA */
    int64_t start;
    /**Track last LBA */
    int64_t end;
    /**Track pregap in sectors */
    int64_t pregap;
    /**Track session */
    uint8_t session;
    /**Track's ISRC in ASCII */
    uint8_t isrc[13];
    /**Track flags */
    uint8_t flags;
} TrackEntry;
```

4.10.4. Field Descriptions

Type	Size	Name	Description
uint8	1 byte	sequence	Track number.
uint8	1 byte	type	Track type (see table below).
int64	8 bytes	start	Track starting LBA (including pregap).
int64	8 bytes	end	Track ending LBA.
int64	8 bytes	pregap	Size of track’s pregap in sectors.
uint8	1 byte	session	Session the track belongs to.
StringA	13 bytes	isrc	Track’s ISRC in ASCIIZ.
uint8	1 byte	flags	Track flags as indicated in TOC if applicable.

4.10.5. Track Types

Type	Value	Description
Audio	0	All sectors in the track contain audio as defined by the Red Book.
Data	1	All sectors in the track contain user data that is not defined by any of the following types.
CdMode1	2	All sectors in the track contain user data according to MODE 1 as defined by the Yellow Book.
CdMode2Formless	3	All sectors in the track contain user data according to MODE 2 as defined by the Yellow and Green Books. Not all sectors belong to the same Form.
CdMode2Form1	4	All sectors in the track contain user data according to MODE 2 Form 1 as defined by the Yellow and Green Books. All sectors belong to the same Form.
CdMode2Form2	5	All sectors in the track contain user data according to MODE 2 Form 2 as defined by the Yellow and Green Books. All sectors belong to the same Form.

4.11. CICM XML Metadata Block (`CICM`)

This block header signifies the inclusion of an embedded CICM XML metadata sidecar. The contents of the XML are preserved in their original form and are not parsed, interpreted, or validated by the format implementation.

4.11.1. Structure Definition

```
#define CICM_MAGIC 0x4D434943
/**Header for the CICM XML metadata block */
typedef struct CicmMetadataBlock
{
    /**Identifier, <see cref="BlockType.CicmBlock" /> */
    uint32_t identifier;
    uint32_t length;
} CicmMetadataBlock;
```

4.11.2. Field Descriptions

Type	Size	Name	Description
uint32	4 bytes	identifier	The CICM XML metadata table identifier, always `CICM`
uint32	4 bytes	length	The size in bytes of the embedded CICM XML metadata that follows this header.

4.12. Checksum Block (CKSM)

This block stores an array of checksums corresponding to the user data embedded in the image. For media formats such as CompactDisc, the checksum is calculated over the complete sector—comprising the prefix, user data, and suffix—totaling 2352 bytes.

If the image is modified, the checksum block is considered outdated and should be either removed or excluded from the most recent index to ensure integrity.

4.12.1. Structure Definition

```
#define CHECKSUM_MAGIC 0x4D534B43
/**
 *     Checksum block, contains a checksum of all user data sectors (except for optical
discs that is 2352 uint8_ts raw
 *     sector if available
 *  */
typedef struct ChecksumHeader
{
    /**Identifier, <see cref="BlockType.ChecksumBlock" /> */
    uint32_t identifier;
    /**Length in uint8_ts of the block */
    uint32_t length;
    /**How many checksums follow */
    uint8_t  entries;
} ChecksumHeader;
```

4.12.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The tracks block identifier, always CKSM
uint32_t	4 bytes	length	The length in bytes of the data following this header.
uint8_t	1 byte	entries	The number of entries following this header

4.12.3. Checksum entries

```
/**Checksum entry, followed by checksum data itself */
typedef struct ChecksumEntry
{
    /**Checksum algorithm */
    uint8_t  type;
    /**Length in uint8_ts of checksum that follows this structure */
    uint32_t length;
} ChecksumEntry;
```

4.12.4. Field Descriptions

Type	Size	Name	Description
uint8_t	1 byte	type	Checksum algorithm.
uint32_t	4 bytes	length	Size in bytes of the checksum that immediately follows this entry.

4.12.5. Checksum algorithms

Type	Value	Description
Invalid	0	Invalid checksum entry, skip.
Md5	1	MD5
Sha1	2	SHA1
Sha256	3	SHA-256
SpamSum	4	SpamSum

4.13. Data Position Measurement Block (`DPM*`)

This block captures measurements of each sector’s position, providing insights into the physical structure of the disc. It is designed to facilitate analysis of disc geometry and sector layout.

The formal definition of this block’s format is reserved for a future revision of the specification.

4.14. Snapshot Block (`SNAP`)

The snapshot block holds a list of historical indexes, representing earlier versions of the media captured within the image. This feature enables users to manually preserve a specific media state, allowing reversion to previous versions or comparison between multiple data capture attempts.

The active index used by the image must always be the one referenced by the image header. If any snapshot block references the current index, it must be ignored and treated as non-existent during image save operations.

Generation 0 refers to the initial image state, where only a single index—pointed to by the header—is present.

The latest image header should reference all available snapshots, unless individual blocks have been explicitly discarded by the user. Once discarded, such blocks become orphaned and are no longer reachable within the image structure.

During conversion from AaruFormat, only one snapshot (or the latest index) should be included, based on user selection.

4.14.1. Structure Definition

```
/* Undefined */
```

4.14.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The snapshot block identifier, always `SNAP`
uint32_t	4 bytes	length	The length in bytes of the data following this header.
uint16_t	2 bytes	generation	The generation, starting from 1, of this snapshot. Every snapshot gets a generation incremented in one from the lastest recorded one.
int64_t	8 bytes	creationTime	Creation time of this snapshot.
uint64_t	8 bytes	index	Offset in bytes where the index marked by this snapshot resides.

4.15. Parent File Block (PRNT)

The parent file block provides metadata required to locate the image file from which the current image is derived. Its primary purpose is to enable hierarchical composition, where non-written sectors in the current image are transparently resolved by referencing their counterparts in the parent image.

All sectors marked as unwritten must be read from the associated parent image, ensuring data completeness and consistency across derivative images.

4.15.1. Structure Definition

```
/* Undefined */
```

4.15.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The parent block identifier, always PRNT
uint32_t	4 bytes	length	The length in bytes of the data following this header.
GUID	16 bytes	parentId	The unique identifier of the parent.
uint16_t	2 bytes	parentClueLength	The size in bytes of the clue string following this field.
String	N bytes	parentClue	A clue, be it a path, filename, UNC, etc., to find the parent. If not valid or not found implementations shall try the directory where the image resides first and the current working directory if not found there.

This block contains metadata essential for locating the corresponding parent image.

All sectors flagged as undumped in the current image must be retrieved from the parent image to ensure completeness. The parent may also store supplementary blocks—such as media tags or metadata—that are not duplicated in the current image. However, any correctly defined data blocks or deduplication tables present in this image will override those found in the parent.

A clue field assists implementations in locating the parent, while a unique parent ID confirms its validity. If the clue fails to resolve the location, the implementation must first scan the directory containing the current image for files with a matching AaruFormat header and expected ID. If unsuccessful, the fallback should be the current working directory.

If this block is present but the parent image cannot be located, the implementation must terminate the open operation, as reconstructing the complete media content depends on the parent's data.

4.16. Dump Hardware Block (`DMP*`)

This block defines the set of hardware components involved in capturing the media content. It includes an array listing each device used during the dumping process, along with the specific extents each device recorded.

This structure allows implementations to trace data provenance and associate dumped regions with their corresponding hardware sources, ensuring accountability and reproducibility in the dumping workflow.

4.16.1. Structure Definition

```
/**Dump hardware block, contains a list of hardware used to dump the media on this
image */
typedef struct DumpHardwareHeader
{
    /**Identifier, <see cref="BlockType.DumpHardwareBlock" /> */
    uint32_t identifier;
    /**How many entries follow this header */
    uint16_t entries;
    /**Size of the whole block, not including this header, in uint8_ts */
    uint32_t length;
    /**CRC64-ECMA of the block */
    uint64_t crc64;
} DumpHardwareHeader;
```

4.16.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The dump hardware block identifier, always `DMP*`
uint16_t	2 bytes	entries	The number of entries following this header
uint32_t	4 bytes	length	The length in bytes of the data following this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header

4.16.3. Dump hardware entries

```
/**Dump hardware entry, contains length of strings that follow, in the same order as
the length, this structure */
typedef struct DumpHardwareEntry
{
    /**Length of UTF-8 manufacturer string */
    uint32_t manufacturerLength;
    /**Length of UTF-8 model string */
    uint32_t modelLength;
    /**Length of UTF-8 revision string */
    uint32_t revisionLength;
    /**Length of UTF-8 firmware version string */
    uint32_t firmwareLength;
    /**Length of UTF-8 serial string */
    uint32_t serialLength;
    /**Length of UTF-8 software name string */
    uint32_t softwareNameLength;
    /**Length of UTF-8 software version string */
    uint32_t softwareVersionLength;
    /**Length of UTF-8 software operating system string */
    uint32_t softwareOperatingSystemLength;
    /**How many extents are after the strings */
    uint32_t extents;
} DumpHardwareEntry;
```

4.16.4. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	manufacturerLength	Length of UTF-8 manufacturer string.
uint32_t	4 bytes	modelLength	Length of UTF-8 model string.
uint32_t	4 bytes	revisionLength	Length of UTF-8 revision string.
uint32_t	4 bytes	firmwareLength	Length of UTF-8 firmware version string.
uint32_t	4 bytes	serialLength	Length of UTF-8 serial number string.
uint32_t	4 bytes	softwareNameLength	Length of UTF-8 software name string.
uint32_t	4 bytes	softwareVersionLength	Length of UTF-8 software version string.
uint32_t	4 bytes	softwareOperatingSystemLength	Length of UTF-8 software operating system string.
uint32_t	4 bytes	extents	How many extents are after the strings.

4.16.5. Extents

```
/**Dump hardware extent, contains the start and end of the extent in the media */
typedef struct DumpHardwareExtent
{
    /**Start of the extent in the media */
    uint64_t start;
    /**End of the extent in the media */
    uint64_t end;
} DumpHardwareExtent;
```

4.16.6. Field Descriptions

Type	Size	Name	Description
uint64_t	8 bytes	start	Starting LBA of the extent (inclusive).
uint64_t	8 bytes	end	Ending LBA of the extent (inclusive).

Each dump hardware entry is followed by a sequence of string fields in the following fixed order:

1. Manufacturer
2. Model
3. Revision
4. Firmware Version
5. Serial Number
6. Software Name
7. Software Version
8. Software Operating System

Immediately after the final string (`Software Operating System`), the list of extents associated with that hardware entry begins.

4.17. Tape File Block (TFLE)

Lists all tape files. Tape files are separations written to media, usually digital tapes, and are marked by filemarks.

4.17.1. Structure Definition

```
#define TAPE_FILE_MAGIC 0x454C4654
/* TODO */
```

4.17.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The tape file block identifier, always TFLE
uint16_t	2 bytes	entries	The number of entries following this header
uint32_t	4 bytes	length	The length in bytes of the data following this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header

4.17.3. Tape file entries

```
/* TODO */
```

4.17.4. Field Descriptions

Type	Size	Name	Description
uint32	4 bytes	file	File number.
uint8	1 byte	partition	Partition number this file belongs to.
uint64	8 bytes	firstBlock	First block number, inclusive, of the file.
uint64	8 bytes	lastBlock	Last block number, inclusive, of the file.

4.18. Tape Partition Block (TPBT)

This block lists all tape partitions. Tape partitions are separations written to media. They are used to distinguish two sets of related data that are distant enough to warrant separation but still belong on the same tape. A well-known example is the LTFS filesystem.

4.18.1. Structure Definition

```
#define TAPE_PARTITION_MAGIC 0x54504254
/* TODO */
```

4.18.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The tape partition block identifier, always TPBT
uint16_t	2 bytes	entries	The number of entries following this header
uint32_t	4 bytes	length	The length in bytes of the data following this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header

4.18.3. Tape partition entries

```
/* TODO */
```

4.18.4. Field Descriptions

Type	Size	Name	Description
uint8_t	1 byte	number	Partition number.
uint64_t	8 bytes	firstBlock	First block number, inclusive, of the partition.
uint64_t	8 bytes	lastBlock	Last block number, inclusive, of the partition.

4.19. Compact Disc Indexes Block (`CDIX`)

On CompactDisc and related media, tracks can contain multiple indexes. These are used to mark separations in the data, such as distinct segments of a musical performance.

The table of contents always references index 1. All other indexes—including index 0 (the pregap)—are stored in the subchannel information.

This block holds a list of all known indexes for quick lookup.

4.19.1. Structure Definition

```
#define CD_INDEXES_MAGIC 0x58444943
/* TODO */
```

4.19.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The compact disc indexes block identifier, always `CDIX`
uint16_t	2 bytes	entries	The number of entries following this header
uint32_t	4 bytes	length	The length in bytes of the data following this header.
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header

4.19.3. Index entries

```
/* TODO */
```

4.19.4. Field Descriptions

Type	Size	Name	Description
uint16_t	2 bytes	track	Track this index belongs to.
uint16_t	2 bytes	index	Index number.
int32_t	4 bytes	lba	LBA where this index starts.

4.20. Flux Data Block (`FLUX`)

This block lists all known flux captures. Certain hardware devices, such as Kryoflux, Pauline, and Applesauce, read magnetic media at the flux transition level.

Flux transition reads are digital representations of the analog properties of the media, and cannot be reliably interpreted on a sector-by-sector basis without further processing. Instead, the data is accessed through capture blocks whose size varies based on the medium and imaging hardware. For example, floppy disk captures typically represent one full track revolution; Applesauce may capture 1¼ revolutions. For Quick Disks, the minimum capture is often an entire side of the media.

Each capture block includes two flux data streams: one for user data and one for the indexing signal.

Flux data is represented as an array of `uint8_t` bytes. Each byte stores the tick count since the last flux transition. If no transition is detected within a byte's range, the value `0xFF` is used, and counting resumes in the next byte with ticks accumulated.

Flux data is stored in `DataBlocks` of the flux data type, referenced from a deduplication table of the same type. Only one flux-type deduplication table is allowed per image, and it must have exactly one level.

4.20.1. Structure Definition

```
/* Undefined */
```

4.20.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The flux data block identifier, always `FLUX`
uint16_t	2 bytes	entries	The number of entries following this header
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header

4.20.3. Flux entries

```
/* Undefined */
```

4.20.4. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	head	Head the data corresponds to.
uint16_t	2 bytes	track	Track the data corresponds to.
uint8_t	1 byte	subtrack	Substep of a track that the data corresponds to.

Type	Size	Name	Description
uint64_t	8 bytes	resolution	Number of picoseconds at which the sampling was performed.
uint64_t	8 bytes	tableEntry	Entry number in the deduplication table where the data corresponding to this flux entry is stored

4.21. Bitstream Data Block (`BITS`)

The `BITS` block contains a list of all known bitstream captures.

A **bitstream** is derived by interpreting flux transitions using an encoding scheme timing table. While bitstreams sit below sector-level data in the hierarchy, they are still a higher abstraction than raw flux transitions.

Storing bitstream data is valuable because multiple dumps from the same media often produce inconsistent and incomparable flux transitions. However, once decoded into bitstreams—regardless of whether sector-level user data can be extracted—the results remain consistent and comparable.

Bitstream-level representations are also preferred in low-level emulation scenarios. Emulators, such as floppy drive emulators, can reconstruct original media more effectively using bitstream data than flux data.

Bitstream data is stored in `DataBlocks` with the `bitstream` data type. Each image must contain exactly one deduplication table of this data type, and that table must have a single level.

NOTE Bitstream deduplication tables provide a reference for associating bitstream captures and their corresponding data blocks.

4.21.1. Structure Definition

```
/* Undefined */
```

4.21.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The bitstream data block identifier, always `BITS`
uint16_t	2 bytes	entries	The number of entries following this header
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header

4.21.3. Bitstream entries

```
/* Undefined */
```

4.21.4. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	head	Head the data corresponds to.
uint16_t	2 bytes	track	Track the data corresponds to.

Type	Size	Name	Description
uint8_t	1 byte	subtrack	Substep of a track that the data corresponds to.
uint64_t	8 bytes	tableEntry	Entry number in the deduplication table where the data corresponding to this bitstream entry is stored

To better understand the relationship between user data, bitstream data and flux data please refer to *Annex F*.

4.22. Track Layout Block (TKLY)

The TKLY block defines the mapping between physical tracks and logical sectors, as referenced by the deduplication table.

Magnetic media such as floppies and hard disks may exhibit complex physical layouts that do not cleanly translate to logical block addresses. This block enables accurate sector location resolution by maintaining explicit layout information.

Each TKLY block corresponds to a unique combination of (sub)track and head, and is followed by a series of sector mapping entries. If known, sectors should be listed in physical order to preserve potential interleaving. Sector numbers may be duplicated.

NOTE This block must not be used for optical or other logically addressable block-based media.

If a referenced LBA is marked as undumped and a FLUX block is present, it indicates the corresponding sector could not be decoded (e.g., damaged or unreadable), and should be considered undumped unless flags state otherwise.

If a FLUX block exists for a given (sub)track but no corresponding TKLY block is present, the entire (sub)track is considered not decoded.

4.22.1. Structure Definition

```
/* Undefined */
```

4.22.2. Field Descriptions

Type	Size	Name	Description
uint32_t	4 bytes	identifier	The track layout block identifier, always TKLY
uint64_t	8 bytes	crc64	The CRC64-ECMA checksum of the data following this header
uint32_t	4 bytes	head	Head the block corresponds to
uint16_t	2 bytes	track	Track the block corresponds to
uint8_t	1 byte	subtrack	Substep of a track the data corresponds to
uint16_t	2 bytes	sectors	Number of sectors in this (sub)track, and therefore, number of entries following this header
uint64_t	8 bytes	flux	Pointer to the flux data block that contains the flux information for this (sub)track
uint64_t	8 bytes	bitstream	Pointer to the bitstream data block that contains the flux information for this (sub)track

4.22.3. Sector Mapping Entries

```
/* Undefined */
```

4.22.4. Field Descriptions

Type	Size	Name	Description
uint16_t	2 bytes	sector	Sector number as present in the appropriate media sector header or equivalent
uint64_t	8 bytes	block	Position in the deduplication table this sector and its flags is stored

Appendix A: Media Types

This annex provides a reference list of known media types at the time this specification was written.

NOTE This list is not exhaustive. The most accurate and current list originates from the `libaaruformat` source.

Content to be defined.

Appendix B: Data Types

This appendix enumerates all known data types that may appear within a data block or be referenced by a deduplication table. These types represent user data, media metadata, or sector-level tags.

NOTE This table is not exhaustive. The most current and authoritative list is always maintained in the `libaaruformat` source.

Value	Data Type
0	No data
1	User data
2	CompactDisc partial Table of Contents
3	CompactDisc session information
4	CompactDisc Table of Contents
5	CompactDisc Power Management Area
6	CompactDisc Absolute Time in Pregroove
7	CompactDisc Lead-in’s CD-Text
8	DVD Physical Format Information
9	DVD Lead-in’s Copyright Management Information
10	DVD Disc Key
11	DVD Burst Cutting Area
12	DVD DMI
13	DVD Media Identifier
14	DVD Media Key Block
15	DVD-RAM Disc Definition Structure
16	DVD-RAM Medium Status
17	DVD-RAM Spare Area Information
18	DVD-R RMD
19	DVD-R Pre-recorded Information
20	DVD-R Media Identifier
21	DVD-R Physical Format Information
22	DVD ADress In Pregroove
23	HD DVD Copy Protection Information
24	HD DVD Medium Status

Value	Data Type
25	DVD DL Layer Capacity
26	DVD DL Middle Zone Address
27	DVD DL Jump Interval Size
28	DVD DL Manual Layer Jump LBA
29	Blu-ray Disc Information
30	Blu-ray Burst Cutting Area
31	Blu-ray Disc Definition Structure
32	Blu-ray Cartridge Status
33	Blu-ray Spare Area Information
34	AACS Volume Identifier
35	AACS Serial Number
36	AACS Media Identifier
37	AACS Media Key Block
38	AACS Data Keys
39	AACS LBA Extents
40	CPRM Media Key Block
41	Hybrid disc recognized layers
42	MMC Write Protection
43	MMC Disc Information
44	MMC Track Resources Information
45	MMC Pseudo-OverWrite Resources Information
46	SCSI INQUIRY response
47	SCSI MODE PAGE 2Ah
48	ATA IDENTIFY response
49	ATAPI IDENTIFY response
50	PCMCIA CIS
51	SecureDigital CID
52	SecureDigital CSD
53	SecureDigital SCR
54	SecureDigital OCR
55	MultiMediaCard CID

Value	Data Type
56	MultiMediaCard CSD
57	MultiMediaCard OCR
58	MultiMediaCard Extended CSD
59	Xbox Security Sector
60	~~Floppy Lead-out~~ **DEPRECATED**
61	DVD Disc Control Block
62	~~CompactDisc First track negative pregap~~ **DEPRECATED**
63	~~CompactDisc Lead-out~~ **DEPRECATED**
64	SCSI MODE SENSE(6) response
65	SCSI MODE SENSE(10) response
66	USB descriptors
67	Xbox Disc Manufacturer Information
68	Xbox Physical Format Information
69	CompactDisc sector prefix (sync, header)
70	CompactDisc sector suffix (EDC, ECC P, ECC Q)
71	CompactDisc subchannel
72	Apple Profile tag (20 bytes)
73	Apple Sony tag (12 bytes)
74	Priam Data Tower tag (24 bytes)
75	CompactDisc Media Catalogue Number
76	CompactDisc sector prefix (only incorrect ones stored)
77	CompactDisc sector suffix (only incorrect ones stored)
78	CompactDisc MODE 2 sector subheader
79	~~CompactDisc Lead-in~~ **DEPRECATED**
80	DVD Disc Key (decrypted)
81	DVD CPI_MAI
82	DVD Title Key (decrypted)
83	Flux data
84	Bitstream data

Appendix C: Compression Types

This apprendix lists all supported compression algorithms used within AaruFormat images.

NOTE Compression method definitions may evolve over time. For the latest and most accurate listing, refer to the `libaaruformat` source.

Value	Algorithm
0	None
1	LZMA — stream prepended by 5 bytes of parameters
2	FLAC
3	LZMA after Claunia Subchannel Transform (see Appendix D) — stream prepended by 5 bytes of parameters

Appendix D: Claunia Subchannel Transform

The subchannel structure in CompactDisc media—and compatible formats—consists of eight interleaved components: `P`, `Q`, `R`, `S`, `T`, `U`, `V`, `W`.

In their raw form, each byte read from the disc contains a single bit from each of these elements, resulting in a highly interleaved data stream. This structure, while efficient for playback, poses challenges for compression algorithms such as LZMA, which struggle with apparent randomness and achieve poor compression ratios (typically less than 2%).

To address this, the **Claunia Subchannel Transform** is applied:

- All bits are **de-interleaved** so that each subchannel (`P` through `W`) is formed into distinct byte streams.
- All `P` bytes from all sectors are written sequentially, followed by all `Q` bytes, then `R`, and so on up to `W`.

While this transform temporarily increases memory usage (approximately 32MiB additional), the benefits are substantial:

- Compression speed improves up to **10× faster**
- Compression gains reach approximately **96%**, particularly on media lacking `R`–`W` subchannel data—as is the case with ~99% of discs.

NOTE For implementation specifics or updates to this method, refer to the authoritative `libaaruformat` source.

Appendix E: Annex E: Deprecated Media Types

NOTE These values must not be used in new images. They may still appear in legacy images. For the most accurate and up-to-date list, refer to the `libaaruformat` source.

Enum	Value	Summary
Unknown	0	Unknown disk type
Unknown MO	1	Unknown magneto-optical
GENERIC_HDD	2	Generic hard disk
Microdrive	3	Microdrive type hard disk
Zone_HDD	4	Zoned hard disk
FlashDrive	5	USB flash drives
Unknown Tape	6	Unknown data tape
CD	10	Any unknown or standard violating CD
CDDA	11	CD Digital Audio (Red Book)
CDG	12	CD+G (Red Book)
CDEG	13	CD+EG (Red Book)
CDI	14	CD-i (Green Book)
CDROM	15	CD-ROM (Yellow Book)
CDROMXA	16	CD-ROM XA (Yellow Book)
CDPLUS	17	CD+ (Blue Book)
CDMO	18	CD-MO (Orange Book)
CDR	19	CD-Recordable (Orange Book)
CDRW	20	CD-ReWritable (Orange Book)
CDMRW	21	Mount-Rainier CD-RW
VCD	22	Video CD (White Book)
SVCD	23	Super Video CD (White Book)
PCD	24	Photo CD (Beige Book)

Enum	Value	Summary
SACD	25	Super Audio CD (Scarlet Book)
DDCD	26	Double-Density CD-ROM (Purple Book)
DDCDR	27	DD CD-R (Purple Book)
DDCDRW	28	DD CD-RW (Purple Book)
DTSCD	29	DTS audio CD (non-standard)
CDMIDI	30	CD-MIDI (Red Book)
CDV	31	CD-Video (ISO/IEC 61104)
PD650	32	120mm, Phase-Change, 1298496 sectors, 512 bytes/sector, PD650, ECMA-240, ISO 15485
PD650_WORM	33	120mm, Write-Once, 1281856 sectors, 512 bytes/sector, PD650, ECMA-240, ISO 15485
CDIREADY	34	CD-i Ready, contains a track before the first TOC track, in mode 2, and all TOC tracks are Audio. Subchannel marks track as audio pause.
FMTOWNS	35	
DVDROM	40	DVD-ROM (applies to DVD Video and DVD Audio)
DVDR	41	DVD-R
DVDRW	42	DVD-RW
DVDPR	43	DVD+R
DVDPRW	44	DVD+RW
DVDPRWDL	45	DVD+RW DL
DVDRDL	46	DVD-R DL
DVDPRDL	47	DVD+R DL
DVDRAM	48	DVD-RAM
DVDRWDL	49	DVD-RW DL
DVDDownload	50	DVD-Download
HDDVDROM	51	HD DVD-ROM (applies to HD DVD Video)
HDDVDRAM	52	HD DVD-RAM
HDDVDR	53	HD DVD-R
HDDVDRW	54	HD DVD-RW

Enum	Value	Summary
HDDVDRDL	55	HD DVD-R DL
HDDVDRWDL	56	HD DVD-RW DL
BDROM	60	BD-ROM (and BD Video)
BDR	61	BD-R
BDRE	62	BD-RE
BDRXL	63	BD-R XL
BDREXL	64	BD-RE XL
UHDBD	65	Ultra HD Blu-ray
EVD	70	Enhanced Versatile Disc
FVD	71	Forward Versatile Disc
HVD	72	Holographic Versatile Disc
CBHD	73	China Blue High Definition
HDVMD	74	High Definition Versatile Multilayer Disc
VCDHD	75	Versatile Compact Disc High Density
SVOD	76	Stacked Volumetric Optical Disc
FDDVD	77	Five Dimensional disc
CVD	78	China Video Disc
LD	80	Pioneer LaserDisc
LDROM	81	Pioneer LaserDisc data
LDROM2	82	
LVROM	83	
MegaLD	84	
CRVdisc	85	Writable LaserDisc with support for component video
HiMD	90	Sony Hi-MD
MD	91	Sony MiniDisc
MDData	92	Sony MD-Data
MDData2	93	Sony MD-Data2
MD60	94	Sony MiniDisc, 60 minutes, formatted with Hi-MD format
MD74	95	Sony MiniDisc, 74 minutes, formatted with Hi-MD format
MD80	96	Sony MiniDisc, 80 minutes, formatted with Hi-MD format

Enum	Value	Summary
UDO	100	5.25", Phase-Change, 1834348 sectors, 8192 bytes/sector, Ultra Density Optical, ECMA-350, ISO 17345
UDO2	101	5.25", Phase-Change, 3669724 sectors, 8192 bytes/sector, Ultra Density Optical 2, ECMA-380, ISO 11976
UDO2_WORM	102	5.25", Write-Once, 3668759 sectors, 8192 bytes/sector, Ultra Density Optical 2, ECMA-380, ISO 11976
PlayStationMemoryCard	110	
PlayStationMemoryCard2	111	
PS1CD	112	Sony PlayStation game CD
PS2CD	113	Sony PlayStation 2 game CD
PS2DVD	114	Sony PlayStation 2 game DVD
PS3DVD	115	Sony PlayStation 3 game DVD
PS3BD	116	Sony PlayStation 3 game Blu-ray
PS4BD	117	Sony PlayStation 4 game Blu-ray
UMD	118	Sony PlayStation Portable Universal Media Disc (ECMA-365)
PlayStationVitaGameCard	119	
PS5BD	120	Sony PlayStation 5 game Ultra HD Blu-ray
XGD	130	Microsoft X-box Game Disc
XGD2	131	Microsoft X-box 360 Game Disc
XGD3	132	Microsoft X-box 360 Game Disc
XGD4	133	Microsoft X-box One Game Disc
MEGACD	150	Sega MegaCD
SATURNCD	151	Sega Saturn disc
GDROM	152	Sega/Yamaha Gigabyte Disc
GDR	153	Sega/Yamaha recordable Gigabyte Disc
SegaCard	154	
MilCD	155	

Enum	Value	Summary
MegaDriveCartridge	156	
_32XCartridge	157	
SegaPicoCartridge	158	
MasterSystemCartridge	159	
GameGearCartridge	160	
SegaSaturnCartridge	161	
HuCard	170	PC-Engine / TurboGrafx cartridge
SuperCDROM2	171	PC-Engine / TurboGrafx CD
JaguarCD	172	Atari Jaguar CD
ThreeDO	173	3DO CD
PCFX	174	NEC PC-FX
NeoGeoCD	175	NEO-GEO CD
CDTV	176	Commodore CDTV
CD32	177	Amiga CD32
Nuon	178	Nuon (DVD based videogame console)
Playdia	179	Bandai Playdia
Apple32SS	180	5.25", SS, DD, 35 tracks, 13 spt, 256 bytes/sector, GCR
Apple32DS	181	5.25", DS, DD, 35 tracks, 13 spt, 256 bytes/sector, GCR
Apple33SS	182	5.25", SS, DD, 35 tracks, 16 spt, 256 bytes/sector, GCR
Apple33DS	183	5.25", DS, DD, 35 tracks, 16 spt, 256 bytes/sector, GCR
AppleSonySS	184	3.5", SS, DD, 80 tracks, 8 to 12 spt, 512 bytes/sector, GCR
AppleSonyDS	185	3.5", DS, DD, 80 tracks, 8 to 12 spt, 512 bytes/sector, GCR
AppleFileWare	186	5.25", DS, ?D, ?? tracks, ?? spt, 512 bytes/sector, GCR, opposite side heads, aka Twiggy

Enum	Value	Summary
DOS_525_SS_DD_8	190	5.25", SS, DD, 40 tracks, 8 spt, 512 bytes/sector, MFM
DOS_525_SS_DD_9	191	5.25", SS, DD, 40 tracks, 9 spt, 512 bytes/sector, MFM
DOS_525_DS_DD_8	192	5.25", DS, DD, 40 tracks, 8 spt, 512 bytes/sector, MFM
DOS_525_DS_DD_9	193	5.25", DS, DD, 40 tracks, 9 spt, 512 bytes/sector, MFM
DOS_525_HD	194	5.25", DS, HD, 80 tracks, 15 spt, 512 bytes/sector, MFM
DOS_35_SS_DD_8	195	3.5", SS, DD, 80 tracks, 8 spt, 512 bytes/sector, MFM
DOS_35_SS_DD_9	196	3.5", SS, DD, 80 tracks, 9 spt, 512 bytes/sector, MFM
DOS_35_DS_DD_8	197	3.5", DS, DD, 80 tracks, 8 spt, 512 bytes/sector, MFM
DOS_35_DS_DD_9	198	3.5", DS, DD, 80 tracks, 9 spt, 512 bytes/sector, MFM
DOS_35_HD	199	3.5", DS, HD, 80 tracks, 18 spt, 512 bytes/sector, MFM
DOS_35_ED	200	3.5", DS, ED, 80 tracks, 36 spt, 512 bytes/sector, MFM
DMF	201	3.5", DS, HD, 80 tracks, 21 spt, 512 bytes/sector, MFM
DMF_82	202	3.5", DS, HD, 82 tracks, 21 spt, 512 bytes/sector, MFM
XDF_525	203	5.25", DS, HD, 80 tracks, ? spt, ??? + ??? + ??? bytes/sector, MFM track 0 = ??15 sectors, 512 bytes/sector, falsified to DOS as 19 spt, 512 bps
XDF_35	204	3.5", DS, HD, 80 tracks, 4 spt, 8192 + 2048 + 1024 + 512 bytes/sector, MFM track 0 = 19 sectors, 512 bytes/sector, falsified to DOS as 23 spt, 512 bps
IBM23FD	210	8", SS, SD, 32 tracks, 8 spt, 319 bytes/sector, FM
IBM33FD_128	211	8", SS, SD, 73 tracks, 26 spt, 128 bytes/sector, FM
IBM33FD_256	212	8", SS, SD, 74 tracks, 15 spt, 256 bytes/sector, FM, track 0 = 26 sectors, 128 bytes/sector
IBM33FD_512	213	8", SS, SD, 74 tracks, 8 spt, 512 bytes/sector, FM, track 0 = 26 sectors, 128 bytes/sector
IBM43FD_128	214	8", DS, SD, 74 tracks, 26 spt, 128 bytes/sector, FM, track 0 = 26 sectors, 128 bytes/sector

Enum	Value	Summary
IBM43FD_256	215	8", DS, SD, 74 tracks, 26 spt, 256 bytes/sector, FM, track 0 = 26 sectors, 128 bytes/sector
IBM53FD_256	216	8", DS, DD, 74 tracks, 26 spt, 256 bytes/sector, MFM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
IBM53FD_512	217	8", DS, DD, 74 tracks, 15 spt, 512 bytes/sector, MFM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
IBM53FD_1024	218	8", DS, DD, 74 tracks, 8 spt, 1024 bytes/sector, MFM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
RX01	220	8", SS, DD, 77 tracks, 26 spt, 128 bytes/sector, FM
RX02	221	8", SS, DD, 77 tracks, 26 spt, 256 bytes/sector, FM/MFM
RX03	222	8", DS, DD, 77 tracks, 26 spt, 256 bytes/sector, FM/MFM
RX50	223	5.25", SS, DD, 80 tracks, 10 spt, 512 bytes/sector, MFM
ACORN_525_SS_SD_40	230	5.25", SS, SD, 40 tracks, 10 spt, 256 bytes/sector, FM
ACORN_525_SS_SD_80	231	5.25", SS, SD, 80 tracks, 10 spt, 256 bytes/sector, FM
ACORN_525_SS_DD_40	232	5.25", SS, DD, 40 tracks, 16 spt, 256 bytes/sector, MFM
ACORN_525_SS_DD_80	233	5.25", SS, DD, 80 tracks, 16 spt, 256 bytes/sector, MFM
ACORN_525_DS_DD	234	5.25", DS, DD, 80 tracks, 16 spt, 256 bytes/sector, MFM
ACORN_35_DS_DD	235	3.5", DS, DD, 80 tracks, 5 spt, 1024 bytes/sector, MFM
ACORN_35_DS_HD	236	3.5", DS, HD, 80 tracks, 10 spt, 1024 bytes/sector, MFM
ATARI_525_SD	240	5.25", SS, SD, 40 tracks, 18 spt, 128 bytes/sector, FM
ATARI_525_ED	241	5.25", SS, ED, 40 tracks, 26 spt, 128 bytes/sector, MFM
ATARI_525_DD	242	5.25", SS, DD, 40 tracks, 18 spt, 256 bytes/sector, MFM
ATARI_35_SS_DD	243	3.5", SS, DD, 80 tracks, 10 spt, 512 bytes/sector, MFM

Enum	Value	Summary
ATARI_35_DS_DD	244	3.5", DS, DD, 80 tracks, 10 spt, 512 bytes/sector, MFM
ATARI_35_SS_DD_11	245	3.5", SS, DD, 80 tracks, 11 spt, 512 bytes/sector, MFM
ATARI_35_DS_DD_11	246	3.5", DS, DD, 80 tracks, 11 spt, 512 bytes/sector, MFM
CBM_35_DD	250	3.5", DS, DD, 80 tracks, 10 spt, 512 bytes/sector, MFM (1581)
CBM_AMIGA_35_DD	251	3.5", DS, DD, 80 tracks, 11 spt, 512 bytes/sector, MFM (Amiga)
CBM_AMIGA_35_HD	252	3.5", DS, HD, 80 tracks, 22 spt, 512 bytes/sector, MFM (Amiga)
CBM_1540	253	5.25", SS, DD, 35 tracks, GCR
CBM_1540_Ext	254	5.25", SS, DD, 40 tracks, GCR
CBM_1571	255	5.25", DS, DD, 35 tracks, GCR
NEC_8_SD	260	8", DS, SD, 77 tracks, 26 spt, 128 bytes/sector, FM
NEC_8_DD	261	8", DS, DD, 77 tracks, 26 spt, 256 bytes/sector, MFM
NEC_525_SS	262	5.25", SS, SD, 80 tracks, 16 spt, 256 bytes/sector, FM
NEC_525_DS	263	5.25", DS, SD, 80 tracks, 16 spt, 256 bytes/sector, MFM
NEC_525_HD	264	5.25", DS, HD, 77 tracks, 8 spt, 1024 bytes/sector, MFM
NEC_35_HD_8	265	3.5", DS, HD, 77 tracks, 8 spt, 1024 bytes/sector, MFM, aka mode 3
NEC_35_HD_15	266	3.5", DS, HD, 80 tracks, 15 spt, 512 bytes/sector, MFM
NEC_35_TD	267	3.5", DS, TD, 240 tracks, 38 spt, 512 bytes/sector, MFM
SHARP_525	264	5.25", DS, HD, 77 tracks, 8 spt, 1024 bytes/sector, MFM
SHARP_525_9	268	3.5", DS, HD, 80 tracks, 9 spt, 1024 bytes/sector, MFM
SHARP_35	265	3.5", DS, HD, 77 tracks, 8 spt, 1024 bytes/sector, MFM
SHARP_35_9	269	3.5", DS, HD, 80 tracks, 9 spt, 1024 bytes/sector, MFM

Enum	Value	Summary
ECMA_99_8	270	5.25", DS, DD, 80 tracks, 8 spt, 1024 bytes/sector, MFM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
ECMA_99_15	271	5.25", DS, DD, 77 tracks, 15 spt, 512 bytes/sector, MFM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
ECMA_99_26	272	5.25", DS, DD, 77 tracks, 26 spt, 256 bytes/sector, MFM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
ECMA_100	198	3.5", DS, DD, 80 tracks, 9 spt, 512 bytes/sector, MFM
ECMA_125	199	3.5", DS, HD, 80 tracks, 18 spt, 512 bytes/sector, MFM
ECMA_147	200	3.5", DS, ED, 80 tracks, 36 spt, 512 bytes/sector, MFM
ECMA_54	273	8", SS, SD, 77 tracks, 26 spt, 128 bytes/sector, FM
ECMA_59	274	8", DS, SD, 77 tracks, 26 spt, 128 bytes/sector, FM
ECMA_66	275	5.25", SS, DD, 35 tracks, 9 spt, 256 bytes/sector, FM, track 0 side 0 = 16 sectors, 128 bytes/sector
ECMA_69_8	276	8", DS, DD, 77 tracks, 8 spt, 1024 bytes/sector, FM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
ECMA_69_15	277	8", DS, DD, 77 tracks, 15 spt, 512 bytes/sector, FM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
ECMA_69_26	278	8", DS, DD, 77 tracks, 26 spt, 256 bytes/sector, FM, track 0 side 0 = 26 sectors, 128 bytes/sector, track 0 side 1 = 26 sectors, 256 bytes/sector
ECMA_70	279	5.25", DS, DD, 40 tracks, 16 spt, 256 bytes/sector, FM, track 0 side 0 = 16 sectors, 128 bytes/sector, track 0 side 1 = 16 sectors, 256 bytes/sector
ECMA_78	280	5.25", DS, DD, 80 tracks, 16 spt, 256 bytes/sector, FM, track 0 side 0 = 16 sectors, 128 bytes/sector, track 0 side 1 = 16 sectors, 256 bytes/sector
ECMA_78_2	281	5.25", DS, DD, 80 tracks, 9 spt, 512 bytes/sector, FM
FDFORMAT_525_DD	290	5.25", DS, DD, 82 tracks, 10 spt, 512 bytes/sector, MFM
FDFORMAT_525_HD	291	5.25", DS, HD, 82 tracks, 17 spt, 512 bytes/sector, MFM
FDFORMAT_35_DD	292	3.5", DS, DD, 82 tracks, 10 spt, 512 bytes/sector, MFM
FDFORMAT_35_HD	293	3.5", DS, HD, 82 tracks, 21 spt, 512 bytes/sector, MFM
Apricot_35	309	3.5", DS, DD, 70 tracks, 9 spt, 512 bytes/sector, MFM
ADR2120	310	
ADR260	311	

Enum	Value	Summary
ADR30	312	
ADR50	313	
AIT1	320	
AIT1Turbo	321	
AIT2	322	
AIT2Turbo	323	
AIT3	324	
AIT3Ex	325	
AIT3Turbo	326	
AIT4	327	
AIT5	328	
AITETurbo	329	
SAIT1	330	
SAIT2	331	
Bernoulli	340	Obsolete type for 8"x11" Bernoulli Box disk
Bernoulli2	341	Obsolete type for 5¼" Bernoulli Box II disks
Ditto	342	
DittoMax	343	
Jaz	344	
Jaz2	345	
PocketZip	346	
REV120	347	
REV35	348	
REV70	349	
ZIP100	350	
ZIP250	351	
ZIP750	352	
Bernoulli35	353	5¼" Bernoulli Box II disk with 35Mb capacity

Enum	Value	Summary
Bernoulli44	354	5¼" Bernoulli Box II disk with 44Mb capacity
Bernoulli65	355	5¼" Bernoulli Box II disk with 65Mb capacity
Bernoulli90	356	5¼" Bernoulli Box II disk with 90Mb capacity
Bernoulli105	357	5¼" Bernoulli Box II disk with 105Mb capacity
Bernoulli150	358	5¼" Bernoulli Box II disk with 150Mb capacity
Bernoulli230	359	5¼" Bernoulli Box II disk with 230Mb capacity
CompactCassette	360	
Data8	361	
MiniDV	362	
Dcas25	363	D/CAS-25: Digital data on Compact Cassette form factor, special magnetic media, 9-track
Dcas85	364	D/CAS-85: Digital data on Compact Cassette form factor, special magnetic media, 17-track
Dcas103	365	D/CAS-103: Digital data on Compact Cassette form factor, special magnetic media, 21-track
CFast	370	
CompactFlash	371	
CompactFlashType2	372	
DigitalAudioTape	380	
DAT160	381	
DAT320	382	
DAT72	383	
DDS1	384	
DDS2	385	
DDS3	386	
DDS4	387	

Enum	Value	Summary
CompactTapeI	390	
CompactTapeII	391	
DECtapeII	392	
DLTtapeIII	393	
DLTtapeIIIxt	394	
DLTtapeIV	395	
DLTtapeS4	396	
SDLT1	397	
SDLT2	398	
VStapeI	399	
Exatape15m	400	
Exatape22m	401	
Exatape22mAME	402	
Exatape28m	403	
Exatape40m	404	
Exatape45m	405	
Exatape54m	406	
Exatape75m	407	
Exatape76m	408	
Exatape80m	409	
Exatape106m	410	
Exatape160mXL	411	

Enum	Value	Summary
Exatape112m	412	
Exatape125m	413	
Exatape150m	414	
Exatape170m	415	
Exatape225m	416	
ExpressCard34	420	
ExpressCard54	421	
PCCardTypeI	422	
PCCardTypeII	423	
PCCardTypeIII	424	
PCCardTypeIV	425	
EZ135	430	SyQuest 135Mb cartridge for use in EZ135 and EZFlyer drives
EZ230	431	SyQuest EZFlyer 230Mb cartridge for use in EZFlyer drive
Quest	432	SyQuest 4.7Gb for use in Quest drive
SparQ	433	SyQuest SparQ 1Gb cartridge
SQ100	434	SyQuest 5Mb cartridge for SQ306RD drive
SQ200	435	SyQuest 10Mb cartridge for SQ312RD drive
SQ300	436	SyQuest 15Mb cartridge for SQ319RD drive
SQ310	437	SyQuest 105Mb cartridge for SQ3105 and SQ3270 drives
SQ327	438	SyQuest 270Mb cartridge for SQ3270 drive
SQ400	439	SyQuest 44Mb cartridge for SQ555, SQ5110 and SQ5200C/SQ200 drives
SQ800	440	SyQuest 88Mb cartridge for SQ5110 and SQ5200C/SQ200 drives
SQ1500	441	SyQuest 1.5Gb cartridge for SyJet drive
SQ2000	442	SyQuest 200Mb cartridge for use in SQ5200C drive

Enum	Value	Summary
SyJet	443	SyQuest 1.5Gb cartridge for SyJet drive
FamicomGamePak	450	
GameBoyAdvanceGamePak	451	
GameBoyGamePak	452	
GOD	453	Nintendo GameCube Optical Disc
N64DD	454	
N64GamePak	455	
NESGamePak	456	
Nintendo3DSGameCard	457	
NintendoDiskCard	458	
NintendoDSGameCard	459	
NintendoDSiGameCard	460	
SNESGamePak	461	
SNESGamePakUS	462	
WOD	463	Nintendo Wii Optical Disc
WUOD	464	Nintendo Wii U Optical Disc
SwitchGameCard	465	
IBM3470	470	
IBM3480	471	
IBM3490	472	
IBM3490E	473	

Enum	Value	Summary
IBM3592	474	
LTO	480	
LTO2	481	
LTO3	482	
LTO3WORM	483	
LTO4	484	
LTO4WORM	485	
LTO5	486	
LTO5WORM	487	
LTO6	488	
LTO6WORM	489	
LTO7	490	
LTO7WORM	491	
MemoryStick	510	
MemoryStickDuo	511	
MemoryStickMicro	512	
MemoryStickPro	513	
MemoryStickProDuo	514	
microSD	520	
miniSD	521	
SecureDigital	522	
MMC	530	
MMCmicro	531	
RSMMC	532	

Enum	Value	Summary
MMCplus	533	
MMCmobile	534	
MLR1	540	
MLR1SL	541	
MLR3	542	
SLR1	543	
SLR2	544	
SLR3	545	
SLR32	546	
SLR32SL	547	
SLR4	548	
SLR5	549	
SLR5SL	550	
SLR6	551	
SLRtape7	552	
SLRtape7SL	553	
SLRtape24	554	
SLRtape24SL	555	
SLRtape40	556	
SLRtape50	557	
SLRtape60	558	
SLRtape75	559	
SLRtape100	560	
SLRtape140	561	
QIC11	570	
QIC120	571	
QIC1350	572	
QIC150	573	

Enum	Value	Summary
QIC24	574	
QIC3010	575	
QIC3020	576	
QIC3080	577	
QIC3095	578	
QIC320	579	
QIC40	580	
QIC525	581	
QIC80	582	
STK4480	590	
STK4490	591	
STK9490	592	
T9840A	593	
T9840B	594	
T9840C	595	
T9840D	596	
T9940A	597	
T9940B	598	
T10000A	599	
T10000B	600	
T10000C	601	
T10000D	602	
Travan	610	
Travan1Ex	611	
Travan3	612	
Travan3Ex	613	
Travan4	614	
Travan5	615	
Travan7	616	
VXA1	620	

Enum	Value	Summary
VXA2	621	
VXA3	622	
ECMA_153	630	5.25", M.O., WORM, 650Mb, 318750 sectors, 1024 bytes/sector, ECMA-153, ISO 11560
ECMA_153_512	631	5.25", M.O., WORM, 600Mb, 581250 sectors, 512 bytes/sector, ECMA-153, ISO 11560
ECMA_154	632	3.5", M.O., RW, 128Mb, 248826 sectors, 512 bytes/sector, ECMA-154, ISO 10090
ECMA_183_512	633	5.25", M.O., RW/WORM, 1Gb, 904995 sectors, 512 bytes/sector, ECMA-183, ISO 13481
ECMA_183	634	5.25", M.O., RW/WORM, 1Gb, 498526 sectors, 1024 bytes/sector, ECMA-183, ISO 13481
ECMA_184_512	635	5.25", M.O., RW/WORM, 1.2Gb, 1165600 sectors, 512 bytes/sector, ECMA-184, ISO 13549
ECMA_184	636	5.25", M.O., RW/WORM, 1.3Gb, 639200 sectors, 1024 bytes/sector, ECMA-184, ISO 13549
ECMA_189	637	300mm, M.O., WORM, ??? sectors, 1024 bytes/sector, ECMA-189, ISO 13614
ECMA_190	638	300mm, M.O., WORM, ??? sectors, 1024 bytes/sector, ECMA-190, ISO 13403
ECMA_195	639	5.25", M.O., RW/WORM, 936921 or 948770 sectors, 1024 bytes/sector, ECMA-195, ISO 13842
ECMA_195_512	640	5.25", M.O., RW/WORM, 1644581 or 1647371 sectors, 512 bytes/sector, ECMA-195, ISO 13842
ECMA_201	641	3.5", M.O., 446325 sectors, 512 bytes/sector, ECMA-201, ISO 13963
ECMA_201_ROM	642	3.5", M.O., 429975 sectors, 512 bytes/sector, embossed, ISO 13963
ECMA_223	643	3.5", M.O., 371371 sectors, 1024 bytes/sector, ECMA-223
ECMA_223_512	644	3.5", M.O., 694929 sectors, 512 bytes/sector, ECMA-223
ECMA_238	645	5.25", M.O., 1244621 sectors, 1024 bytes/sector, ECMA-238, ISO 15486
ECMA_239	646	3.5", M.O., 310352, 320332 or 321100 sectors, 2048 bytes/sector, ECMA-239, ISO 15498
ECMA_260	647	356mm, M.O., 14476734 sectors, 1024 bytes/sector, ECMA-260, ISO 15898
ECMA_260_Double	648	356mm, M.O., 24445990 sectors, 1024 bytes/sector, ECMA-260, ISO 15898
ECMA_280	649	5.25", M.O., 1128134 sectors, 2048 bytes/sector, ECMA-280, ISO 18093
ECMA_317	650	300mm, M.O., 7355716 sectors, 2048 bytes/sector, ECMA-317, ISO 20162

Enum	Value	Summary
ECMA_322	651	5.25", M.O., 1095840 sectors, 4096 bytes/sector, ECMA-322, ISO 22092, 9.1Gb/cart
ECMA_322_2k	652	5.25", M.O., 2043664 sectors, 2048 bytes/sector, ECMA-322, ISO 22092, 8.6Gb/cart
GigaMo	653	3.5", M.O., 605846 sectors, 2048 bytes/sector, Cherry Book, GigaMo, ECMA-351, ISO 17346
GigaMo2	654	3.5", M.O., 1063146 sectors, 2048 bytes/sector, Cherry Book 2, GigaMo 2, ECMA-353, ISO 22533
ISO_15286	655	5.25", M.O., 1263472 sectors, 2048 bytes/sector, ISO 15286, 5.2Gb/cart
ISO_15286_1024	656	5.25", M.O., 2319786 sectors, 1024 bytes/sector, ISO 15286, 4.8Gb/cart
ISO_15286_512	657	5.25", M.O., ??????? sectors, 512 bytes/sector, ISO 15286, 4.1Gb/cart
ISO_10089	658	5.25", M.O., 314569 sectors, 1024 bytes/sector, ISO 10089, 650Mb/cart
ISO_10089_512	659	5.25", M.O., ?????? sectors, 512 bytes/sector, ISO 10089, 594Mb/cart
CompactFloppy	660	
DemiDiskette	661	
Floptical	662	3.5", 652 tracks, 2 sides, 512 bytes/sector, Floptical, ECMA-207, ISO 14169
HiFD	663	
QuickDisk	664	
UHD144	665	
VideoFloppy	666	
Wafer	667	
ZXMicrodrive	668	
MetaFloppy_Mod_II	669	5.25", SS, DD, 77 tracks, 16 spt, 256 bytes/sector, MFM, 100 tpi, 300rpm
BeeCard	670	
Borsu	671	
DataStore	672	
DIR	673	
DST	674	

Enum	Value	Summary
DTF	675	
DTF2	676	
Flextra3020	677	
Flextra3225	678	
HiTC1	679	
HiTC2	680	
LT1	681	
MiniCard	872	
Orb	683	
Orb5	684	
SmartMedia	685	
xD	686	
XQD	687	
DataPlay	688	
AppleProfile	690	
AppleWidget	691	
AppleHD20	692	
PriamDataTower	693	
Pippin	694	
RA60	700	2382 cylinders, 4 tracks/cylinder, 42 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 204890112 bytes
RA80	701	546 cylinders, 14 tracks/cylinder, 31 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 121325568 bytes
RA81	702	1248 cylinders, 14 tracks/cylinder, 51 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 456228864 bytes
RC25	703	302 cylinders, 4 tracks/cylinder, 42 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 25976832 bytes
RD31	704	615 cylinders, 4 tracks/cylinder, 17 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 21411840 bytes

Enum	Value	Summary
RD32	705	820 cylinders, 6 tracks/cylinder, 17 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 42823680 bytes
RD51	706	306 cylinders, 4 tracks/cylinder, 17 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 10653696 bytes
RD52	707	480 cylinders, 7 tracks/cylinder, 18 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 30965760 bytes
RD53	708	1024 cylinders, 7 tracks/cylinder, 18 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 75497472 bytes
RD54	709	1225 cylinders, 8 tracks/cylinder, 18 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 159936000 bytes
RK06	710	411 cylinders, 3 tracks/cylinder, 22 sectors/track, 256 words/sector, 16 bits/word, 512 bytes/sector, 13888512 bytes
RK06_18	711	411 cylinders, 3 tracks/cylinder, 20 sectors/track, 256 words/sector, 18 bits/word, 576 bytes/sector, 14204160 bytes
RK07	712	815 cylinders, 3 tracks/cylinder, 22 sectors/track, 256 words/sector, 16 bits/word, 512 bytes/sector, 27540480 bytes
RK07_18	713	815 cylinders, 3 tracks/cylinder, 20 sectors/track, 256 words/sector, 18 bits/word, 576 bytes/sector, 28166400 bytes
RM02	714	823 cylinders, 5 tracks/cylinder, 32 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 67420160 bytes
RM03	715	823 cylinders, 5 tracks/cylinder, 32 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 67420160 bytes
RM05	716	823 cylinders, 19 tracks/cylinder, 32 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 256196608 bytes
RP02	717	203 cylinders, 10 tracks/cylinder, 22 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 22865920 bytes
RP02_18	718	203 cylinders, 10 tracks/cylinder, 20 sectors/track, 128 words/sector, 36 bits/word, 576 bytes/sector, 23385600 bytes
RP03	719	400 cylinders, 10 tracks/cylinder, 22 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 45056000 bytes
RP03_18	720	400 cylinders, 10 tracks/cylinder, 20 sectors/track, 128 words/sector, 36 bits/word, 576 bytes/sector, 46080000 bytes
RP04	721	411 cylinders, 19 tracks/cylinder, 22 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 87960576 bytes
RP04_18	722	411 cylinders, 19 tracks/cylinder, 20 sectors/track, 128 words/sector, 36 bits/word, 576 bytes/sector, 89959680 bytes
RP05	723	411 cylinders, 19 tracks/cylinder, 22 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 87960576 bytes

Enum	Value	Summary
RP05_18	724	411 cylinders, 19 tracks/cylinder, 20 sectors/track, 128 words/sector, 36 bits/word, 576 bytes/sector, 89959680 bytes
RP06	725	815 cylinders, 19 tracks/cylinder, 22 sectors/track, 128 words/sector, 32 bits/word, 512 bytes/sector, 174423040 bytes
RP06_18	726	815 cylinders, 19 tracks/cylinder, 20 sectors/track, 128 words/sector, 36 bits/word, 576 bytes/sector, 178387200 bytes
LS120	730	
LS240	731	
FD32MB	732	
RDX	733	
RDX320	734	Imation 320Gb RDX
VideoNow	740	
VideoNowColor	741	
VideoNowXp	742	
Bernoulli10	750	8"x11" Bernoulli Box disk with 10Mb capacity
Bernoulli20	751	8"x11" Bernoulli Box disk with 20Mb capacity
BernoulliBox2_20	752	5▯" Bernoulli Box II disk with 20Mb capacity
KodakVerbatim3	760	
KodakVerbatim6	761	
KodakVerbatim12	762	
ProfessionalDisc	770	Professional Disc for video, single layer, rewritable, 23Gb
ProfessionalDiscDual	771	Professional Disc for video, dual layer, rewritable, 50Gb
ProfessionalDiscTriple	772	Professional Disc for video, triple layer, rewritable, 100Gb

Enum	Value	Summary
ProfessionalDiscQuad	773	Professional Disc for video, quad layer, write once, 128Gb
PDD	774	Professional Disc for DATA, single layer, rewritable, 23Gb
PDD_WORM	775	Professional Disc for DATA, single layer, write once, 23Gb
ArchivalDisc	776	Archival Disc, 1st gen., 300Gb
ArchivalDisc2	777	Archival Disc, 2nd gen., 500Gb
ArchivalDisc3	778	Archival Disc, 3rd gen., 1Tb
ODC300R	779	Optical Disc archive, 1st gen., write once, 300Gb
ODC300RE	780	Optical Disc archive, 1st gen., rewritable, 300Gb
ODC600R	781	Optical Disc archive, 2nd gen., write once, 600Gb
ODC600RE	782	Optical Disc archive, 2nd gen., rewritable, 600Gb
ODC1200RE	783	Optical Disc archive, 3rd gen., rewritable, 1200Gb
ODC1500R	784	Optical Disc archive, 3rd gen., write once, 1500Gb
ODC3300R	785	Optical Disc archive, 4th gen., write once, 3300Gb
ODC5500R	786	Optical Disc archive, 5th gen., write once, 5500Gb
ECMA_322_1k	800	5.25", M.O., 4383356 sectors, 1024 bytes/sector, ECMA-322, ISO 22092, 9.1Gb/cart
ECMA_322_512	801	5.25", M.O., ??????? sectors, 512 bytes/sector, ECMA-322, ISO 22092, 9.1Gb/cart
ISO_14517	802	5.25", M.O., 1273011 sectors, 1024 bytes/sector, ISO 14517, 2.6Gb/cart
ISO_14517_512	803	5.25", M.O., 2244958 sectors, 512 bytes/sector, ISO 14517, 2.3Gb/cart
ISO_15041_512	804	3.5", M.O., 1041500 sectors, 512 bytes/sector, ISO 15041, 540Mb/cart
MetaFloppy_Mod_I	820	5.25", SS, DD, 35 tracks, 16 spt, 256 bytes/sector, MFM, 48 tpi, ???rpm
AtariLynxCard	821	
AtariJaguarCartridge	822	

Appendix F: User Data, Bitstream, Fluxes and Tags

This appendix explains the relationships between user data, bitstream data, flux data, and both sector and media tags in the context of digital imaging and data preservation.

F.1. □□ User Data

User data represents the information a user interacts with—such as a document or file. This data is typically split into discrete units called *sectors*.

- A sector (also known as a block) is the smallest unit a medium can read or write.
- Most media divide user data into independent sectors.

F.2. □□ Sector Tags

A sector may include metadata not visible to the user, but accessible to the operating environment.

Examples:

- Apple Lisa filesystem tags
- CompactDisc subchannel data

These metadata elements are referred to as *sector tags* and are stored alongside user data.

F.3. □ Media Tags

Media tags relate to the storage medium as a whole rather than individual sectors.

Examples:

- CompactDisc Absolute Time In Pregroove (ATIP)
- DVD Disc Manufacturing Information (DMI)

Media tags may or may not be accessible to end users but are often essential for authentication, playback, or archival purposes.

F.4. □ Bitstream Encoding

User data and associated sector tags must be encoded into a binary format before being stored physically. This encoded data is called the *bitstream*.

Common encoding formats include:

- FM (Frequency Modulation): used in early floppies

- MFM (Modified FM): used in most floppy formats
- GCR (Group Code Recording): used by Apple and Commodore
- EFM (Eight-to-Fourteen Modulation): used in CompactDiscs

The bitstream is composed of sequences of 0s and 1s derived from the digital content.

F.5. □□ Flux Data

For physical media (e.g., magnetic or optical), the bitstream must be translated into *flux data*, which captures physical transitions over time.

- A flux transition is a change in magnetic polarity or optical reflectivity.
- Flux data represents the time elapsed since the last transition.
- In FM encoding, for instance:
    - Every 4µs there's a guaranteed transition.
    - A transition at 2µs represents a `1`.
    - Absence of transition at 2µs represents a `0`.

These transitions are what ultimately get recorded onto physical media.

F.6. □ Data Conversion Path

```
User Data → Sector Tags → Bitstream → Flux Data → Physical Media
```

Reverse path (during reading or imaging):

```
Physical Media → Flux Data → Bitstream → User Data + Sector Tags + Media Tags
```

F.7. □ Image Composition

Digital images can contain different combinations of data types:

- Flux data only
- Bitstream data only
- User data only
- Any combination of the above

Each format has specific use cases depending on the accuracy, fidelity, and target preservation needs.