**ECM2 specifications by Daniel Carrasco**

**Rev-4, October 2021**

**Alpha Version**

# Changelog

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date** | **Modifications** |
| R01 | 2021-07 | First version |
| R02 | 2021-08 | Revised document to add the new features |
| R03 | 2021-08 | New ECM format (v3). |
| R04 | 2021-10 | Better design of ECM format (v3.1) |

Contenido

[Changelog 2](#_Toc85579835)

[Document 4](#_Toc85579836)

[Document Objectives 4](#_Toc85579837)

[History 5](#_Toc85579838)

[How it works 6](#_Toc85579839)

[CDDA Sector 6](#_Toc85579840)

[Mode 1 Sector 6](#_Toc85579841)

[Mode 2 Sector 6](#_Toc85579842)

[Mode 2 XA 1 Sector 7](#_Toc85579843)

[Mode 2 XA 1 Sector 7](#_Toc85579844)

[Why to use ECM2 7](#_Toc85579845)

[Container specifications 8](#_Toc85579846)

[Main Header 8](#_Toc85579847)

[Blocks 8](#_Toc85579848)

[Block Header 8](#_Toc85579849)

[Block Data 8](#_Toc85579850)

[Allowed compressions 12](#_Toc85579851)

[LZ4 13](#_Toc85579852)

[Zlib 13](#_Toc85579853)

[LZMA 13](#_Toc85579854)

[FLAC 13](#_Toc85579855)

[None 13](#_Toc85579856)

# 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. Container specification
5. 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 1 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.

# 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. The V3.1 is an upgraded version with little modifications to make it easier, and to add support for patches files.

## Main Header

The main header starts at the beginning of the file and is composed by 12 bytes. Three of them are fixed, the 4th defines the file version and the rest is the TOC position in file (from now eTOC).

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 3 bytes | ECM | The ECM sign |
| 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. |
| 0x04 | 8 bytes |  | eTOC Position in uint64\_t |

## Blocks

Blocks are like boxes for every item in file. Will help to keep a track of the kind, size… which will help to manage the container.

### Block Header

Every block must have a header, which will help to determine the block type, size, and if it uses zlib compression. Every block header will contain a header with the following data:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Block type   1. Deleted block 2. Metadata block 3. Disk image block 4. Image patch block 5. File block |
| 0x01 | 1 byte |  | Compression (refer to compression section) |
| 0x02 | 8 bytes |  | The block size in uint64\_t (compressed size) |
| 0x10 | 8 bytes |  | The real block size in uint64\_t (without compression) |

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

#pragma pack(push, 1)

struct block {

uint8\_t type;

uint8\_t compression;

uint64\_t block\_size;

uint64\_t real\_block\_size;

};

#pragma pack(pop)

### Block Data

Block data is located after the block header, and it contents depends of the kind of block. We will explain the main blocks data here.

#### Metadata

Metadata block must be unique (only one block in the file). It can contain one or more metadata sub-blocks. Every sub-block will have 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 |
| 0x05 | Variable |  | The sub-block data |

#### Disk Image block

The Disk Image block will contain the CD-ROM image data, which can be optimized at any level (even without optimizations). This block mustn’t be compressed using zlib, because the image can have its own compression method like lzma, flac…

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

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Image ID starting at 1 (0 mean no ID), used to attach patches and files to the image. |
| 0x01 | 1 byte |  | 1 Byte with the optimizations used in this file. Useful in decoding process. |
| 0x02 | 1 byte |  | Sector per blocks in compressed data or 0 if this option was not used. |
| 0x03 | 1 byte |  | CRC type used to verify the image:   1. None 2. Standard Error Code Detection CRC 3. MD5 4. SHA1 |
| 0x04 | 8 bytes |  | Relative position of streams and sectors TOCs |
| 0x12 | 2 bytes |  | The image title size in uint16\_t |
| 0x14 | 1 byte |  | The image title encoding:   * 0: ASCII   For now, only ASCII is allowed |
| 0x15 | Variable |  | The image title |

After the Disk Image block header, the file will contain the streams and sectors TOCs, and the image data in any order (streams and sectors TOCs must be together and in that order).

The image data must be optimized using the configuration of the header, or decoding will fail. This configuration can be “zero” to just copy the raw image.

##### Streams block

The streams block localization will be before or after the image data. It will contain its own header which will help to determine the sub-blocks count and the size of the block:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 8 bytes |  | Streams count in uint32\_t |
| 0x04 | 8 bytes |  | Streams header size in uint32\_t |

The equivalent struct will be:

struct sec\_str\_size {

uint32\_t count;

uint32\_t size;

};

After the header, the block will contain a series of sub-blocks with the info of the streams that contains the file. These sub-blocks will be composed by the following:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Stream info:   * 0: Stream format → 0 = Audio, 1 = Data. * 1 - 3: Stream compression.   1. Zlib   2. LZMA   3. LZ4   4. FLAC   5. APE   6. WAVPACK * 4 - 7: Reserved |
| 0x01 | 4 bytes |  | The number of sectors that contains the stream |
| 0x05 | 4 bytes |  | The end position in output file |

And 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)

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 WavPack for audio streams, and also will help to calculate how to decompress the input stream. This blocks must be compressed using zlib compression.

##### Sectors block

Sectors block must be placed just after the Streams block. It will also contain a header to help the program to determine the sub-blocks count and also its size. The header will be the same as the Streams block:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 8 bytes |  | Sectors count in uint32\_t |
| 0x04 | 8 bytes |  | Sectors header size in uint32\_t |

The equivalent struct will be the same as the Streams block header.

As the Streams block, after the header the block will contain a series of sub-blocks with the info of the sectors kind and count, that the image has. These sub-blocks will be composed by the following:

|  |  |  |  |
| --- | --- | --- | --- |
| 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 * 4 – 7: Reserved |
| 0x01 | 4 bytes |  | The number of sectors of that type processed. This number is equivalent to an uint32\_t variable. |

In this case the equivalent struct will be:

#pragma pack(push, 1)

struct sector {

uint8\_t mode: 4;

uint32\_t sector\_count;

};

#pragma pack(pop)

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. These blocks must be compressed using zlib compression.

##### Image Data

The image data isn’t special. Just the resulting data of optimizing the sectors (or not, because is optional), and compressing the data using any of the supported compression libraries. Of course, this data must coincide with which is stored into the headers data or decoding process will not work.

The CRC must be stored at the end of the image data to verify that the read data is correct.

##### Graphical Structure of the block

In this section will show you how the block looks like with the different sections in block.

**The entire block:**

|  |  |
| --- | --- |
| ECM Block Header | Disk Image Block |

**Disk Image Block**

|  |  |  |
| --- | --- | --- |
| DIB Header | Streams and Sectors blocks | Image data |

or

|  |  |  |
| --- | --- | --- |
| DIB Header | Image data | Streams and Sectors blocks |

**Streams and Sectors headers**

|  |  |  |  |
| --- | --- | --- | --- |
| Streams header | Streams data | Sectors header | Sectors data |

**Image data**

|  |  |
| --- | --- |
| Image data | CRC |

#### Image Patch block

The image patch block is not implemented yet.

#### File block

The file block will allow to store any kind of file into the ECM2 container, allowing to link the file with a Disk Image block, or just add it as a global file. The File block will allow you to add files like for example the cue sheets, subchannels… to an image and transport all the files in just one container. This block can be compressed using zlib, so the included file size will be reduced.

Like all the blocks, this block must have the [block header](#_Block_Header) too. After the header, the block must include the File block header which is composed by the following data:

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Size | Value | Description |
| 0x00 | 1 byte |  | Disk Image block ID to which the file will be linked. Zero means global file without linked Disk Image. |
| 0x01 | 1 byte |  | CRC type used to verify the image:   1. None 2. Standard Error Code Detection CRC 3. MD5 4. SHA1 |
| 0x02 | 2 bytes |  | Filename size in uint16\_t |
| 0x04 | 1 byte |  | The File filename encoding:   * 0: ASCII   For now, only ASCII is allowed |
| 0x05 | Variable |  | File filename in ASCII |

The file size will not be included into the header, because it can be easily determined with the block size, the File block header size and the CRC size.

The file data will be included after this File block header, completing the block. The CRC data will be included at the end of the file data if required.

## Allowed compressions

The container allows to store the data in any compression method, but only the following will be used in the standard container. Those compression methods can be used in any of the stream types without problem, but to achieve the best compression ratio is better to use it in the correct way. For example, you can compress either an audio and a data stream using the FLAC compressor, but the data will achieve less compression than other compression methods like LZMA. The same will happen to the contrary, you can compress an audio stream using LZMA, but it will compress less than FLAC compression method.

The allowed standard compression methods are the following:

* LZ4
* Zlib
* LZMA
* FLAC
* None

Every compression method has its own advantages and disadvantages.

### LZ4

The LZ4 compression method is one of the best for low end platform like for example Raspberry Pi Zero. The compression speed is fast and the decompression speed is very fast. The disadvantage of this compression method is the compression ratio, which is the lower of all the allowed compression methods.

### Zlib

The Zlib compression method compress more than the LZ4 compression method, but with a tradeoff: Is slower compressing and decompressing. Is useful when the platform has a medium CPU power or the compressed data is accessed very often and there is no need to have a fast decompression.

### LZMA

LZMA compression method is the best in compression ratio, but to achieve that compression ratio it is very slow compressing and decompressing, and also is an high memory consumer, so it is recommended mainly to store the images as backup, or when the platform has much CPU power and memory.

### FLAC

The FLAC compression is the best compression method for audio streams, achieving much higher compression ratios in this kind of streams, than the rest of compression methods. It also will allow to compress data streams, but its compression ratio is lower compared with other compression methods. Is the best to store images with CDDA tracks.

### None

Well, no introduction is needed to the “none” compression method, it just doesn’t compress it all. It is included in all the above methods so its size reduction is the lower of all. On the other hand, this method will have the lower overhead when reading data, because it will only have to read the required data and regenerate the sector. All the above methods will have to decompress an entire block (which can be from 1 to several sectors), to read the sector data, and then regenerate that sector too. This method is able to achieve reduction ratios (because it doesn’t compress), of about 10-20% depending of the image data. Some games with a lot of “dummy” sectors can achieve reductions up to 80% (for example, the BassRise Fishing [SLUS-00905] game reduction is about 75%).