

# Interchangeable Preservation Format (IPF) Documentation

*By Jean Louis-Guerin (DrCoolZic)*

*Version 1.6 April 2018*

## Table of Content

<b>1. Presentation</b> .....	<b>3</b>
1.1 IPF (Interchangeable Preservation Format).....	3
1.2 IPF and Openness.....	3
1.3 Software Preservation Society.....	3
1.4 How Floppy Disks were produced.....	3
1.5 Floppy Disk Preservation with IPF files.....	4
<b>2. CAPS Analyzer Description Language</b> .....	<b>5</b>
2.1 Disk Images.....	5
2.2 Format /Track Descriptors.....	5
2.3 Block/Sector Descriptors.....	6
2.4 Data Descriptors.....	6
2.4.1 Data type field.....	6
2.4.2 Some Data type.....	6
2.4.3 Data type table.....	7
2.5 Atari Example.....	8
2.5.1 Atari Format/Track descriptor.....	8
2.5.2 Atari Block & Data descriptors.....	8
2.5.3 A word of caution.....	10
<b>3. IPF/IPX file content</b> .....	<b>11</b>
3.1 IPF File organization.....	11
3.2 IPX File Organization.....	12
3.3 Record Header.....	12
3.4 CAPS Record.....	12
3.5 Info Record.....	13
3.6 Image Record.....	14
3.7 Data Record.....	15
3.7.1 Block Descriptor.....	15
3.7.2 Data Area.....	16
3.8 CTEI Record.....	17
3.9 CTEX Record.....	17
<b>4. Atari Example</b> .....	<b>18</b>
4.1 Atari FD Format short presentation.....	18
4.1.1 Atari Track Format.....	18
4.1.2 Atari Sector Format.....	18
4.1.3 Atari Sector write splices.....	18
4.1.4 Atari Track write splices.....	19
4.2 Example of Theme Park Mystery Floppy.....	20
4.2.1 Analysis of Track 00.0 in the Stream file.....	20
4.2.2 Analysis of track 00.0 in the IPF file.....	21
<b>5. CAPS/IPF Programing</b> .....	<b>27</b>
5.1 IPF Decoder Library.....	27
5.2 My IPF File reader / writer programs.....	27
5.2.1 IPFinfo Dos console program.....	27
5.2.2 IPF File reader / Writer.....	27
<b>6. Glossary</b> .....	<b>28</b>
<b>7. References</b> .....	<b>31</b>
<b>8. History</b> .....	<b>31</b>

## 1. Presentation

---

### 1.1 IPF (*Interchangeable Preservation Format*)

IPF stands for [Interchangeable Preservation Format](#), and is the file format defined and used by Software Preservation Society (SPS) to preserve content, that is, floppy disk images. It's naming is quite abstract for very good reasons:

- It is platform agnostic.
- It is low-level, representing the information as it would have been read by a drive head.
- It is not just for disks: Any type of digital media can be contained by the format. Currently it is used for disk images and ROM files, but in the future it may possibly be used for CDROMs and information contained in hardware “dongle” protections.
- The underlying file format has many similarities to the IFF (Interchangeable File Format) standard on the Amiga, which was a very forward-thinking format. The same techniques are used today in other file formats such as PNG.

### 1.2 IPF and Openness

This subject is described in [IPF and Openness subject](#) in the Kryoflux forum. After long debate the IPF source code has been made available under MAME license (see [IPF DECODER LIBRARY source code released](#)). Therefore, the IPF format is now open but unfortunately completely undocumented. This is not felt as a problem by most user as the content of IPF files is supposed to be read through the IPF decoder library and **written only** by SPS people. In this document I try my best to provide information about the IPF file format so it should be possible to **read** and **write** IPF files **directly**.

---

 However, beware that the information presented in this documents comes from different sources and inevitably must contain errors and therefore must be used with caution.

---

### 1.3 Software Preservation Society

[The Software Preservation Society \(SPS\)](#) dedicates itself to the preservation of software for the future, namely classic games. As it is, these items are no longer available from their original suppliers, and are mainly in the possession of an ever diminishing community of well willed collectors. However, just by the passage of time these games are affected by the gradual deterioration of the media that stores them. These classics risk being lost forever in the near future, a tragedy that must be prevented.

### 1.4 How Floppy Disks were produced

First it is important to understand how diskettes *were* produced. Although it would have been possible to create non-protected diskettes directly on the target machine, in practice, especially to manufacture large quantity of disks, it was necessary to use dedicated *commercial floppy disk mastering machines*. Beside the capability to produce large quantity of disks these machines were also capable to write information that *could not be reproduced* on the target machine. This technique is often referred as [floppy disk copy protections](#). The copy protection method has at least two obvious qualities: first, a *key protected disk* can be simultaneously used as *protection* and *distribution* disk and second, this type of protection is very cheap but nevertheless hard to tamper with. The mastering machines were using dedicated description languages to describe pretty much any disk format, including the ones with copy protection. The duplication machines knew about complex magnetic recording theory in order to be able write perfect disks based on the description. For example the mastering machines from [Trace Products](#) were using the *Freeform* description language for that matter. The same description language could also be used to check that what has been written to a disk was correct.

### 1.5 Floppy Disk Preservation with IPF files


The term preservation means different things to different people. Most of the time, only the minimum required information about a floppy disk is saved so that it is possible to **emulate** correctly, for a specific platform, what would be read from the original disk. While this may be sufficient to run on an emulator a “preserved game” it should **not** be considered as preservation because a lot of information from the original master is lost. This is why SPS people have taken an innovative approach that consists in retrieving the original master description of the disk to preserve using a language equivalent to Freeform. Basic information about this language is described in the [CAPS Analyzer Description Language](#) chapter. Finding the original “Freeform” description can be compared to breaking a vault combination lock: it is a very hard process but once done you know that you have a perfect result. The description is then stored in IPF files that contain a combination of both the disk data and how this data is stored. IPF files can then be used either to master the disk images back to floppy disk or in emulation tool.


SPS have developed image creation tools (See [Kryoflux web site](#)) that can read disk information at very low level. The disk image containing the raw signal information is loaded into an Analyzer (**CTA CAPS Analyzer**) where known disk formats are semi-automatically detected. The disk image is saved using the IPF format which contains the disk data, how it is stored (and hence how to write it back), and also a few other things needed to store copy protection (like how dense the data is at any part of each track). It is interesting to note that the IPF files does not store directly detailed information often found in other preservation formats. For example, the timing information measured by the preservation tool is **not stored** but the “format description” can for example indicate that specific sectors of a track use a well-known bit-cell density variation. The result is therefore perfect as the timings are not subject to measurement errors.

☞ *Note that the CAPS Analyzer does not support archival of modified disks that are based on generic MFM coding and were written on a standard machine (PC, Atari ST, etc.). The reason is that, except the Commodore Amiga that will always write a complete track at once, other machines equipped with standard controllers (e.g. Western Digital WD177x) replace single sectors only. Starting and stopping the magnetic fields of the write head will generate undefined data in GAPS as well as slight drift of the complete sector in relation to the rest of the track (also known **as write splice**).*

## 2. CAPS Analyzer Description Language

This chapter is a short presentation of the mastering description language used by the **CAPS Analyzer** (also referred as the **Analyzer** or **CTA** in this document) as described in the [WIP of July 18, 2002: \*Time for an even more generic way to describe disk formats\*](#) as well as some other WIP (see [references](#)). This document does not go into details (mainly because I do not know them) but nevertheless presents useful information required to better understand the IPF Format. The “CAD” Language is the SPS equivalent of the “Freeform” Language used by [Trace Machine](#).

 The information presented in this chapter is mainly coming from several WIP documents and from discussion with SPS’s developers. Therefore, beware that some descriptions are speculative and may be wrong.

 Note that I am primarily knowledgeable about the Floppy Disks format used on Atari ST. Therefore, all my examples are given using this format but it should be possible to transpose the information presented here to your format of choice.

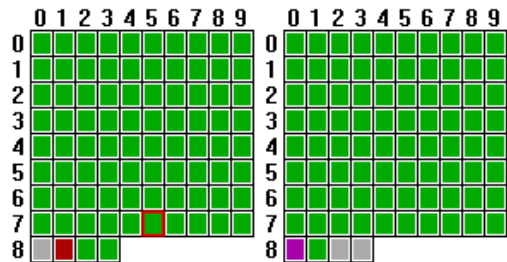
We do the description from the top level of a simple disk image and we go downwards to the most basic level used by the **Analyzer** (the [Data Descriptor](#) type):

- Disk Images
- Format /Track Descriptors
- Block Descriptors
- Data Descriptors
- Data Field Types

We first give a generic explanation of the different descriptors followed by a detail example of an Atari disk format description.

### 2.1 Disk Images

The disk images going into the **Analyzer** are floppy disks that are dumped by contributors with the [Kryoflux tools](#). They normally contain 84 cylinders (0-83) if the disk drive used can safely read up that far, but the image format itself may contain any number. It is unlikely many games will use cylinders that high, but it is best to be sure.



### 2.2 Format /Track Descriptors

In order to recognize a track, we need to describe it. When the analysis occurs the Analyzer finds the format that best matches a track or indicates that the track is unknown. Any integrity information held by the track format is put into the track description and checked against to indicate any errors on the disk.

Most of the formats are known to be re-used several times. When we think the format was not re-used we name it after the game, otherwise we name it after the author (if known) or company. If something makes it clear it was derived from another format, it will be normally named as a variant, but it may be still be marked as such after author/company info if nothing better comes along.

Each track can contain an entirely different format to the next as **so we define disk formats on a track-by-track basis**.

A track can contain one or more blocks of data; the exact number depends on the format. For example, a standard Atari format track contains 9 blocks. Each of these blocks is described by a block descriptor.

## 2.3 Block/Sector Descriptors

A track can contain one or more blocks of data (sometimes referred as sectors); the exact number depends on the format. As the name implies, a Block Descriptor is the representation of the block data format in the **Analyzer**.

A block descriptor is described by a series of Data Descriptors and their parameters.

## 2.4 Data Descriptors

Format descriptors are defined by what block descriptors are used in them and whatever parameters are needed. In turn, blocks are described in terms of Data descriptors and their parameters.

Data descriptors are user-defined descriptions for a chunk of data. They indicate the “data field type” along with parameters to nail down exactly what the chunk of data is, and what it is used for. A data chunk or item is the smallest element of a format.

Block descriptor uses data descriptors as user definable functions. If you were programming, think of it as defining your functions first (in our case this would be the data descriptors) and then using them in context (in our case in the block descriptors). So basically, all the names are arbitrary. The person who writes the descriptor chooses the actual name of the format.

Data descriptor contains generic parameter fields:

- **Data Type:** The data field type to use, this is explained below.
- **Count:** Area number or data count.
- **Value:** Expected value.
- **SubType:** Depending on its context, this is information about the item, like encoder type or the basic type of data (byte, word, long word, etc.).
- **Process:** Processing requirement, like encryption.
- **Bit Start, Bit Stop:** These are values are always represented as 32 bits internally and it is possible to reference only a bit field of them so the complete value is marked as 0, 31. But any valid field selected can now be processed.
- **Arg1...4:** Generic placeholders for arguments to be passed to (for example) external processes.
- **VarSet:** Selects a variable where decoded data is placed. If more than one piece of data is being decoded by the data element, the variable is the starting point of the array being filled with decoded data.

### 2.4.1 Data type field

The description language is a typed system. Every data descriptor must indicate the type of data it represents; they are hard coded functions that define the “type” of each chunk of “data” on a block. Just like the C programming language where you might define a data type to be “byte”, “int” or “long”, here we define things as “**Mark**”, “**Sync**”, “**Data**”, etc.

### 2.4.2 Some Data type

In this section we provide some examples of data type that we will be using in the [Atari example](#).

#### 2.4.2.1 Area start/stop

Sets the beginning and the end of a **logical data block**. A data block is an area made of one or more data elements. After each we define the area number, which can be used to reference the area. This is any arbitrary value in the range **1..n**.

# IPF – Interchangeable Preservation Format Documentation

## 2.4.2.2 Encode

An arbitrary number of layers can be applied to any area. The encoding layer has the following parameters:

- **count**: Area number where encoding applies.
- **value**: Physical encoding (MFM, MFM2, FM, GCR...).
- **subtype**: Logical encoding
- **process**: Any process to be applied over the data, like encryption.

## 2.4.2.3 Mark / Sync

Also known as **Sync**, renamed to be more in line with Trace terminology. Treated as data, but encoding can be modified, and of course used to spot areas on a track in the first place.

- **count**: number of sync
- **value**: sync value
- **subtype**: 1 (one byte)

## 2.4.2.4 Data

Data area, all of which support:

- **count**: Count of data in its un-encoded format, i.e. the real size, which completely depends on the encoding used.
- **value**: Expected value.
- **subtype**: Type of un-encoded data, 1=byte, 2=word, 3=long-word.

## 2.4.2.5 Gap

This is another data area, but its size may be altered via mastering - if it is allowed, and errors found after it can normally be ignored.

## 2.4.2.6 EDC / Checksum

Better known as a **Checksum**, but this is more in line with Trace terminology. A checksum is generated for the selected area, and stored on the spot of the command. Checksums can be nested and/or overlapped; the **Analyzer** resolves the dependencies and thus the correct order of calculating the values.

- **count**: Area number where the process is performed on original data. Due to various shortcuts in their code, some EDCs seem like they were originally performed on raw data but in reality they worked with un-encoded data.
- **value**: Starting value for the checksum process. Sometimes it is an arbitrary value, though most of the time 0.
- **subtype**: Type of un-encoded data, 1=byte, 2=word, 3=long-word.
- **process**: The checksum method.

## 2.4.3 Data type table

Function	Type	Count	Value	SubType	Process	VarSet
<b>Encode</b>	Area encode	BP #0	BP #1	BP #2	BP #3 (unused in example)	BP #4 (unused in example)
<b>Area start</b>	Area start	BP #0	n/a	n/a	n/a	n/a
<b>Gap, byte</b>	Gap	BP #0	BP #1	1 (i.e. byte)	n/a	n/a
<b>Mark, byte</b>	Mark	BP #0	BP #1	1 (i.e. byte)	BP #2	n/a
<b>Data, byte</b>	Data	BP #0	BP #1	1 (i.e. byte)	n/a	BP #2 (unused in example)
<b>Data, byte f0</b>	Data	BP #0	FP #0	1 (i.e. byte)	n/a	BP #2 (unused in example)
<b>Sector, byte</b>	Data	BP #0	FP #0	1 (i.e. byte)	n/a	n/a
<b>Area stop</b>	Area stop	BP #0	n/a	n/a	n/a	n/a
<b>EDC, Xor16, long</b>	EDC	BP #0	BP #1	3 (i.e. longword)	5 (i.e. CRC16 method)	n/a

## 2.5 Atari Example

Now we have gone through a short description of the descriptors let's see an example of how we describe a standard Atari track format.

### 2.5.1 Atari Format/Track descriptor

```
.Header, Data block: 1, 512 *1 <1st>
.Header, Data block: 2, 512 *1
.Header, Data block: 3, 512 *1
.Header, Data block: 4, 512 *1
.Header, Data block: 5, 512 *1
.Header, Data block: 6, 512 *1
.Header, Data block: 7, 512 *1
.Header, Data block: 8, 512 *1
.Header, Data block: 9, 512 *1
```

Each line looks like this

```
[Block Descriptor] : [block_number], [number_bytes], *weight
```

The parameters are passed down to the block and data descriptors. These lower level descriptors have access to Format Parameter #0, Format Parameter #1 and Format Parameter #2. So now let's have a look at the data descriptors that access them.

- Format Parameter #0: set to 1...9, used by the "Sector, byte" data descriptor, i.e., we search for blocks (sector) 1 to 9.
- Format Parameter #1: set to 512, used by the "Data, byte f1" data descriptor.

You might wonder why not use fixed values. The reason is that many games on floppies use slight differences in the format descriptor.

### 2.5.2 Atari Block & Data descriptors

```
Encode: 1, 1
_Area start: 1
  _Area start: 2
    Mark, byte: 3, $4489 *1 <esc>
    ID, byte: 1, $FE *1 <esc>
    Track, byte: 1 *1
    Side, byte: 1 *1
    Sector, byte f0: 1 *1 <esc>
    Length, byte: 1 *1
  _Area stop: 2
  EDC, CRC16: 2, $FFFF *2
  GAP,byte: 34 *1
  _Area start: 3
    Mark, byte: 3, $4489 *1 <esc>
    ID, byte: 1, $FB *1 <esc>
    Data, byte f1 *1
  _Area stop: 3
  EDC, CRC16,short: 3, $FFFF *3
_Area stop:1
Set GapPatern: 40 $4E
Set GapPatern
Set GapPatern: 12, $0
Set GapPatern
```

Description of the statements

```
Encode: 1, 1
```

First we define Area 1 (first parameter) to be encoded as MFM (second parameter). The whole block is covered by Area 1, and therefore all data elements are set to be MFM encoded.

```
_Area start: 1
```

Start Area 1. Just marking an area to cover all the data to encode.

```
_Area start: 2
```

Next is where Area 2 starts. This is the ID/Address area of the sector.



## IPF – Interchangeable Preservation Format Documentation

```
Mark, byte: 3, $4489 *1 <esc>
```

We have 3 sync bytes with value 4489 (\$A1 with missing clock) in MFM. Weight 1. We skip (escape) to the next sync if there is no match

```
ID, byte: 1, $FE *1 <esc>
```

1 ID byte with value \$FE, weight 1, and skip to next block if not equal

```
Track, byte: 1 *1
```

1 byte that contains the track value weight 1

```
Side, byte: 1 *1
```

1 byte that contains the side value weight 1

```
Sector, byte f0: 1 *1 <esc>
```

1 byte that contains the sector value. This value must match the value given in parameter 0 of the format descriptor (Format Parameter #0). Skip to next block if not equal

```
Length, byte: 1 *1
```

1 byte that contains the size value

\_Area stop: 2

End of the ID/Address field of the sector.

```
EDC, CRC16: 2, $FFFF *2
```

The checksum for the ID/Address field is next, which is a CRC16 type checksum on Area 2. Starting value=\$FFFF. It is weighted by a factor of 2 so the score will be worse if the checksum calculated on the header does not match the value stored here.

```
GAP,byte: 34 *1
```

There are then 34 bytes, with the value of 0 defining a gap. Weight 1x.

```
_Area start: 3
```

Next is where Area 3 starts. This is the Data area.

```
Mark, byte: 3, $4489 *1 <esc>
```

We have 3 sync bytes with value 4489 (\$A1 with missing clock). Weight 1. We skip (escape) to the next sync if there is no match

```
ID, byte: 1, $FB *1 <esc>
```

1 ID (identification) byte with value \$FB, weight 1, skip to next block if not equal

```
Data, byte f1 *1
```

Then we have the data field. The length of the field is fixed by parameter 1 (sector size = 512) of the format descriptor (Format Parameter #1).

```
_Area stop: 3
```

End of the data field of the sector.

```
EDC, CRC16,short: 3, $FFFF *3
```

The checksum for the data field is next, which is a CRC16 type checksum on Area 3. Starting value=\$FFFF. It is weighted by a factor of 3 so the score will be worse if the checksum calculated on the data does not match the value stored here, i.e. the integrity of data area is the most important.

```
_Area stop:1
```

End of area 1.

```
Set GapPatern: 40 $4E  
Set GapPatern  
Set GapPatern: 12, $0  
Set GapPatern
```

These lines define the forward and backward gaps between sectors.

## 2.5.3 A word of caution

Now that the IPF format is supported by several Atari emulators it is easier to check if an Atari floppy disk has been correctly analyzed by the CTA program: you just need to write an IPF file for this floppy and test it under an emulator like [Steem](#) or [Hatari](#).

In the previous revision of this document the support of IPF was not available and therefore it was harder to check, at that time, if a correct analysis had been done by the CTA program. During my tests, I thought that many “protections” were not detected correctly because most of them appears under a generic ***\_MFM xxx*** format descriptor name. Furthermore, if you looked inside this generic ***\_MFM*** format descriptor it uses itself a generic ***Header, Data Block:*** block descriptor.

To get to the actual protection you have to look inside the block descriptor to find out that even if uses the same generic name, its content can be quite different.


For example, the ***Header, Data Block*** descriptor used in Jupiter Masterdrive is

```
Encode: 1, 1
_Area start: 1
  Data, byte: 8 *1
  Mark byte: 1, $4489 *1
  Data byte: 16 *1
  _Area start: 2
    Mark, byte: 3, $4489 *1 <esc>
    ID, byte: 1, $FE *1 <esc>
    Track, byte: 1 *1
    Side, byte: 1 *1
    Sector, byte f0: 1 *1 <esc>
    Length, byte: 1 *1
  _Area stop: 2
  EDC, CRC16: 2, $FFFF *2
  GAP,byte: 34 *1
  _Area start: 3
    Mark, byte: 3, $4489 *1 <esc>
    ID, byte: 1, $FB *1 <esc>
    Data, byte f1 *1
  _Area stop: 3
  EDC, CRC16,short: 3, $FFFF *3
_Area stop:1
Set GapPattern: 40 $4E
Set GapPattern
Set GapPattern: 12, $0
Set GapPattern
```

We can see at the beginning that we have 8 data bytes followed by a sync mark followed by 16 data bytes. This information is coded as the following blocks in the IPF file:

```
28 29 55 42 49 4E 4E 29
4489
CD B4 F7 00 DE AD C0 DE 00 00 00 00 00 00 00
```

---

 So the rule is: Don't just rely on a Format or Block descriptor name to find out about its actual content. You need to realize that the same name can cover quite different content.

---

### 3. IPF/IPX file content

This section provides information about the content of IPF/IPX files, as produced by **CTA CAPS Analyzer**. This can be useful to better understand what information is stored in IPF/IPX file and it can be used to write software that directly read or write IPF files.

- ✍ However it is **highly recommended** not to access the content of IPF files directly but through the [IPF/CAPS library](#) since the IPF format is complex and can be modified at any time in order to add new capabilities.
- ✍ The IPF/IPX files content description provided here is largely based on my interpretation of the [IPF/CAPS library](#) and therefore beware that some descriptions may be wrong. So again prefer the usage of the provided library.

🔊 You first have to realize that the content of an IPF file is not what you would typically read from a specific FDC on a specific machine (for example what you would read using a WD1772 Floppy Disk Controller on an Atari). The content of an IPF file is what is **written** on a disk using a specific hardware like a **Kryoflux board** connected to a floppy drive (same as what would have been written on a **Trace machine**).

This imply that the content of an IPF file is directly used to write a Floppy disk with a KryoFlux board, but in order to be used by an emulator the file needs to be interpreted as a FDC would do. This is another reason why it is recommended to use the CAPS/IPF library to read an IPF file.

#### 3.1 IPF File organization

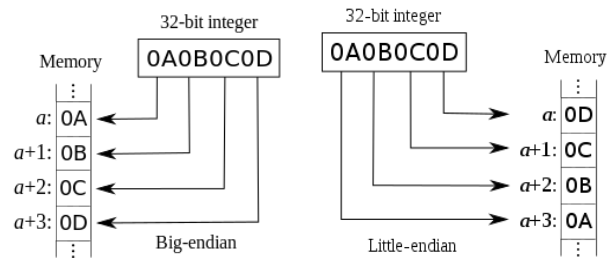
The IPF file is organized in chunks of data that I call **records**. A record is composed of a fixed size **Header** block followed by a **Type Specific** block and finally by an optional **Extra data** block.

Therefore, a generic IPF record looks like this:



Most of the information in the file is stored as **4 bytes** (32 bits) integers using **big-endian** ordering. When reading the information on an **Intel** platform (**little-endian**) it is necessary to swap the order of the bytes to interpret the information correctly.

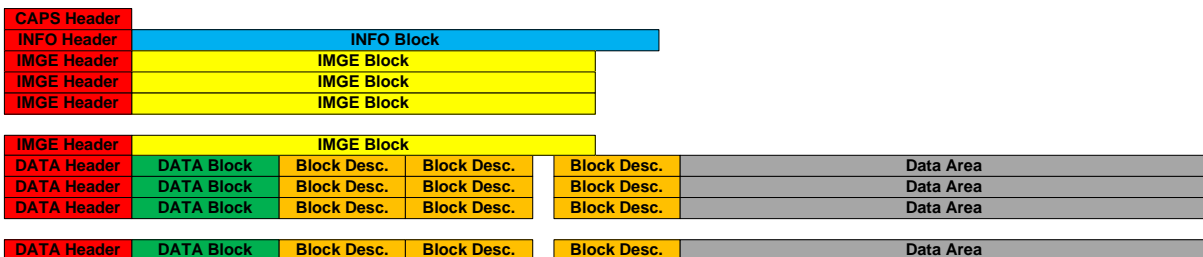
Each block of data in an IPF file is complemented with a CRC32 value that allows to verify that the block of data read has not been corrupted.



An IPF file describes the content of a floppy disk and is typically composed of the following records:

- First a [CAPS record](#) used to identify an IPF file
- This record is followed by an [INFO record](#) that gives general information about the IPF file.
- This record is followed by [IMGE records](#): usually one per track and per side of the floppy disk.
- These records are followed by [DATA records](#): one for each IMGE record.

This can be represented with the following diagram:

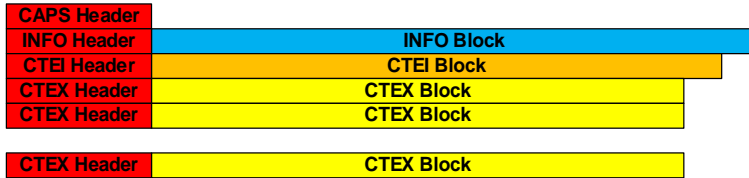


### 3.2 IPX File Organization

The CAPS Analyzer can write an IPX file at the same time it writes an IPF file. An IPX file provides information about the analysis of a floppy disk and is composed of the following records:

- First a [CAPS record](#) to identify an IPF/IPX file
- This record is followed by an [INFO record](#) that gives general information about the IPF file
- This record is followed by a [CTEI records](#): provides information about the Analyzer.
- These records are followed by [CTEX records](#): one for each track.

This can be represented with the following diagram:



### 3.3 Record Header

A record header is composed of 3 fields:

- **type** (4 characters): that indicates the record's type. Currently defined types:
  - ◆ [CAPS](#): Used to identify an IPF file
  - ◆ DUMP: Unknown to me
  - ◆ [DATA](#): contains the actual data from the floppy disk
  - ◆ TRCK: Unknown to me
  - ◆ [INFO](#): contains general information about the IPF file
  - ◆ [IMGE](#): contains track information
  - ◆ [CTEI](#): Contains information about the Analyzer.
  - ◆ [CTEX](#): Contains information about analysis of a track.
- **length** (4 bytes): Size in bytes of the complete record (i.e. the header followed by specific block).
- **CRC** (4 bytes): Store the computed CRC32 value for the complete record (i.e. the record header block plus the specific block). The CRC value is computed on the complete record with the CRC32 field set initially at 0x0000, and then the resulting CRC32 value is recorded in this field.

To move from one record start address to the next record start address you need to add the length field to the start address. The only exception is for data record as explained in [data record](#).

### 3.4 CAPS Record

#### CAPS Header

The content of the **type** field in this record header is “CAPS” in uppercase.

The CAPS record must be the first record in the file and is used to identify a CAPS/IPF file.

A CAPS record is composed of only a header block and therefore is not followed by any other block as it is the case for all other types of records.

The overall size of a CAPS record is 12 bytes.

### 3.5 Info Record



The **Info record** provides general information about the IPF file. The Info record is composed of a [record header](#) followed by an info block. The **type** field in the record header is “**INFO**” in uppercase.

The **Info header** is followed by an **Info block** that contains the following information:

- **mediaType** (4 bytes): The type of media imaged.
  - ◆ 00 = Unknown
  - ◆ 01 = Floppy disk (the only type supported at time of writing).
- **encoderType** (4 bytes): The image encoder type. Depending on the encoder type some fields in the [IMAGE records](#) are interpreted differently. This field can take the following values:
  - ◆ 00 = Unknown
  - ◆ 01 = CAPS encoder (historically first encoder)
  - ◆ 02 = SPS encoder (more recent should be preferred for writing new files).
- **encoderRev** (4 bytes): The image encoder revision of a specific **encoderType**. Currently only revision 1 exists for CAPS or SPS encoders.
- **fileKey** (4bytes): Each IPF file should have a unique ID that can be used as a key for database. More than one file can have the same **fileKey** for example for a game that has more than one disk. It is the responsibility of the writer to use a unique value.
- **fileRev** (4 bytes): Revision of the file. Initially the revision is one.
- **origin**: The CRC32 value of the original .ctr source file.
- **minTrack**: The lowest track number of the floppy image. Usually 0
- **maxTrack**: The highest track number of the floppy image. Usually 83
- **minSide**: The lowest side (head) number of the floppy image. Usually 0
- **maxSide**: The highest side (head) number of the floppy image. Usually 1
- **creationDate** (4 bytes): Specify the image creation date (year, month, and day) encoded.
- **creationTime** (4 bytes): Specify the image creation time (hour, minute, second, and tick) encoded.
- **platforms**: An array of four 4 bytes’ integers used to store the value of the intended platforms (one image can run on several platforms). Platform can take the following values:
  - ◆ 00 = None
  - ◆ 01 = Amiga
  - ◆ 02 = Atari ST
  - ◆ 03 = PC
  - ◆ 04 = Amstrad CPC
  - ◆ 05 = Spectrum
  - ◆ 06 = Sam Coupe
  - ◆ 07 = Archimedes
  - ◆ 08 = C64
  - ◆ 09 = Atari 8-bit
- **diskNumber** (4 bytes): Number of the disk in a multidisc release otherwise 0
- **creatorId** (4 bytes): A unique ID of the disk image creator.
- **reserved** (12 bytes): an array of three 4 bytes’ integer reserved for future use.

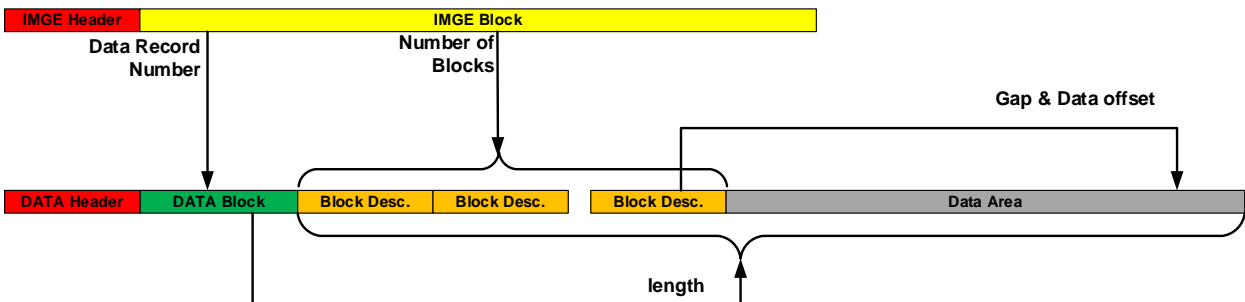
The overall size of an **Info record** is 96 bytes (12 for the header block and 84 for the Info block).

### 3.6 Image Record



The content of the **name** field in the [record header](#) field is “**IMGE**” in uppercase. An **Image record** provides general information *for a specific track* of the floppy and is composed of an **Image header** followed by an **Image block** that contains the following information:

- **track** (4 bytes): The track (cylinder) number. Usually in the range 0-83
- **side** (4 bytes): The side (head) number. Usually 0 or 1
- **density** (4 bytes): Density of the track. Currently density can take the following values:
  - ◆ 00 = Unknown
  - ◆ 01 = Noise: cells are unformatted (random cell size)
  - ◆ 02 = Auto: size of cells is automatically adjusted to track size
  - ◆ 03 = Copylock Amiga
  - ◆ 04 = Copylock Amiga new
  - ◆ 05 = Copylock ST
  - ◆ 06 = Speedlock Amiga
  - ◆ 07 = Old Speedlock Amiga
  - ◆ 08 = Adam Brierley Amiga
  - ◆ 09 = Adam Brierley, density key Amiga
- **signalType** (4 bytes): Signal processing type:
  - ◆ 00 = Unknown
  - ◆ 01 = 2µs cells
- **trackBytes** (4 bytes): Rounded number of decoded bytes (clock + data) on track
- **startBytePos** (4 bytes): The rounded **startBitPos** position in byte – *useless*.
- **startBitPos** (4 bytes): Start position in bits (clock + data) – from Index - of the first sync bit of the first block. Note that the track write splice is located between the last block and this block.
- **dataBits** (4 bytes): number of decoded data bits (clock + data). Note that this number also includes the gap bits located **inside** a block/sector (i.e. the **intra** id-data gap bit). This value is equal to the sum of the **dataBits** specified for all [block descriptors](#).
- **gapBits** (4 bytes): number of decoded gap bits (clock + data) Note that this number only includes the gap bit **between** blocks/sectors (i.e. the **intra** id-data gap bits are excluded). This value is equal to the sum of the **gapBits** specified for all [block descriptors](#).
- **trackBits** (4 bytes): Number of bits of the track. **trackBits = dataBits + gapBits – useless**
- **blockCount** (4 bytes): Number of blocks used to describe one track. Usually (but not necessarily) this is the number of sectors for the current track. This field also indicates the number of data [block descriptors](#) in the matching [DATA record](#).
- **encoderProcess** (4 bytes): Encoder process = 0
- **trackFlags** (4 bytes):
  - ◆ Bit 0 = Fuzzy (indicates that track contains fuzzy bits).
- **dataKey** (4 bytes): Unique key used to match the [DATA record](#) that uses the same **dataKey**.
- **reserved** (12 bytes): an array of three 4 bytes’ integer reserved for future use.



The overall size of an IMGE record is 80 bytes (12 for the header block and 68 for the IMGE block).

### 3.7 Data Record



The content of the **type** field in the [record header](#) is “DATA” in uppercase. The **Data record** contains the data information for *a specific track* of the floppy. A **Data header** is followed by the **Data block**, and *eventually* by an **Extra Data Block**. The **Data block** contains the following information:

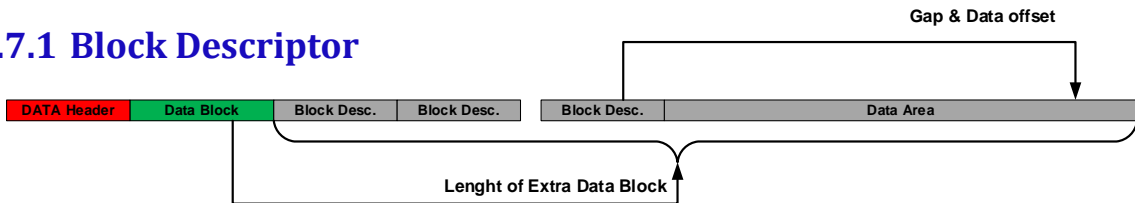
- **length** (4 bytes): size in bytes of the **Extra Data Block** following the data block (can be zero).
- **bitSize** (4 bytes): data area size in bits (**length** \* 8) – *useless*
- **CRC** (4 bytes): CRC32 of the complete Extra Data Block
- **dataKey** (4 bytes): Unique key used to match the same key in [IMGE record](#).

The overall size of an IMGE record is 28 bytes (12 for the header block and 16 for the IMGE block).

If **length** is null, this indicates that the **Data block** is **not** followed by an **Extra Data Block**. To get to the *next record* you must add the **length** field to the start address of the **Extra Data Block**.

The **Extra Data Block** starts with **blockCount Block Descriptors** (**blockCount** is specified in the matching [IMGE record](#)) and is followed by the actual data content (**Data Area**) for all these blocks

#### 3.7.1 Block Descriptor



Each Block Descriptor contains the following information:

- **dataBits** (4byte): size in bits of the decoded block data (sync + data + intra\_gap)
- **gapBits** (4byte): size in bits of the decoded gap (only inter\_gap)
- If the **encoderType** = 1 (CAPS) in the [INFO record](#) we have the following fields
  - ◆ **dataBytes** (4bytes): decoded data size, rounded
  - ◆ **gapBytes** (4 bytes): decoded gap size, rounded
- Else if the **encoderType** = 2 (SPS) in the [INFO record](#) we have the following fields:
  - ◆ **gapOffset** (4bytes): offset of the first gap stream elements in data area
  - ◆ **cellType** (4bytes): bit cell type (1 = 2 μs cell)
- **encoderType** (4bytes): Block encoder type
  - ◆ 00 = Unknown
  - ◆ 01 = MFM
  - ◆ 02 = Raw (for test only)
- **blockFlags** (4bytes): set to zero and ignored if **encoderType** = 1 (CAPS) in [INFO record](#)
  - ◆ Bit 0 ForwardGap = When set it indicates associated forward gap stream elements
  - ◆ Bit 1 BackwardGap = When set it indicates associated backward gap stream elements
  - ◆ Bit 2 DataInBit: = Unit of the data stream sample length: 0 = bytes, 1 = bits. **Warning:** only apply to data stream element but **not** to gap stream element (always in bits).
- **gapDefault** (4bytes): Default Gap value
- **dataOffset** (4bytes): Offset value of the data stream in the extra data area

The Block Descriptors are followed by the actual **Data Area**. To get to a specific data or gap stream element you must use the **dataOffset** and/or the **gapOffset** information from the Block Descriptor.

A block descriptor is 32 bytes long.

The first block descriptor corresponds to the first block written on disk *after* the write gate is turned on (and off). Normally this is the first block after the index pulse, but on “shifted tracks” this first block can be placed anywhere relative to the index. The **startBitPos** in the [Image Record](#) gives the position of this block relative to the index. The exact location where the write gate is turned on can be computed from the [gap elements specification](#) of the last block. The sum of the **dataBits** and **gapBits** fields in all block descriptors is equal respectively to the **dataBits** and **GapBits** values in the [IMGE record](#).

## 3.7.2 Data Area

The **Data Area** starts immediately after the last block descriptor and is composed of lists of stream elements. The stream elements in the data area can be freely organized as they are pointed by the **dataOffset** and **gapOffset** fields of the [block descriptors](#). CAPS analyzer writes first all the gap stream elements lists, followed by all the data stream elements lists.

### 3.7.2.1 Gap Stream Elements

If the Gap stream list exists (i.e. **encoderType** = SPS, **gapBits** not null, and **blockFlags** not null) the **gapOffset** field of the [block descriptor](#) indicates the position of the first gap streams element for this block (relative to the start of the **Extra Data Block**).

There can be 0, 1, or 2 Gap stream element list depending of the **blockFlags** in the [Block descriptor](#).

- **Bits 1-0 = 00**: No gap stream. In this case the gap is filled with the **gapValue** byte specified in the Block Descriptor. This byte will be repeated (including a final partial repetition, if necessary) as many times as necessary to fill the gap. The decoder fills forward and backward in the gap at the same time, hence the “write splice” or “overlap” occurs in the exact middle of the gap area.
- **Bits 1-0 = 01**: Forward gap stream list only. The gap stream area contains a list of gap stream elements used to fill the gap in forward direction (from the end of current data block to the beginning of the next one).
- **Bits 1-0 = 10**: Backward gap stream list only. The gap stream area contains a list of gap stream elements used to fill the gap in backward direction (from the beginning of the **next** data block to the end of the current one).
- **Bits 1-0 = 11**: Forward and Backward stream lists. The gap stream area contains two lists of gap stream elements used by the decoder to fill the gap in forward direction (from the end of current data block to the beginning of the next one) and in backward direction (from the beginning of the next data block to the end of the current one) simultaneously. The track “write splice” or “overlap” occurs in the gap area where the two streams meets.

Each gap stream element is composed of the following:

- **gapHead** (one byte) decomposed into:
  - ◆ **gapSizeWidth** (3 MSB - bits 7-5): number of bytes used to specify the **gapSize**
  - ◆ **gapElemType** (5 LSB - bits 4-0): type of gap element: 1 = **GapLength**, 2 = **SampleLength**
- **gapSize** (**gapSizeWidth** bytes using big-endian ordering): Indicates the size:
  - ◆ If **gapElemType** =1 this size indicates the number of times the **gapSample** must be repeated.
  - ◆ if **gapElemType** =2 this size indicates the size in bits of the following **gapSample**
- **gapSample** (**gapSize** bits): This is a field that only exists for **gapElemType** =2 and contains the actual **gapSample** composed of **gapSize** bits.

A list of gap stream elements is terminated by a gap stream element with a null **gapHead**.

Normally a gap stream element specification is composed of a gap repeat number (specified with a **GapLength** element) followed by a gap value (specified with a **SampleLength** element). However, the gap value repeat number is optional. If not specified this indicates that the gap sample value need to be repeated *as many times as necessary* to fill the gap field as specified in the **gapBits** size of the [block descriptor](#).

When both forward and backward gap specifications are given without repeat number the decoder fill the gap in forward and in backward direction simultaneously and the *track write splice* is where they meet.



## 3.7.2.2 Data Stream Element

If the data stream element list exists (i.e. **dataBits** is not null) the **dataOffset** indicates the position of the first Data Streams element from the start of the extra data block.

Each data stream element is composed of the following sequence:

- **dataHead** (one byte)
  - ◆ **dataSizeWidth** (bits 7-5): number of bytes used to specify the sample size
  - ◆ **dataType** (bits 4-0): type of data: 1 = Sync, 2 = Data, 3 = Gap, 4 = Raw, 5 = Fuzzy
- **dataSize** (**dataSizeWidth** bytes using big-endian ordering): The size of the following data stream samples. The size unit is specified by the **DataInBit** flag (bit 2) in the [block descriptor](#).
- **dataSample** (**dataSize** bit/bytes): The actual stream samples. This field does **NOT** exist if the **DataType** = 5 (Fuzzy) – In that case the data size specify the length of the fuzzy data block but not the size of the **dataSample** field which is null in that case – it is the responsibility of the consumer to generate random data with the specified size.

The list of data stream elements is terminated by a stream element with a null **dataHead**.

## 3.8 CTEI Record



The content of the name field in the record header field is “**CTEI**” in uppercase.

The CTEI record provides information about the analyzer used to create the IPF file.

The CTEI header is followed by the CTEI block that contains the following information:

- **releaseCrc** (4 bytes): The CRC of the associated IPF file released
- **analyzerRev** (4 bytes): The revision of the Analyzer used
- **Reserved** (14 bytes): Reserved for future used usually set to 0

The overall size of a CTEI record is 76 bytes (12 for the header block and 64 for the CTEI block).

## 3.9 CTEX Record



The content of the name field in the record header field is “**CTEX**” in uppercase.

The CTEX record provides analyzer information for a specific track of the floppy.

The CTEX header is followed by the CTEX block that contains the following information:

- **track** (4 bytes): The cylinder (track) number. Usually in the range 0-83
- **side** (4 bytes): The head (side) number. Usually 0 or 1
- **density Type** (4 bytes): Indicate the density type. Currently density can take the following values:
  - ◆ 01 = Noise cell are unformatted (random size)
  - ◆ 02 = Auto automatic cell size, according to track size
  - ◆ 03 = Copylock Amiga
  - ◆ 04 = Copylock Amiga new
  - ◆ 05 = Copylock ST
  - ◆ 06 = Speedlock Amiga
  - ◆ 07 = Old Speedlock Amiga
  - ◆ 08 = Adam Brierley Amiga
  - ◆ 09 = Adam Brierley, density key Amiga
- **formatId** (4 bytes): Selected Analyzer format ID
- **fix** (4 bytes): Analyzer Format Fix method
- **trackSize** (4 bytes): Analyzer track size
- **reserved** (2 bytes): Reserved for future used usually set to 0

The overall size of a CTEX record is 44 bytes (12 for the header block and 32 for the CTEX block).

## 4. Atari Example

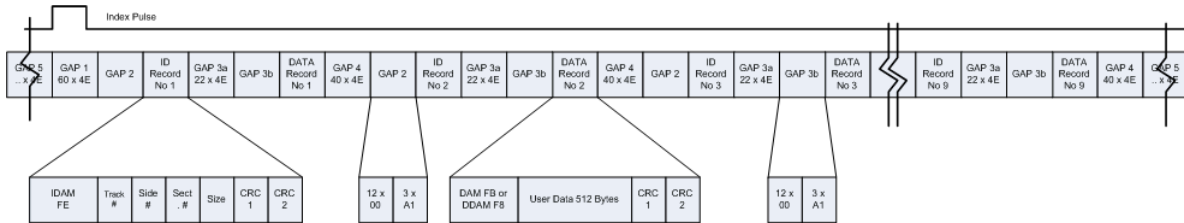
### 4.1 Atari FD Format short presentation

First let's quickly review the format used on the Atari machines. The Atari ST uses the Western Digital WD1772 Floppy Disc Controller (FDC) to access the 3 1/2 inch floppy diskettes.

#### 4.1.1 Atari Track Format

Below is a description of a **Standard Atari Double Density Format** as created by the TOS.

*Note:* Many different conventions have been used to decompose and name the different GAPS of a track. Here I use a GAP numbering scheme which is a combination of the IBM and ISO standards. The GAP between the ID record and the DATA record is decomposed into an ID **postamble** gap (Gap 3a) and a DATA **preamble** gap (Gap 3b). This allows a more detail description, but of course they can be recombined into a more standard intra-sector gap (Gap3).

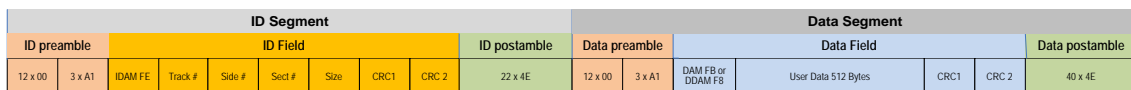


#### 4.1.2 Atari Sector Format

A sector is composed of an **ID Segment** that allow identifying the sector number, and a **Data Segment** that contains the actual data. These two segments contain at their ends specific gaps.

The ID and DATA preamble gaps are used to provide enough time to lock the PLL and are normally composed of 12 \$00 bytes. The ID postamble gaps provides enough time for the WD1772 to get prepared for execution of a read or write command on the matching Data segment and is typically composed of 22 \$4E bytes. The Data postamble gaps provides enough space between sectors for the WD1172 to get prepared to work on the next sector and it also provides room for *data segment drift* during write operation. It is typically composed of 40 \$4E bytes. The Atari FD format do not specifically define a sector write splice area (where the head write current is switched on or off) but in practice it should be considered as part of the data preamble field for format and write operations, and as part of the ID postamble for read operation (see below).

One complete sector looks like this



#### 4.1.3 Atari Sector write splices

It is not possible with the WD1772 FDC to write an entire track in one operation. This is due to the fact that when writing a track, the data in the range \$F5 to \$F7 are treated as special control bytes and therefore they cannot be written directly during a write track (format) operation. Therefore, on an Atari the first operation consists in formatting a track with a **format command** then the data field of each sector is written using **write sector commands**.

Here is a simplified description of what is happening during the write sector command:

Upon receipt of the Write Sector Command the FDC searches on the track an ID field that has the correct track number, correct sector number, and correct CRC. If such an ID field is found the WD1772 counts **22 bytes** from the end of the ID field CRC, then it activates the WG output.

# IPF – Interchangeable Preservation Format Documentation

The following data are then written to the disk:

- Twelve bytes of zeroes
- Three A1 Sync
- One Data Address Mark
- The complete data field (usually 512 bytes)
- A two-byte computed CRC
- One byte of logic ones.

Then the Write Gate (WG) output is deactivated.

It is easy to understand that the activation and deactivation of the Write Gate cannot be aligned precisely with the original data written during the format operation and furthermore, as the speed of the drive fluctuates, the overall size of the data segment will most likely not be exactly the same. This results in writing non-aligned (drifted) transitions, which most likely violates MFM rules, when the WG is activated **and** deactivated. These two areas are called *sector write splices*.

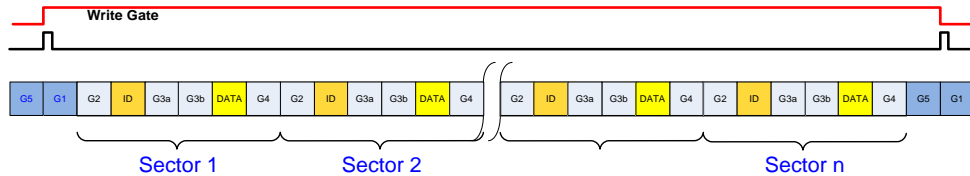
When data are written on a floppy disk with professional duplication equipment the complete track is written *in one operation* and therefore these sector write splice areas **do not exist**. This is one way for the Analyzer to verify if we are working on a “master floppy disk” written on professional duplication equipment and that the sector data on the FD has not been tempered.

## 4.1.4 Atari Track write splices

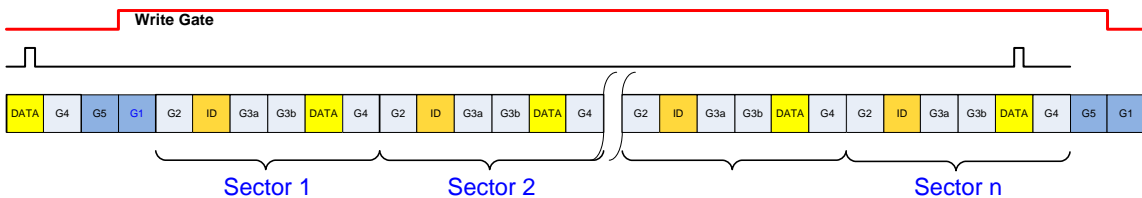
As said before when using professional duplication equipment, it is possible to write a complete track in one operation and this will result in a track without any sector write splices. The writing of the complete track starts at a particular point of the track and ends up at a particular point (typically the same point). For a non-protected track these starting (WG activation) and ending (WG deactivation) points are usually located at the track index.

But even with professional equipment it is not possible to activate and deactivate the write gate at **exactly** the same position. This results in writing a non-aligned (drifted) transition, which most likely violates MFM rules, when the WG is deactivated. This area is called the *track write splice*.

In the case of a standard track the writing starts in the Post-index gap G1 and stops in the Pre-index gap G5 and therefore the track splice happens in this non critical area.



However, some protections are based on shifting the position of the complete track in respect with the index. For example, the index might be positioned in the middle of the last data field (this protection is often referred as *data over the index*).



In such a case it is not possible to activate and deactivate the Write Gate at the position of the index because this would result in a track write splice located in the middle of the data field and therefore this will result in corrupted data.

For this kind of protection, it is therefore important for the duplication equipment to activate and deactivate the write gate in a position which is still located at the boundary of the pre-index and post-index gaps, knowing that this location is **not** aligned with the index.

## 4.2 Example of Theme Park Mystery Floppy

It is interesting to use a real Atari FD example to illustrate the information stored in a RAW stream file and the corresponding IPF file. For that matter we will take the Atari's game: **Theme Park Mystery**. We first analyze the Stream files of this game using my [KFAalyze](#) or [Aufit](#) programs. Then we will look at the content of the IPF file as produced by the **CAPS Analyzer** by using my [IPF File reader](#). For more information about protection refer to my [Atari copy protection](#) document.

### 4.2.1 Analysis of Track 00.0 in the Stream file

Let's first look at the [results of analysis](#) with the KFAalyze program of the flux transitions as provided by the Kryoflux board on track 00 head 0 of the game.

Several protections are detected by KFAalyze on this track:

- A No Flux Transition Area (NFA)
- Sectors with Fuzzy Data (FZD)
- Data Over Index (DOI)
- Sector within sector (SWS)
- Sector with Data CRC Error (DCE)

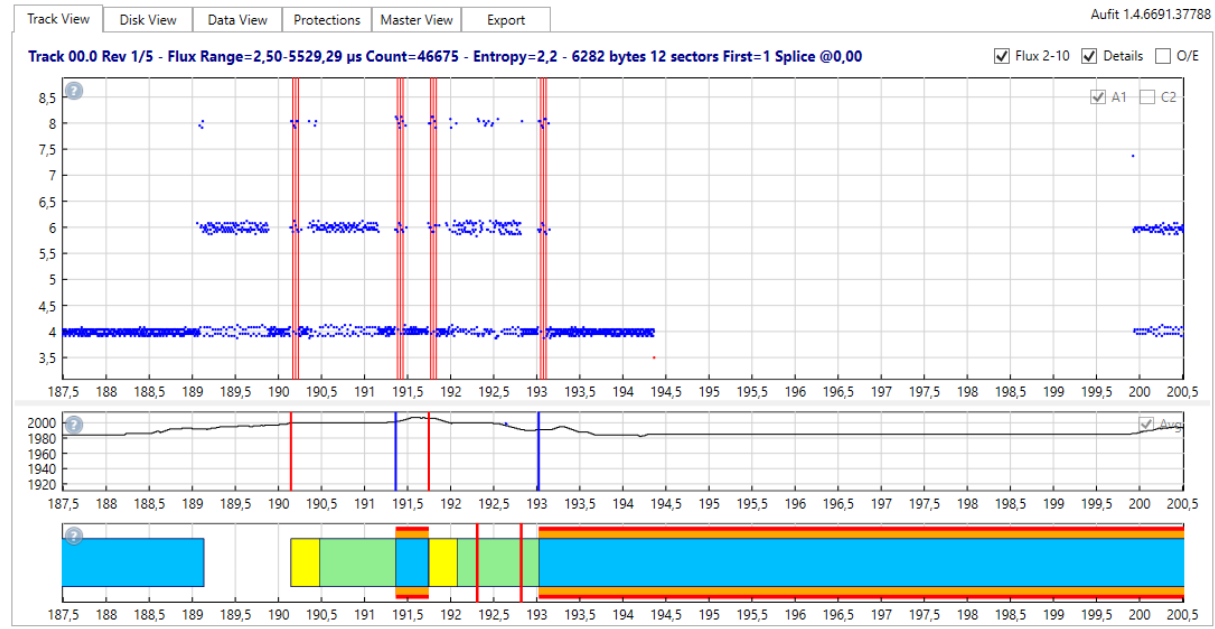
As well as timing violations, timing in ambiguous area, and sync errors.

The picture below shows the overall layout of the track as detected by KFAalyze:



We can see that the last two sectors are detected as continuing beyond the end of the track (DOI) that they overlap (SWS) and as they contain Fuzzy data (FZD displayed in orange). But we can also see that sector 1 starts at a normal position and therefore this indicates that sectors 11 & 12 are in fact specials short sectors.

In order to see the no flux transition area (NFA) we zoom toward the end of track 00.0.



Here we can see that around the position 194500µs till almost 200000µs (the end of track) there are **no flux transitions**. This NFA is directly located at the end of the track inside the sector 11 and 12. This alone explains why these two sectors read with fuzzy bits due to multiple timing errors.



# IPF – Interchangeable Preservation Format Documentation

Let's look at the IMGE record for track 00.0:

```
Record=IMGE Size=80 CRC=E35D87F2 Good - (108-187)
T00.0 Size=12557 bytes (100456 bits = Data=93056 + Gap=7400) Start Byte=60 Bit=482
DataKey=001 Block=12 Density=Auto Signal=cell_2us Encoder=0 Flags=None
```

The Track 00 Side 0 contains 12557 MFM (clock + data) bytes equivalent to 100456 bits that are decomposed as 93056 bits of Data and 7400 bits of Gap.

The track starts at position 60 bytes or more precisely 482 bits and contains 12 blocks. The DataKey for this track IMGE record is set to 1. The density (cell size) is adjusted automatically with the track size.

Now let's look at the matching DATA record (with the same DataKey=1 value).

```
Record=DATA Size=28 CRC=C07C042C Good - (13548-13575)
DataKey=001 Size=6652 bytes (53216 bits) CRC=53E54C0E Good [T00.0] - (13576-20227)
```

The DATA Block indicates that there is an **Extra data block** of 6652 bytes that contains: all [the block descriptors](#) as well as the Gap and Data stream information in the [Data Area](#). This extra block is "checked" with a second CRC.

Following the data block we have the block descriptors (12 in this case):

- 10 blocks describe standard Atari sectors
- 2 blocks describe the Copylock protection

## 4.2.2.1 Standard Block description

Let's look at the first one (block 0 at offset 0 in the extra data record):

```
Block=0 Data=8992 Off=515 Gap=512 Off=384 cell_2us MFM GapDef=0 Flags=FwGap, BwGap (13576-13607)
```

This block descriptor 0 indicates that the data area on the track is 8992 bits long in MFM equivalent to 562 bytes (8992/16) of data and has gap areas 512 bit long in MFM equivalent to 32 bytes (512/16).

Let's look at the Gap stream data elements. The flag in the block descriptor is equal to 3. This indicates that we have *forward and backward* gap streams. The forward gap stream is from the end of data of this block toward the data of the next block and the backward gap stream is from the beginning of this data block toward the end of the previous data block. We find the following information:

```
--- GAP --- 512 bits 32 bytes @13960
Forward W=1 Gap_Length 192 bits (24 bytes) @13962
Forward W=1 Sample_Length 8 bits (1 bytes) Value=4E @13965
Backward W=1 Gap_Length 64 bits (8 bytes) @13968
Backward W=1 Sample_Length 8 bits (1 bytes) Value=00 @13971
```

As we can see the forward gap (post-data gap) contains 192 bits (24 bytes) of 4E bytes and the backward gap (the pre-address gap) contains 64 bits (8 bytes) of 00 bytes.

We can see that the gap stream exactly matches the gap length:

$$512 = 192 * 2 + 64 * 2$$

Now let's look at the Data stream elements.

First it is interesting to note that the data element contains the complete sector description including the ID sync marks (3 x \$4489), the ID field, the intra sector gaps (22 x \$4E + 12 x \$00), the data sync marks (3 x \$4489) and the Data content:

```
--- DATA --- 8992 bits 562 bytes @14091
Sync W=1 (in bytes) 48 bits (6 bytes) @14093 44 89 44 89 44 89
Data W=1 (in bytes) 56 bits (7 bytes) @14101 FE 00 00 01 02 CA 6F
Gap W=1 (in bytes) 272 bits (34 bytes) @14110 4E 4E 4E 4E 4E 4E ... 00 00
Sync W=1 (in bytes) 48 bits (6 bytes) @14146 44 89 44 89 44 89
Data W=2 (in bytes) 4120 bits (515 bytes) @14155 FB 60 1E 00 00 00 00 ... B7 2C
```

We can see at the beginning of the sector the 3 sync bytes (\$A1) displayed as MFM encoded bytes \$4489 (\$A1 with missing clock bit):

```
+ Sync 3 bytes
0000 4489 4489 4489                               D.D.D.
```

## IPF – Interchangeable Preservation Format Documentation

This is followed by the sector ID field. We have the IAM \$FE followed by the Track, Side, Sector, Size and the two CRC bytes. We can see that this is **sector 01**

```
+ Data 7 bytes
0000 FE 00 00 01 02 CA 6F                p...Êo
```

This is followed by the intra-sector ID/Data gap: 22 x \$4E bytes + 12 x \$00 bytes

```
+ Gap 34 bytes
0000 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E  NNNNNNNNNNNNNNNNNNN
0016 4E 4E 4E 4E 4E 4E 00 00 00 00 00 00 00 00 00  NNNNNN.....
0032 00 00                                           ..
```

This is followed by 3 sync bytes (\$A1):

```
+ Sync 3 bytes
0000 4489 4489 4489                        D.D.D.
```

Followed by the actual data field: First the DAM (\$FB) followed by 512 bytes of data, followed by 2 bytes of CRC

```
+ Data 515 bytes
0000 FB 60 1E 00 00 00 00 00 58 00 00 00 02 02 01  û`. . . . . X . . . . .
0016 00 02 70 00 20 03 00 05 00 0A 00 01 00 00 00  ..p. . . . .
0032 00 20 3C 00 00 00 07 22 3C 00 00 00 03 20 79 00  . <. . . . " <. . . . y.

...
0176 EF 00 14 4A 80 4C DF 01 06 4E 75 00 00 00 00 00  i. .J.Lß..Nu. . . . .
0192 00 43 6F 70 79 6C 6F 63 6B 20 53 54 20 28 63 29  .Copylock ST (c)
0208 31 39 38 38 2D 39 30 20 52 6F 62 20 4E 6F 72 74  1988-90 Rob Nort
0224 68 65 6E 20 43 6F 6D 70 75 74 69 6E 67 2C 20 55  hen Computing, U
0240 2E 4B 2E 20 41 6C 6C 20 52 69 67 68 74 73 20 52  .K. All Rights R
0256 65 73 65 72 76 65 64 2E 00 00 00 00 00 00 00 00  eserved. . . . .

...
0496 00 00 00 00 00 00 00 00 00 00 00 00 00 00 AD  .. . . . . . . . . -
0512 0A B7 2C
```

It is interesting to note that in the IPF the ID field, inter-id-data gap field, and the Data field of a sector are kept together into one block. The included gap is where a potential “sector write splice” can occur and it would not be possible to describe it. However, as we are working on master FD created on professional duplicator machines there should not be any write splice in this area (otherwise this would indicate a tempered FD).

The same kind of information is provided for the sectors 2 to 10.

### 4.2.2.2 Copylock blocks description

The content of sectors 11 and 12 is quite different because these pseudo-sectors are not standard. They are part of the “Copylock short” protection.

The CTA block descriptor for the **Copylock short sgn** (sector 11) is:

```
Encode: 1, 1
_Area start: 1
  _Area start: 2
    Mark, byte: 3, $4489 *1 <esc>
    ID, byte: 1, $FE *1 <esc>
    Track, byte: 1 *1
    Side, byte: 1 *1
    Sector, byte f0: 1 *1 <esc>
    Length, byte: 1 *1
  _Area stop: 2
  EDC, CRC16: 2, $FFFF *2
  GAP, byte: 28 *1
  Mark, byte: 3, $4489 *1 <esc>
  ID, byte: 1, $FB *1 <esc>
  _Area start: 3
    Data, byte 8 *1
  _Area stop: 3
  EDC, MFM Encoding: 3, 0 *3
_Area stop: 1
```

We see that we have a “normal” ID field, followed by a 28 bytes gap, followed by a very short Data 8 bytes field not “protected” by a CRC field.





## IPF – Interchangeable Preservation Format Documentation

In the IPF file we find the following information for block descriptor 12

```
+ Block=11 Data=2336 Off=6382 Gap=2280 Off=504 cell_2us MFM GapDef=0 Flags=BwGap (13928-13959)
```

The gap data for this last sector contains the following information:

```
--- GAP --- 2280 bits 142 bytes @14080
Backward W=1 Gap_Length 192 bits (24 bytes) @14082
Backward W=1 Sample_Length 8 bits (1 bytes) Value=4E @14085
Backward W=1 Gap_Length 64 bits (8 bytes) @14087
Backward W=1 Sample_Length 8 bits (1 bytes) Value=00 @14090
Incomplete specification 512 bits out of 2280
```

There is no forward gap description. This indicates no gap following the last chunk of data.

The first backward gap description indicates that we first have 192 bits (24 bytes) of \$4E gap-bytes followed by 64 bits (8 bytes) of \$00 gap-bytes. As sector 12 is the last sector, the next sector is in fact the first sector of the track and all these bytes are therefore placed in front of this first sector.

The overall length of the gap is 2272 bit and we have only 512 bits specified. Therefore the decoder needs to loop the gap samples (4E) toward the beginning of the track.

The data stream elements are:

```
--- DATA --- 2336 bits 146 bytes @19958
Sync W=1 (in bytes) 48 bits (6 bytes) @19960 44 89 44 89 44 89
Data W=1 (in bytes) 56 bits (7 bytes) @19968 FE 00 00 0C 02 BC 33
Gap W=1 (in bytes) 64 bits (8 bytes) @19977 4E 4E 4E 4E 4E 4E 4E 4E
Data W=1 (in bytes) 64 bits (8 bytes) @19987 D9 23 76 C5 E6 D3 31 B2
Gap W=1 (in bytes) 64 bits (8 bytes) @19997 4E 4E 4E 4E 4E 4E 4E 4E
Data W=1 (in bytes) 48 bits (6 bytes) @20007 FF FF FF FF FF FE
Sync W=1 (in bytes) 1696 bits (212 bytes) @20015 89 12 89 12 89 12 AA 8A 55 55 55 55 ...
00 00 00 00 00 0000 00 00 00 ...00 00
```

We can see at the beginning of the sector the 3 sync bytes (\$A1) displayed as MFM encoded bytes \$4489 (A1 with missing clock bit):

```
+ Sync 3 bytes
0000 4489 4489 4489 D.D.D.
```

This is followed by the sector ID field. Here we have the IAM (\$FE) followed by the Track, Side, Sector, Size and the two CRC bytes. Here we see that this is **sector 12** (\$0c)

```
+ Data 7 bytes
0000 FE 00 00 0C 02 BC 33 p...¼3
```

This is followed by a short gap: 8 x \$4E bytes

```
+ Gap (Type=3) size=8 Bytes (@19655)
19657 4e 4e 4e 4e 4e 4e 4e NNNNNNNN
```

Followed by a very short data field of 8 bytes (the Copylock key?)

```
+ Data 8 bytes
0000 D9 23 76 C5 E6 D3 31 B2 Û#vÅæÓ1²
```

This is followed by a short gap: 8 x 4E bytes

```
+ Gap 8 bytes
0000 4E 4E 4E 4E 4E 4E 4E NNNNNNNN
```

Followed by a very short data field of 6 bytes: 5 x 255 (\$FF) and 1 x 254 (\$FE) as described in the Block descriptor.

```
+ Data 6 bytes
0000 FF FF FF FF FF FE ŷŷŷŷŷþ
```

## IPF – Interchangeable Preservation Format Documentation

Followed by an unusual 212 bytes' field of **Synch** data!

+ Sync 106 bytes										
0000	8912	8912	8912	AA8A	5555	5555	5555	5555	5555	..... <sup>a</sup> .UUUUUUUU
0016	5555	5555	5555	5555	5555	5555	5555	5555	5555	UUUUUUUUUUUUUUUU
0032	5555	5555	5555	5555	5555	5555	5555	5555	5555	UUUUUUUUUUUUUUUU
0048	5555	5555	5555	5555	5555	5555	5555	5555	5555	UUUUUUUUUUUUUUUU
0064	5555	5555	5555	5555	5555	5555	5555	5555	5555	UUUUUUUUUUUUUUUU
0080	5555	5555	0000	0000	0000	0000	0000	0000	0000	UUUU.....
0096	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0112	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0128	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0144	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0160	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0176	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0192	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
0208	0000	0000								.....

This field starts with 3 x \$8912 MFM sync marks. These \$8912 sync marks correspond to \$4489 sync marks shifted by a half clock. Following the MFM sync marks we have \$AA8A that correspond to the \$FB DAM shifted by a half clock. Following we have 38 x \$5555 MFM marks which are bytes \$00 when decoded. Following we have 64 x \$0000 MFM marks. Note that all these sync marks are directly expressed in MFM. Therefore, this indicates that there are no clock transitions written which defines a **No Flux Transition Area**.

## 5. CAPS/IPF Programing

---

This section is dedicated to programmers that develop application around the IPF format /or Library.

This section need to be developed ☺

### 5.1 IPF Decoder Library

The source of the CAPS/IPF library has now been [publicly released under license by SPS](#). This license is based on the MAME license and is intended for use in all non-commercial projects and environments.

The IPF 5.1 source tree released by SPS compiles without problem using [Microsoft Visual Studio Community 2017](#).

### 5.2 My IPF File reader / writer programs

[You will find on my site](#) several IPF reader / writer with the corresponding source code.

#### 5.2.1 IPFinfo Dos console program

This is a small console program that I have written in C++ to dump / check the content of an IPF file. This output of this program can be useful to understand the content as described in [IPF file content](#) chapter of this document. This program is based on an original code written in 2011 by Keir Fraser that I have extensively modified to make it more generic and to provide more information. This program is not supported anymore.

To run the program type:

```
IPFInfo [-i] [-d] [-n] input_ipf_file [output_file]
```

The -i and -d options can be used to add the dump of respectively the INFO and DATA records information in the output file. The -n option is used to filter the dump of unformatted track (noise track) in the output file.

The input\_ipf\_file is a .ipf or .ipx file as produced by CAPS Analyzer.

The output\_file is optional and by default the output goes to the console.

#### 5.2.2 IPF File reader / Writer

This is a small program with a nice GUI that I have written in C# to read and write the content of an IPF file. This can be useful to understand the content as described in [IPF file content](#) chapter of this document. This program has been written from scratch and is still under development as part of the public domain [AIR project](#).

## 6. Glossary

---

This section contains few useful definitions.

### Bit Cell

This is essentially the distance along the track allocated for the recording of an area of Flux. That is, in each “cell” there are magnetic particles that together indicate a detectable magnetic polarity of set length. Note that bit cells are not read as bits of data directly, they are used to form the flux transitions used to make up the Encoding. Every bit cell contains either a change of polarity (1) or not (0). Writing through the Atari FDC hardware is limited to bytes, and bit writes are clocked by the hardware clock to produce 4us wide bit “cells” for the MFM (Modified Frequency Modulation) Encoding format. For this reason, modifying the width of the bit cells written using Atari hardware is limited. This is one of the reasons why Long Tracks have to be copied using a hardware copier. Most games were mastered more professionally, with programmable duplicator systems - like from Trace. These were capable of writing any cell width desired.

### Bit Shifting

Also called peak shift. An effect that happens when reading from magnetic media. Essentially, boundary bits may be interpreted as part of adjacent data values. This means the value of the original element, and the value of the one next to it may change. Another way of putting it, would be to say the timing detected (and hence the data read) for a Flux Transition can change, caused by the influence of a neighboring flux transition. The IPF (Interchangeable Preservation Format) decoder (library) generates bit shifting into the decoded stream automatically at the most likely position it would occur if the data was being written to a real disk for absolutely faithful reproduction of the real media properties. This helps emulation of certain protections, without any kludges or any extra information required to distinguish between data meant for mastering and data meant for actual use (emulation).

### Block (aka Sector)

Each Track is made up of one or more blocks. The PC/ Atari ST format usually uses 9 blocks but “generic hardware MFM” formats use whatever number of blocks and sizes as programmed into the controller. Blocks are used to reduce the amount of data stored and encoded in one go, thus modifying just a small amount of data does not require a whole track to be modified, and errors on a block does not render the rest of the data unreadable. Blocks are normally separated by gaps.

### Copy Protection

Copy protection was used on media for various systems to protect the investments of the people involved in creating the software content. Various techniques were used to this end. Some of the common copy protection techniques are:

- ◆ Fuzzy Bits: random data is read, which becomes non-random when copied.
  - ◆ Long/Short tracks: the data is packed on a track, more and less dense respectively. This needs to be checked by the program code.
  - ◆ Density timing. Various patterns of variable density are used across a track and checked to exist.
- The physical properties of magnetic media can be used in many ways to hinder copying of the data stored on it. For magnetic media, it is difficult to read something without knowing how it is structured, and it is impossible to reliably write something without knowing how to write it. This “how” varies between publishers and game developers, but they used a wide variety of techniques. The whole emphasis in Copy Protection is to write the data in such a way that it can be read, but cannot be written using consumer products, and hence cannot normally be reproduced.

### Custom Disk Format

Floppy disk based systems organize data on the media in different ways. Some examples of possible variations are:

- ◆ Physical Encoding
- ◆ Number of blocks
- ◆ Data/value representation
- ◆ Header information (current Track number, head number, data size, etc.)

## IPF – Interchangeable Preservation Format Documentation

- ◆ Location and method of integrity information
- ◆ Location and value of the Sync (aka Mark)
- ◆ Location, value and size of Gap
- ◆ Location and size of actual data part

This is simply just layout (aka geometry) information. Computer designers would either create their own format, or they would use an existing one depending on their system's hardware. If they used a fully-fledged FDC (Floppy Disk Controller), the format had to conform to predefined rules of geometry as allowed by that hardware. One reason to use custom format was to prevent the disk from being copied in a normal way.

### Density

How closely bit cells are packed together on a section of track. This is effectively determined by the bit cell width. Higher density (using bitcells of shorter width) means more data can be packed into the same length of track. Commercial duplicator machines were able to write bit cells of different lengths. For protections varying density was used across a track (e.g. Copylock, Speedlock. The Atari can read disks of varying density with a limit of roughly +/- 10% of normal density.

### Encoding

Data encoding is the process of converting binary information (programs and data) into magnetic impulses that can be stored on the magnetic surface of the disk. There are several different ways that this can be done. Floppy disks generally use the MFM (Modified Frequency Modulation) encoding technique, which is also used on old hard disks.

Due to the nature of magnetic recording, you cannot just store a meaningful sequence of bits in the same number of bit cells, since you would not be able to read it. Instead, a bit is stored by the presence, or lack of, a Flux Transition. That is, the change of polarity between two bitcells. This is further complicated because you cannot (reliably) read this data unless you use some form of clocking so you know when a flux transition has occurred (a binary 1) or could have occurred but did not (a binary 0).

### FDC Emulator

An FDC emulator job is to implement a definitive, low level, and most importantly accurate emulation for various brands of generic hardware floppy controllers. Almost of these “off the shelf” controllers are derivatives of the NEC765A FDC. The FDC emulator acts as a “plugin”, or “library” for emulators to allow inclusion of true floppy system emulation without needing to re-implement this complicated piece of functionality themselves.

### Gap

This is a data area that can be found either side of a Block (aka Sector) or Track, though it does not indicate the separation like a Sync (aka Mark). The size of a gap area may be altered via mastering if allowed by the format. Any errors found after the gap can normally be ignored, though a few protections have been found that store data in this area.

Gaps are normally used to allow re-writing of specific blocks on a track. As this operation is not timing accurate, the gap provides the threshold needed to cover for the inaccuracies that occur in the timing of the write.

There is at least one gap between the last block on one track, and the first block of the next. Its purpose is to compensate for the different properties of a track written on specific locations on the disk as well differences between drive speed alignment. If the drive speed is different within reason, this gap will have some of its content overlapping i.e., overwritten when mastering a whole track.

### Index Pulse

Indicates the hardwired beginning of a Track (actually the full rotation cycle of the disk), that can be used as a fixed reference point by software or hardware.

### Jitter:

This is the difference between the expected time of arrival of some data, and the real time of arrival of that data. You would be tempted to think of this as tolerance, but it isn't. Tolerance means to compensate for jitter, or “de-jitter”. Most FDC's incorporate some kind of jitter hardware so that they

## IPF – Interchangeable Preservation Format Documentation

are tolerant within some boundaries if the Bit Cell widths are not quite the ones expected within a given amount of time, i.e. the clock speed selected for cell width. This is why it is possible for the FDC to read bit cells that are not exactly the same width as expected - and indeed they are not usually the width expected, due to the nature of magnetic recording.

This fact was well used by Copy Protection designers who made the bit cells different widths on purpose, something that could only be reproduced by special mastering hardware, and due to the tolerance built into the FDC's, the data could be read properly, it was just impossible to write by the user. This hardware is built into most FDC's, and what you normally read using the FDC is not actually the same as the data as it is present on the disk surface. This is a very important fact to bear in mind when preserving magnetic media! *For preservation, you need to insure that you get 1:1 what is on the disk surface, NOT what the FDC interprets it has.*

### Long Track

One of the simplest and most effective Floppy Disk Copy Protection schemes. This is where the disk mastering hardware reduces the size of the bit cells, which allows more data to be written on any one track. This is something which cannot be done on an Atari platform. An alternate way of producing a long track is to slow down the disk drive which in practice it has the same effect - the bit cells take less space, so there can be more cells written to the same area. Writing short tracks uses exactly the same techniques, but obviously using a wider bit cell width rather than shorter.

### MFM (Modified Frequency Modulation)

Modified Frequency Modulation, also known as Delay Modulation or Miller Code is a self-clocking Encoding used on floppy disks of many systems including the Commodore Amiga, Atari ST and PC. It was an enhancement over FM encoding, storing roughly double the amount of "meaningful" data using the same amount of bits due its more efficient clocking, which worked by considering not just the current bit, but also the previous one. This is why disks are marked as DD or 2D - "Double Density" since they are MFM not FM.

### Sync (aka Mark)

Special marker used to separate different areas of data. The sync is (in theory) not produced by the Encoding used on a data area. Therefore it can mark the beginning (sometimes not really the beginning) of a new physical/logical data area, known as a Block (aka Sector). The PC and the Atari format both use MFM "0x4489" as the sync pattern. In the analyzer it is referred to as the "Mark" to be more in line with Trace conventions. It is treated as data, but the encoding can be modified, but it is used by the analyzer to spot areas on a track in the first place.

### Track:

A "ring" around one side of a disk containing data. Tracks begin and end at the Index Pulse.

## 7. References

---

- [CAPS Library and API Documentation Revision 1.02](#)
- SPS WIP: [Time for an even more generic way to describe disk formats...](#)
- SPS WIP: [Analyzer: Generic MFM Support - Gap Analyzer](#)
- SPS WIP: [Analyzer: Generic MFM Support - Gap](#)
- [CAPS Source Code Release 4.2](#)
- [All you always wanted to know about IPF](#)
- [Amiga Disk Utilities & IPF creation and disk writing via Kryoflux](#) by Keir Fraser
- [Structure of descriptors and data for a track](#) by Keir Fraser
- [Atari Protection based on Key Disk](#) by Jean Louis-Guerin
- [KFAalyze Program](#) and [documentation](#)
- Microsoft Walkthrough: [Creating and Using a Dynamic Link Library](#)
- Microsoft Walkthrough: [Creating and Using a Static Library \(C++\)](#)

## 8. History

---

- V1.6 April 2018 – Miscellaneous correction and fixed broken links. Initial release to SPS.
- V1.5 January 2016 – Added linkage information about the **GapBits** and **DataBits** fields in [IMGE record](#) and [block descriptors](#). Corrected the info in section [3.7.2.2](#) about not having **dataSample** record in case of fuzzy bits.
- V1.4 July 2015 – Fixed several minor errors while writing new code for [AIR-IPF reader Writer](#). Especially errors in GAP and DATA stream Elements.
- V1.3 March 2015: Feedback about Atari tests. IPF/IPX file content has been widely rewritten. Atari example clarified with example using AUFIT. CAPS/IPF programming updated to 5.1 and new updated reader/writer section.
- V1.2 September 2013: Fixed errors, added more references to SPS and Kryoflux forum.
- V1.1 June 2013: Added information on IPX format and CTEI/CTEX blocks. Fixed errors.
- V1.0 June 2013: First official release. Fix information based on feedback. Added the Copylock protection description in [Analysis of Theme Park Mystery](#). TODO: Some record types are not described (DUMP, TRCK, CTEX, CTEI) but they do not seem to be used in standard IPF files. The CAPS/IPF Programming section needs to be completed / reworked.
- V0.0 January 2012: Working review version for feedback