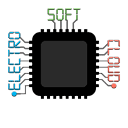
04/09/2021

Daniel Carrasco Marín

ElectroSoftCloud.com

ECM2 file specifications



|  |  |  |
| --- | --- | --- |
| Revision | Date | Modifications |
| R01 | 2021-07 | First versión |
| R02 | 2021-08 | Revised document to add the new features |
| R03 | 2021-08 | New ECM format (v3) |

Content

[Document 5](#_Toc80780969)

[Document Objectives 5](#_Toc80780970)

[History 7](#_Toc80780971)

[How it works 9](#_Toc80780972)

[CDDA Sector 9](#_Toc80780973)

[Mode 1 Sector 9](#_Toc80780974)

[Mode 2 Sector 9](#_Toc80780975)

[Mode 2 XA 1 Sector 9](#_Toc80780976)

[Mode 2 XA 2 Sector 10](#_Toc80780977)

[Why to use ECM2 11](#_Toc80780978)

[Container specifications 13](#_Toc80780979)

[Main Header 13](#_Toc80780980)

[TOC position Header 13](#_Toc80780981)

[Blocks 13](#_Toc80780982)

[Block Header 13](#_Toc80780983)

[Block Data 14](#_Toc80780984)

[Metadata 14](#_Toc80780985)

[ECM Data block 14](#_Toc80780986)

[Data compression 17](#_Toc80780987)

# Document

I know that looks weird to use a privative format like docx on a GNU project. Sorry for that, but I was tired of a bug in LibreOffice that was clearing the data of the document tables all the time.

## Document Objectives

This document defines the ECM2 file format and how the ecm-tools-reloaded program works, and is divided into the following sections:

1. This section which describes the document itself.
2. History
3. How it Works
4. Why to use ECM2
5. Container specification
6. Error Detection Code

# History

The Error Code Modeler format works by removing the unnecessary data from the CD Sectors. This data is mostly the Error Detection Code and the Error Correction Code, that is main reason why the format is called ECM.

The original ECM version removes all the ECM data so is good enough and it helped me to save a lot of space on my disks. The way it works is by detecting the Mode 2 XA sectors and removing the EDC and ECC data from it, and treating the rest of the data as raw bytes (including part of the Mode 2 XA sectors).

I wanted to improve the program by removing even more data, like for example the sync, the address and redundant sub-header data. Also, I wanted to create a seekable file by processing the input file block by block and placing the index in the file header. This allows to know the exact position of every sector in file by just reading the header, which opens the window to the possibility of create a plugin for PCSX to read it directly.

The first program modification works with the above in mind and reduces the file size of the resulting ecm file up to 8%. This version already creates seekable files, and just reading the header you will be able to create an index of the file to be able to seek to the desired sector very fast. Also, I have moved the functions to a class to allow to reuse it into another programs.

The second program modification adds some compression methods to it, allowing to no depend of external tools. Also, unlike most of the external programs, will allow to set the compression method of every stream, allowing to compress the data using LZMA and the CDDA. Also, I wanted to keep the ability to seek into the file. I have modified the headers to add this functionality, and improve the readability by creating fixed size blocks.

# How it works

The ECM format removes data from CD-ROM sectors depending of which kind of sector contains the source, and the following sectors are compatible with this tool:

* CDDA
* MODE1
* MODE2
* MODE2 XA1
* MODE2 XA2

The sector size is CD-ROM is fixed and contains 2352 bytes.

## CDDA Sector

The CDDA sector is entirely composed by data bytes, so nothing can be removed and thus the sector size is not reduced.

|  |
| --- |
| 2352 data bytes |

There also exists a variant of this sector type which is filled by zeros. This sector variant can be easily generated and then can be safely fully removed.

## Mode 1 Sector

The mode 1 sector contains 12 sync bytes, 4 header bytes, 2048 data bytes, 4 EDC bytes, 8 blank bytes and 276 ECC bytes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sync: 12b | Header: 4b | Data: 2048b | EDC: 4b | Blank: 8b | ECC: 276b |

The ECM program will be able to keep only the data and remove the rest of the data, which can be easily generated later to restore the original sector. This will reduce the sector size by about 13%.

There also exists a variant of this sector type which its data is filled by zeros. This sector variant will be fully removed as it can be fully generated without problem.

## Mode 2 Sector

The mode 2 sector is not widely used, but is also processed. This sector contains 12 sync bytes, 4 header bytes and 2336 data bytes.

|  |  |  |
| --- | --- | --- |
| Sync: 12b | Header: 4b | Data: 2336b |

The ECM program will be able to keep only the data part of the sector and remove the rest, which can be easily be generated later to restore the original sector. This will reduce the sector size by about 0.7% (not too much).

There also exists a variant of this sector type which its data is filled by zeros. This sector variant can be easily generated and then can be safely fully removed.

## Mode 2 XA 1 Sector

This is the first Mode 2 Extended sector type and is very similar to Mode 1 sector. The 8 bytes blank sector is moved between the header and the data and is used as sub-header. This sector contains 12 sync bytes, 4 header bytes, 8 sub-header bytes, 2048 data bytes, 4 EDC bytes and 276 ECC bytes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sync: 12b | Header: 4b | Sub-Header: 8b | Data: 2048b | EDC: 4b | ECC: 276b |

In this sector mode the data is required and also the sub-header. As the sub-header is redundant (4 bytes repeated 2 times), only 4 bytes are required. The ECM program will be able to remove the rest of data and then reduce the size about 13%. The removed data also can be generated easily later to restore the original sector.

There also exists a variant of this sector type which its data is filled by zeros. This sector variant can be mostly easily generated and then almost all can be removed. Only 4 sub-header bytes are required.

## Mode 2 XA 2 Sector

This mode removes the ECC bytes and uses its space to store data, so it contains 12 sync bytes, 4 header bytes, 8 sub-header bytes, 2324 data bytes and 4 EDC bytes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sync: 12b | Header: 4b | Sub-Header: 8b | Data: 2324b | EDC: 4b |

Like the XA 1 sector mode, it only requires the data bytes and 4 sub-header bytes, and the rest can be generated. This allows to reduce the size about 1.2%.

There also exists a variant of this sector type which its data is filled by zeros. This sector variant can be mostly easily generated and then almost all can be removed. Only 4 sub-header bytes are required.

# Why to use ECM2

The main reason to use ECM2 is the size reduction, which in some situations is about the 13% of the original file size and can reach higher reductions if GAP sectors are found. For example, The Final Fantasy VII CD1 image has reached a reduction of about the 21% without compression, so it can be implemented into an emulator and then reduce your collection size with a much lower CPU overhead. The format also allows some compression methods which will help to reduce the file size even more.

The removed data can be generated later and recover the original sector data, so this tool will produce a lossless reduction method, which can be complemented with another compression methods like zlib, lzma, lz4 and even flac for CDDA tracks.

Starting in the V3 format you’ll be able to store multiple disks in the same container, and tags to it like title, cover, publisher, year… including a title for every disk. Also, this new format includes the possibility to add external files, so you’ll be able to keep the cue file or even patches on it.

# Container specifications

In the ECM V3 version, I have decided to convert the format to a container. This will allow to implement advanced features like tags, multiple images in the same file and even extra files. In this section we will take a look to the supported specifications of this container.

## Main Header

The main header starts at the beginning of the file and is composed by 4 bytes. Three of them are fixed and the 4th defines the file version.

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 3 bytes | ECM | ECM container header |
| 0x03 | 1 byte | 3 | This byte is the file version. Program should be compatible with this version or will not be able to decode the file. |

These directions are absolute.

## TOC position Header

The next 8 bytes of the file will be used to store the Table of Content position. Will be used to seek to the point where the TOC is stored to seek to it directly without read the entire file. Uint64\_t will be used to allow bigger files:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x04 | 8 bytes |  | Toc Position in uint64\_t |

This direction is absolute.

## Blocks

### Block Header

Blocks of data contains a different kind of data, for example metadata, processed image data, files… Every block will contain a header with the following data:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Block type   1. Deleted block (trimable) 2. Metadata block 3. Table of content block 4. ECM data block 5. File block |
| 0x01 | 1 byte |  | Compression (refer to [compression section](#_Data_compression)) |
| 0x02 | 8 bytes |  | The block size in uint64\_t (compressed size) |
| 0x10 | 8 bytes |  | The real block size in uint64\_t (without compression) |

These directions are relative.

If the block is not compressed, then block size and real block size will be the same. The equivalent struct will be:

struct block {

uint8\_t type;

uint8\_t compression;

uint64\_t block\_size;

uint64\_t real\_block\_size;

};

### Block Data

Block data depends of the kind of block. We will explain the main blocks data here.

#### Metadata

Metadata block contains all the metadata info into sub-blocks. The block will start by the header specified in the Block Header section, and a series of sub-blocks with the following structure:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Metadata type in uint8\_t   1. Cover 2. Title 3. ID 4. Release Date 5. Developer 6. Publisher 7. Genre |
| 0x01 | 4 bytes |  | The size of the block in uint32\_t (limited to 4Gb) |
| 0x09 | Variable |  | The sub-block data |

These directions are relative.

Every sub-block must be unique, and only one metadata block can exist in the file.

#### ECM Data block

The ECM data block will contain the optimized CD-ROM image data. Like all the blocks in the file, it must start with the header specified in the Block Header. This stream is special because can contains different compression methods, so the compression in header will be None (0), and both sizes will be equal.

After the block header, this block will contain the following header:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | 1 Byte with the optimizations used in this file. Useful in decoding process. |
| 0x01 | 1 byte |  | Sector per blocks in compressed data or 0 if this option was not used. |
| 0x02 | 1 byte |  | CRC type used to verify the image:   1. Standard Error Code Detection CRC 2. MD5 3. SHA1 |
| 0x03 | 8 bytes |  | Streams TOC data position (relative to block) |
| 0x11 | 8 bytes |  | Sectors TOC data position (relative to block) |
| 0x19 | 8 bytes |  | ECM data position (relative to block) |
| 0x27 | 1 byte |  | ECM title size in uint8\_t |
| 0x28 | - |  | Title string (max 256 characters) |

These directions are relative and the equivalent struct is the following:

struct ecm\_header {

uint8\_t optimizations;

uint8\_t sectors\_per\_block;

uint64\_t crc\_mode;

uint64\_t streams\_toc\_pos;

uint64\_t sectors\_toc\_pos;

uint64\_t ecm\_data\_pos;

int8\_t title\_length;

std::string title;

};

Streams TOC, Sectors TOC and ECM data positions were added to the header, to allow a fast seek to their position. This will allow to place it anywhere instead to need to place it in header.

##### Streams TOC header

The streams TOC header will contain the compressed and uncompressed sizes of the streams TOC data blocks, which will help to decompress the data if required.

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Compression:   1. None 2. Zlib 3. LZMA 4. LZ4 |
| 0x01 | 4 bytes |  | Streams TOC size (Uncompressed) |
| 0x06 | 4 bytes |  | Streams TOC size (Compressed) |

These directions are relative and the equivalent struct will be:

struct sec\_str\_size {

sector\_tools\_compression compression;

uint32\_t count;

uint32\_t uncompressed\_size;

uint32\_t compressed\_size;

};

This header will be followed by one or more blocks of Streams TOC data, that can be compressed using zlib compression to save space.

##### Streams TOC data

Every streams TOC data block will be composed by the following bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Stream info:   * 0: Stream format → 0 = Audio, 1 = Data. * 1 - 3: Stream compression.   1. None   2. Zlib   3. LZMA   4. LZ4   5. FLAC  1. 4 - 7: Reserved |
| 0x01 | 4 bytes |  | The number of sectors that contains the stream (uint32\_t) |
| 0x05 | 4 bytes |  | The end position in output file (uint32\_t) |

These directions are relative.

The equivalent struct will be:

#pragma pack(push, 1)

struct stream {

uint8\_t type: 1;

uint8\_t compression: 3;

uint32\_t end\_sector;

uint32\_t out\_end\_position;

};

#pragma pack(pop)

The pragma is important to create blocks of 9 bytes and reduce the size. Otherwise, every block will be aligned to 16 bytes.

This header is intended to contains the different streams types in file, with their end position in the file and compression type. This will allow to have different compression types, for example, Zlib for data streams and FLAC for audio streams, and also will help to calculate how to decompress the input stream.

##### Sectors TOC header

This header like the Streams TOC header, will contains the compressed and uncompressed size of the Sectors TOC data blocks:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Compression:   1. None 2. Zlib 3. LZMA 4. LZ4 |
| 0x01 | 4 bytes |  | Sectors TOC size (Uncompressed) |
| 0x06 | 4 bytes |  | Sectors TOC size (Compressed) |

These directions are relative and the equivalent struct will be the same as in [Streams TOC header](#_Streams_TOC_header).

This header will be followed by one or more blocks of Sectors TOC data, that can be compressed using zlib compression to save space.

##### Sectos TOC data

Every Sectors TOC data block will be composed by the following bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Sector info:   * 0 - 3: Sector type.   1. CDDA   2. CDDA GAP   3. Mode 1   4. Mode 1 GAP   5. Mode 2   6. Mode 2 GAP   7. Mode 2 XA1   8. Mode 2 XA1 GAP   9. Mode 2 XA2   10. Mode 2 XA2 GAP  1. 4 – 7: Reserved |
| 0x01 | 4 bytes |  | The number of sectors of that type processed. This number is equivalent to an uint32\_t variable. |

These directions are relative and in this case the equivalent struct will be:

#pragma pack(push, 1)

struct sector {

uint8\_t mode: 4;

uint32\_t sector\_count;

};

#pragma

The pragma is important to create blocks of 5 bytes and reduce the size. Otherwise, every block will be aligned to 8 bytes.

This header is intended to contains the different sectors types and number of sectors of this type in the source file. This info will be used to recover the original sector state. The entire header can be compressed using zlib to save some space.

##### ECM Data

The ECM data is the image data after process it. All the data will be stored continuously without any separator (the Streams and Sectors TOC will be enough to recover the original data). Compressed streams will require to create checkpoints in the file to provide a faster seek time, but these checkpoints will be created by the compression library. The CRC will be stored at the end of the ECM Data block and its size will depend of the kind of CRC used.

## Table of Content

The table of Content will contain an index of the blocks in the file with its format. The position of this Table of Content must be stored in the [file header](#_TOC_position_Header) for a faster seeking. Like the rest of the blocks, this block must start with the [Block Header](#_Block_Header). After the block header, this block will contain a series of sub-blocks with the index info as explained in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Block type |
| 0x01 | 8 bytes |  | Block start position in file in uint64\_t |

These directions are relative.

This Table of Content will allow to easily navigate into the container.

## Data compression

The blocks in container can be compressed using the zlib, lzma, lz4 and flac compression methods. This will help to reduce the size of some parts of the container, like for example the headers, and the extra files included.

### Zlib

Enabling the zlib compression method, the program will compress every data stream using the zlib library. This will reduce the output file size. Zlib is a good balance between speed and compression ratio.

Optionally, a set of “checkpoints” can be created into the stream with the zlib flush option Z\_FULL\_FLUSH. This will allow to seek into the file with tools that requires random access to sectors, like for example an emulator plugin. This option will not have any impact in normal decompression but will reduce the compression ratio, so is recommended to keep it disable if ECM file will not be used for random access. A maximum of 255 sectors per block is allowed and the number must be stored in the 5th byte of the main header to help the tool used for random access to know how many sectors exists in every block.

### LZMA

Enabling the lzma compression method, the program will compress every data stream using the xz library. This will reduce the output size. Lzma compress at higher ratios than zlib compression method, but is slower in compression/decompression.

### LZ4

The lz4 compression method compress less than the zlib and lzma methods, but compression/decompression is much faster, allowing to seek into the file in less time. This is very usefull when the resources in the system are limited or you want a fast seek time for an emulator.

### FLAC

FLAC compression is available only as audio compression method because FLAC is an audio format. Is able to also compress data sectors but only if they are untouched, and that is not useful in blocks were data was removed (like ECM format does). This make it useful only in CDDA sectors on which ECM format is not able to remove any data. Also can be used in WAV data, but a way to recover the WAV header must be implemented.