

- Pasti File Documentation -

Jean Louis-Guérin (DrCoolZic)
Version 0.5 January 2014

Pasti File Documentation

Table of Content

Chapter 1.	About this Document.....	2
Chapter 2.	Pasti file content	3
2.1	File Descriptor.....	3
2.2	Track records.....	4
2.2.1	Track descriptor	4
2.2.2	Optional Sector descriptors	5
2.2.3	Optional Fuzzy Mask record	6
2.2.4	Track Data record	6
2.2.5	Optional Timing record	7
References	9
History	9


Chapter 1. About this Document

First it is important to understand how Atari diskettes were manufactured. 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.

Pasti is a package of software tools for imaging and preservation of Atari software. The two major components are the imaging tools and the emulation helper tools (used by Steem emulator).

The imaging tools produce a disk image from an original disk known as a Pasti STX file (file with a .stx extension). They works very similarly to other imaging tools like Makedisk, but they can image virtually any ST disk including **copy protected disks**.

This document describes the Pasti .stx file format produced by these imaging tools.

 **Note:** The author (**Ijor**) has not published the Pasti format. Therefore this description is mainly based on reverse engineering performed by **Markus Fritze**, **P. Putnik**, and myself (see references at the end of the document). **Ijor** has also kindly replied to several of my questions. However beware that this description of the Pasti format might contain errors.

Chapter 2. Pasti file content

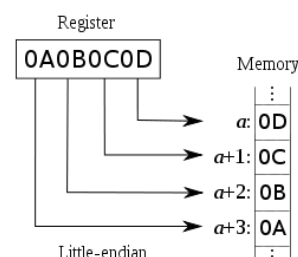
This section provides information about the content of a **Pasti** file as it would be read for example with a hexadecimal editor. This information can be useful to decode or create Pasti files.

The overall Pasti file organization is composed of a **File Descriptor** record followed by **Track Records**:

File Descriptor **Track Records**

Note that unless specifically indicated the information in the file is stored using **little-endian** ordering. This applies to two or four bytes fields. Therefore, in most cases, on an **Intel** platform you can use the values read from the file directly.

When the format was designed for the first time, it was not possible to anticipate all the issues that would eventually arise. So for historical reasons everything is not as clean as it would be possible to design now, with the current knowledge. Other oddities result from bugs in some versions of the imaging tool and/or in some versions of the library. And finally some of the fields on the headers were reserved for future expansion but were never used.



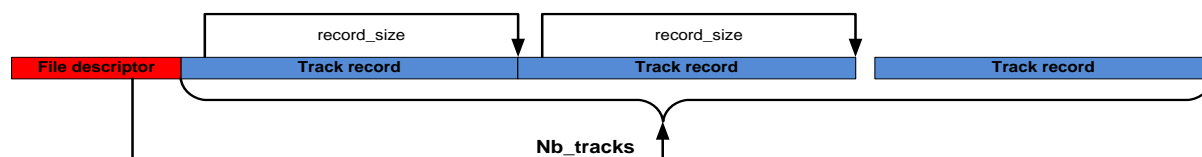
2.1 File Descriptor

The first record in a Pasti file is the [File Descriptor](#). It is 16 bytes long and can be described with the following C Structure:

```
typedef struct file_desc_ {
    char    pastiFileId[4];           // File Identifier "RSY\0"
    uint16_t version;                // File version number
    uint16_t tool;                   // Tool used to create image
    uint16_t reserved_1;             // reserved 1
    uint08_t trackCount;             // Number of track records following
    uint08_t revision;               // File revision number
    uint32_t reserved_2;             // reserved 2
} FileDesc;
```

- **pastiFileId** (4 characters): Identify a Pasti file and should be equal to string "RSY\0".
- **version** (2 bytes): Should be equal to 3. This should always be checked as we do not know anything about the format of Pasti file in version 1 and 2.
- **tool** (2 bytes): Indicate the imaging tool used to create the Pasti file. Current known value are:
 - ◆ 0x01 for file generated with the Atari imaging tool, and
 - ◆ 0xCC for file generated with the Discovery Cartridge (DC) imaging tool
- **reserved_1** (2 bytes): not used. Its value is usually set to zero.
- **trackCount** (1 byte): indicates the number of tracks (usually 80 or 160 for double face FD). This field also indicates the number of [Track records](#) following the file descriptor record.
- **revision** (1 byte): revision number of the file. Works in conjunction with version. Known values:
 - ◆ 0x00 Old Pasti file format or
 - ◆ 0x02 New Pasti file format (usage of this field is explained later in [Sector descriptor](#))
- **reserved_2** (4 bytes): Not used and usually set to 0x0000¹.

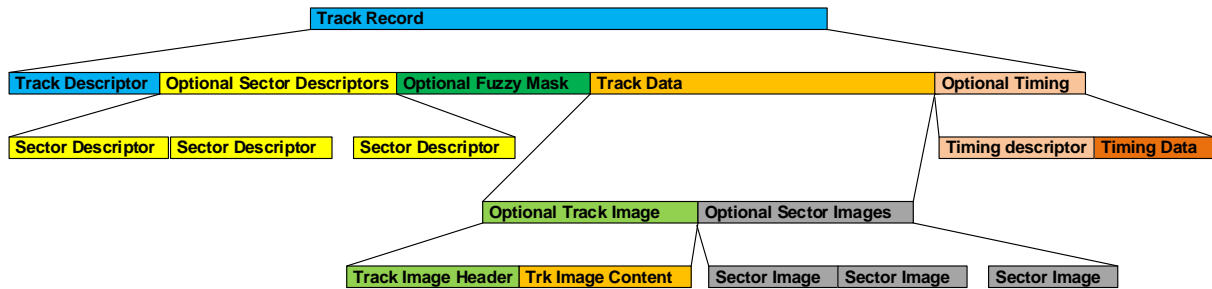
In summary you should check that **pastiFileId** is equal to string "RSY\0", that **version** is equal to 03 and store information about **trackCount** and **revision**.



¹ With the exception of few early DC generated file where the value was accidentally set to 0xCCCC

2.2 Track records

Pasti tools store information about each track imaged in a **track record**. The number of Track record is specified in the **trackCount** field of the [File descriptor](#). It contains the following records (some are optional):



2.2.1 Track descriptor

The first structure of a [Track record](#) is always a [Track descriptor](#). A [Track descriptor](#) is 16 bytes long. It can be described with the following C structure:

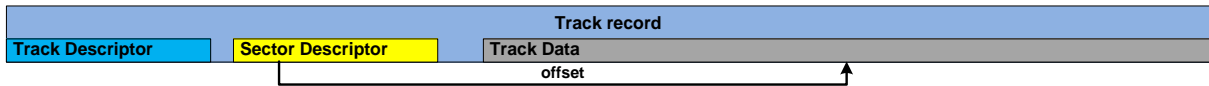
```
typedef struct track_desc_ {
    uint32_t recordSize;           // Track record size
    uint32_t fuzzyCount;          // number of bytes in fuzzy mask record
    uint16_t sectorCount;         // number of sector in track
    uint16_t trackFlags;          // bitmask info for track record
    uint16_t trackLength;         // Total bytes (Bit rate)
    uint08_t trackNumber;         // track number (coded for both side)
    uint08_t trackType;           // track image type
} TrackDesc;
```

- **recordSize** (4 bytes): This field defines the overall length of the current [Track record](#). Therefore the position of the next [Track descriptor](#) (if any) in the file is equal to the position of the current [Track descriptor](#) plus **recordSize**.
- **fuzzyCount** (4 bytes): This field indicates the size in bytes of the optional [Fuzzy Mask record](#) in the [Track record](#). Sectors with fuzzy bytes have the fuzzy bit indicator set (described by the bit 7 of the **fdcFlags** in the [Sector descriptor](#)). The bytes of the fuzzy mask record may need to be “dispatched” to the different sectors in the track that have the fuzzy bit indicator set. For example we can have a [Fuzzy mask record](#) of 1024 bytes that need to be dispatched between sector 1 and 2 each having 512 bytes.
- **sectorCount** (2 bytes): This field indicates the number of sectors of the track. If bits 0 of **trackFlags** (described below) is set this also indicates the number of [Sector descriptors](#) in the [Track record](#). A value of 0 indicate an empty or unformatted track.
- **trackFlags** (2 bytes): This field contain bitmask flags that provide information about the content of the track record as follow:
 - ◆ Bit 8-15: Not used and set to 0
 - ◆ Bit 7: This bit is set to indicate that the [Track image](#) record contain a four bytes Track Image header, otherwise the Track image header is only two bytes. **This bit can only set if bit 6 is set.**
 - ◆ Bit 6: This bit is set to indicate that the track record contains an optional [Track image](#).
 - ◆ Bit 1-5: not used
 - ◆ Bit 0: When this bit is set the track descriptor is followed by **sectorCount** [Sector descriptors](#). When not set the track record only contains [Sector image](#) following immediately this track descriptor. This can only be used for standard 512 bytes sectors with standard sector numbering from 1 to n. In this case the size of the track record is equal to 16 + (**sectorCount** * 512).
- **trackLength** (2 bytes): length of the track in number of bytes. Usually around 6250 bytes.
- **trackNumber** (1 bytes): Bit 7 contains the side (0 or 1) and bits 6-0 contain the track number (usually 0 to 79).
- **trackType** (1 byte): Not used

The trackFlags descriptors:

```
// Track descriptor flags
const uint08_t TRK_SYNC = 0x80; // track image header contains sync offset info
const uint08_t TRK_IMAGE = 0x40; // track record contains track image
const uint08_t TRK_PROT = 0x20; // track contains protections ? not used?
const uint08_t TRK_SECT = 0x01; // track record contains sector descriptor
```

2.2.2 Optional Sector descriptors



Following the [Track descriptor](#) we find an **optional** set of records used to provide additional information about all the sectors of a track. These descriptors are only present if bit 0 of the [track descriptor TrackFlags](#) is set. The number of sector descriptors is equal to the **sectorCount** field in the [track descriptor](#). Each Sector Descriptor is 16 bytes long and can be described by the following C structure:

```
typedef struct sector_descriptor_ {
    uint32_t dataOffset;           // offset of sector data in the track data record
    uint16_t bitPosition;         // Position in bits of the sector from start of track
    uint16_t readTime;           // sector read time in ms
    Address id;                   // Copy of the id field of a sector (6 bytes)
    uint08_t fdcFlags;           // fdc status and flags
    uint08_t reserved;           // Not used Always 00
} SectorDesc;
```

Where Address is the following structure:

```
typedef struct _address_ {
    uint08_t track;              // Track number from Address block
    uint08_t head;              // Head (side) number from address block
    uint08_t number;            // Sector number from address block
    uint08_t size;              // Size number from address block
    uint16_t crc;               // CRC Value from address block
} Address;
```

- **dataOffset** (4 bytes): This is the offset to access the data for this sector inside the track data. The [Track Data](#) record is located just after the optional [Sector descriptor](#) and [Fuzzy Mask record](#). Therefore we have `sector_data_position = track_data_position + dataOffset`. This can point either inside a [Track image](#) or to a [sector image](#).
- **bitPosition** (2 bytes): This field store the position in number of **data** bits (not MFM bits) of this sector address block in reference to the index pulse. If you multiply this value by 4 you get roughly the time it takes in μs to reach the sector address block.
- **readTime** (2 bytes): This field contains either zero or the read time for the sector data block.
 - ◆ If the value measured by the imaging tool is within 2% of the standard sector read time ($16384 \mu s = 512 * 32 \mu s$) the value written is 0 to indicate a sector with "standard" timing.
 - ◆ Otherwise the measured value in μs is directly stored.
- **id** (6 bytes): this field contains a copy of the content of the id field for this sector:
 - ◆ **track** (1 byte): the track number
 - ◆ **head** (1 byte): the head/side number (usually 0 or 1)
 - ◆ **number** (1 byte): the sector number (usually 1 to 9)
 - ◆ **size** (1 byte): the encoded size of the following sector data block. The actual value is equal to 128 shifted left by the provided value. Normally 0 to 3 (128 to 1024 bytes).
 - ◆ **CRC** (2 bytes): the value of the two CRC bytes using the CCITT CRC16 polynomial.
- **fdcFlags** (1 byte): This field contains a mixture of the FDC status, as it would have been read by the WD1772, and other flags used to interpret the track record content.
 - ◆ Bit 7: When set the sector contains fuzzy bits described by a [Fuzzy Mask record](#).
 - ◆ Bit 6: not used
 - ◆ Bit 5: FDC Record Type (1 = deleted data, 0 = normal data)
 - ◆ Bit 4: FDC RNF (record not found). When set the sector contains only an address block but no associated data block. In that case there is no data information associated to this descriptor.
 - ◆ Bit 3: FDC CRC error. If RNF=0 it indicates a CRC error in the data field if RNF=1 it indicates a CRC error in address field.
 - ◆ Bit 1-2: not used
 - ◆ Bit 0: Intra-sector bit width variation. This flag is used to indicate a protection based on bit width variation inside a sector. Currently the only known protection of this type is MacroDOS/Speedlock
 - If the *revision number* of the [File descriptor](#) is 0 the macrodos/speedlock protection is assumed. In that case an [internal table](#) needs to be used to simulate the bit width variation.
 - If the revision number in the [File descriptor](#) is 2 the track record contains an extra [Timing record](#) following the [Track Data](#) record. This allow a more precise description of the bit width variations and is open to other kind of intra-sector bit width variation protection if discovered. Note that with revision 2 the [Timing Record](#) are not limited to the MacroDOS/Speedlock but can be used for any intra sector bit width variation.
- **reserved** (1 byte): not used usually 0x00

2.2.3 Optional Fuzzy Mask record

The Fuzzy Mask record is used to store the mask bytes to be used when a sector contains fuzzy bytes (in this case bit 7 of the **fdcFlags** field is set in the [sector descriptor](#)). The size of the fuzzy mask record is specified in the **fuzzyCount** field of the [Track descriptor](#). It is also equal to the sum of the all the sector sizes that contains fuzzy bytes. For example if a track contains two 512 bytes sectors with fuzzy bits the fuzzy mask record size will be 1024. The mask record bytes are then “dispatched” to the different sectors.

To find fuzzy bytes the imaging program read the same sector multiple times. If the content of a byte in the same position is different the mask value is the result of a **xor** operation complemented.

```
Mask[offset] = ~(Sector_read_1[offset] ^ Sector_read_2[offset])
```

Thus same value (1&1 or 0&0) gives a result of one and different value (0&1 or 1&0) gives a result of zero. For example if the same byte is read one time as 0000 1111 and the next time as 0101 0101 the mask will be 1010 0101.

The mask can then be used to emulate fuzzy bits as follow:

```
FuzzyByte[offset] = (Sector[offset] & Mask[offset]) | (rand() & ~Mask[offset])
```

Where rand() is a function that randomly returns true or false.

Not that usually the first 32 and last 32 bytes of a sector are not fuzzy (they read the same).

2.2.4 Track Data record

2.2.4.1 Optional Track Image

If bit 6 of the flags field is set in the [Track descriptor](#) this indicates that the track record contains a Track Image record. The Track Image record is composed of a **Track Image Header** followed by a Track Image Content record. Typically the content of the Track Image Data record is what would be read by the WD1772 using a **read track** command.

Track Image Header	Track Image Content
--------------------	---------------------

2.2.4.1.1 Track Data Image Header


The Track Image Header size depends of bit 7 of the **trackFlags** field in the [Track descriptor](#).

- If bit 7 is set then the track header contains:
 - ◆ **FirstSyncOffset** (two bytes): This is the offset in byte of the first 0xA1 sync byte found in the track. This is usually the synch byte in front of the first address field.
 - ◆ **TrackImageSize** (two bytes): the size of the track image data record that follows the header.
- If bit 7 is not set the track image header only contains:
 - ◆ **TrackImageSize** (two bytes): the size of the track image data record that follows the header.

2.2.4.1.2 Track Data Content

The actual Track Image Data record follow directly the Track Image Header.

This record only exists when a protected track is detected.

 In most cases the **TrackImageSize** value is equal to the **trackLength** of the Track Descriptor. However in some cases the two values can differ. For this reason always use **TrackImageSize** to read the track data. Note also that if you need to write a Pasti file the Track Data seems can be padded with an extra byte to always end up on an even boundary.

2.2.4.2 Optional Sector images

These records are used to store sector data information.

- If there is no [Track Image record](#) (bit 6 of the **trackFlags** not set in the [Track Descriptor record](#)) all the sector images are added one after the other.
- If a [Track image record](#) exists the imaging tools optimizes the size of the file by only adding the Sectors Image records if necessary. If the data for a sector in the **Track Image Data** exactly matches the actual **Sector Data**, then the data in the Track Image is used. In this case the **dataOffset** of the [Sector Descriptor](#) points inside the Track Image Data. Otherwise (if doesn't match) a Sector Image is written for this sector and the **dataOffset** of the [Sector Descriptor](#) points to this Sector Image. Note: Sector Data and Track Data can differ because the data may be shifted whenever a sync mark pattern is found while reading the sector using a **read track** command.

Note that some sectors **may be missing** if no data has been found for this sector. This is indicated by Bit 4 (RNF) of the **fdcFlags** in the [Sector Descriptor](#).

2.2.5 Optional Timing record

When the imaging program detect that variable bit rate inside a specific sector contains several bytes above **and** several bytes below a certain bit rate threshold (usually 2%) then an **intra-sector** variable bit rate timing protection flag is raised.

- With **revision 0** (as defined in the [File Descriptor](#)) only macrodos / speedlock intra-sector variable bit rate were detected and flagged. In this case the bit 0 of the **fdcFlags** field in the [Sector descriptor](#) is used to indicate a macrodos / speedlock protection but no timing information is stored. In this a case the emulation program needs to use internal tables to return variable intra-sector variable bit rate values.
- With **revision 2** the imaging program can potentially detect any kind of intra-sector variable bit rate protection. In this case the bit 0 of the **fdcFlags** is set and a [Timing record](#) is recorded. This optional **Timing record** is located just after the [Track Data](#) record. It is not always clear on when imaging tool decide to write timing records. This information is complementary to [Sector Descriptor readTime](#).

The Timing record is composed of a 4 bytes Timing descriptor followed by a Timing Data record.

Timing descriptor	Timing Data
-------------------	-------------

2.2.5.1 Timing descriptor

The Timing descriptor has 2 fields:

- Flags (2 bytes): Always 5 ?
- Size (2 bytes): The total size of the timing record including the header. The size of the timing data is therefore (size - 4). These timing data need eventually to be dispatched to different sectors.

2.2.5.2 Timing Data record

The Timing Data record contains the actual bit rate information. Each timing entry is 2 bytes long and corresponds to the time it takes to read a 16 bytes block (this is the best precision we can get on an Atari machine). For a 512 bytes sector the Timing Data record will contains 32 two bytes values (512 / 16).

 The timing values are stored in two bytes using **big-endian** notation. This is the only place where big-endian storage is used in Pasti file.

Each timing value is expressed by a number of 4 μ s ticks and is equal to the time necessary to transfer 16 bytes through the DMA to the FDC.

The nominal value is 128 (0x007F) ticks equivalent to 512 μ s (= 16 bytes * 32 μ s per byte).

The timing values may need to be "dispatched" to the different sectors in the track that have the timing bit indicator set (described by the bit 0 of the **fdc_flag** in the [Sector descriptor](#)). For example we can have 64 values that need to be dispatched between sector 1 and 2 of 512 bytes each having 32 values.

2.2.5.3 Timing Table for revision 0

With revision 0 only macrodos / speedlock variable bit width protection is supported. In this case as no Timing Data record is provided you can use the following values for the timing:

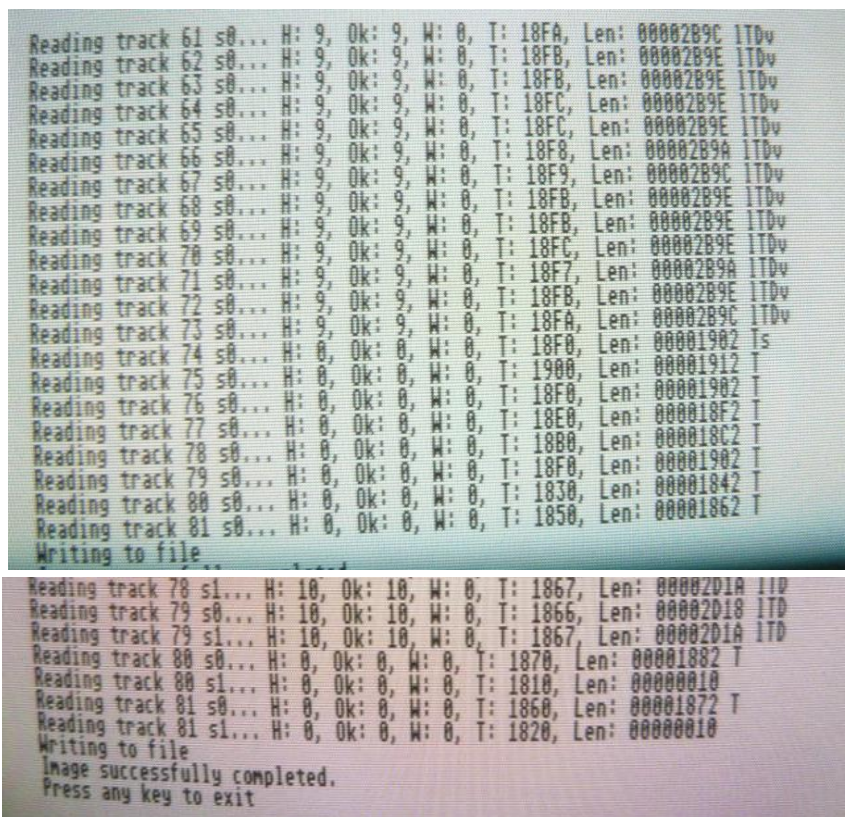
- The timing for the bytes in the first quarter of the sector should use value = 127
- The timing for the bytes in the second quarter of the sector should use value = 133
- The timing for the bytes in the third quarter of the sector should use value = 121
- The timing for the bytes in the fourth quarter of the sector should use value = 127

Chapter 3. Imaging Tool

Some information about the Pasti Imaging tool.

If you ask for verbose output Pasti imaging display a line like this

```
Reading track 0 s0 ... H:10 Ok:10 W:0 T: 1752 LEN: 000029F4 ITDvs
```



So we have the track number, the side then

- H: number of sectors in the track
- Ok: number of sector that read Ok
- W:
- T: number of bytes of the track in hexadecimal
- LEN: length of the track data in hexadecimal
- Indicators:
 - ◆ I track has sector ?
 - ◆ D track read directly. If no D then program had to retry (nothing apparent in STX file)
 - ◆ T track image required
 - ◆ v track contains bytes with bit width variation ?
 - ◆ s
 - ◆ Nothing = no track data, no sector data.

References

- [Pasti Atari ST Imaging & Preservation Tools](#)
- [Pasti File Format](#) by Markus Fritze
- [Pasti STX floppy image format](#) by P. Putnik

History

- V0.5 January 2014: Added information about track data alignment, added [Timing Table](#) information for revision 0 of the file. Corrected several errors.
- V0.4 January 2014: Fixed timing information and minor fix.
- V0.2 December 2013: Fixed few mistakes (first release to public domain). Still need validation.
- V0.1 June 2013: Updated based on feedback from Ijor.
- V0.0 January 2012: initial review version