

# **finefiles .timeline and Sidecar Specification**

**Revision 01.05.00**

**Christian Schnettelker**

**March 2012**

[www.finefiles.com](http://www.finefiles.com) [mail@finefiles.com](mailto:mail@finefiles.com)

# 1. Conventions

---

## 1.1 Data types

The following basic data types are used in this document:

BYTE	8 bit value or character
CHAR	8 bit character
UCHAR	8 bit unsigned character
INT8	8 bit signed integer value
WORD	16 bit unsigned integer value
UINT32	32 bit unsigned integer
INT64	64 bit signed integer

## 1.2 Used field types and their sizes

field- and variable names in this format specification are named in the [Hungarian notation](#): 'i' for signed and 'u' for unsigned integer, 'c' for char, 'w' for word, 'x' for a unspecified type or a structure. "i64Value" means a 64 bit integer value, "i8Value" means a 8 bit integer value, "iValue" without any number means a 32 bit integer value. The NULL value means a null pointer indicating that it refers to no object, the value is zero. Values written with a leading 0x means hexadecimal values.

## 1.3 PIT sub-structure

The PIT (Point in Time) structure reduces a date and time value to a short and handy format. This definition used for both .timeline and Sidecar files. The PIT structure has a total size of 8 bytes.

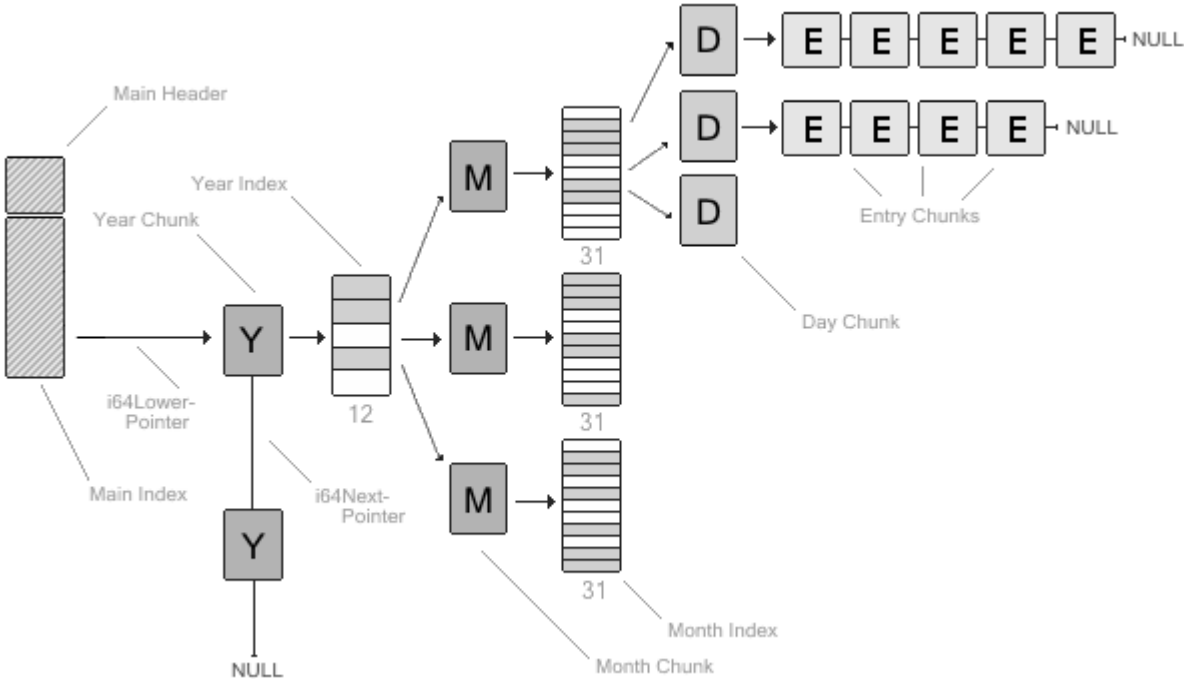
Length	Name	Description
WORD	wYear	The year. Can be 0 if the year is unknown.
INT8	i8Month	The month. January = 1, February = 2 etc. Can be 0 if the month is unknown. Highest possible value is 12 for December
INT8	i8Day	The day of the month. Can be 0 if the day is unknown. Highest possible value is 31.
INT8	i8DayOfWeek	The day of the week, Sunday = 0, Monday = 1 etc. Highest possible value is 6 for Saturday
INT8	i8Hour	The hour. The valid values for this member are 0 through 23
INT8	i8Min	The minute. The valid values for this member are 0 through 59
INT8	i8Sec	The second. The valid values for this member are 0 through 59

## 2. Timeline file format specification

Here are the file specification of .timeline files which finefiles creates to realize the timeline feature. The specifications may be useful if you would like to write your own routines to read out the data they contains.

### 2.1 The year trees

Basically there are two ways to take access to the data: First you can follow the "year trees" - the i64Lower value in the Index header at file start points to the first year in a queue. Or, second, you can read out every chunk serially if the search doesn't depend on the date structure, if you have to access every chunk or if the tree index is corrupted. In this case the chunk tag contains the type and the length of the complete chunk so it should be very easy to jump from one chunk to the next. For details of the chunk tag please see below.



Let's take a look at the first and usual way for reading out: here is a symbolic illustration of a so called "year tree" (Y=Year, M=Month, D=Day, E=Entry):

Please note: Read this tree from the left (high) to the right (low). Every i64Lower pointer somewhere always points to an chunk in the next lower layer of the tree. Every i64Next pointer somewhere always points to another chunk in a queue on the same level (this is used for year and entry chunks only) or to NULL. All pointers are 64 bit pointers.

As you see in the illustration above all starts from the main index i64Lower pointer to a series of year chunks. These year chunks are organized as a queue using their i64Next pointers, the last year in the queue has the i64Next pointer pointing to NULL. finefiles ensures that years are sorted in ascending, the highest year is always the last entry in this queue. By skipping unwanted years you skip all data they contains which speed up list operations dramatically.

The year chunks itself points to a year index chunk using their i64Lower pointer. This index is basically a index of the possible twelve month of the year and directly follows every year chunk so you could read out the year chunk and the year index chunk together. If there is data for a month present the

corresponding pointer in the year index points to a month chunk. The month chunk itself points to a month index using the i64Lower pointer (to speed up you can read out these two chunks together, a month chunk is always followed by a month index chunk).

The month index itself is an index of the possible 31 days for the specific month. If there are data (entries) for the day the index value points to a day chunk. The day chunk itself points to a queue of entry chunks using the i64Lower pointer. The entry chunk queues are build using their i64Next pointer, the last entry points to NULL. The entry chunks itself contains the file data like file path, file name, size, type etc.

Here the field description for every header and chunk:

## 2.2 Main header

A timeline file (TLF) starts with an 40 byte header to identify the file as valid. The design of this header is based on the file header of a [PNG graphic file](#). Each of the header bytes is there for a specific reason, it contains no pointer.

Length	Name	Value	Description
UCHAR	cStart	0x89	Has the high bit set to detect transmission systems that do not support 8 bit data and to reduce the chance that a text file is mistakenly interpreted as a TLF, or vice versa.
33 CHARS	cSignature[ 33 ]	ffTLF-130 finefiles timeline file	In ASCII, the signature "ffTLF-130 finefiles timeline file", allowing to identify the format clearly if it is viewed in a text editor. Please note that the 130 is a version marker and might be changed (higher) in future versions.
2 x UCHAR	cCRLF[ 2 ]	0x0D 0x0A	A DOS-style line ending (CR/LF) to detect DOS-Unix line ending conversion of the data.
UCHAR	cEOF	0x1A	The end-of-file (EOF) character stops display of the file under DOS when the command type has been used.
UCHAR	cLF	0x0A	A Unix-style line ending (LF) to detect Unix-DOS line ending conversion.
WORD	wReserved	0	Reserved 16 bit unsigned integer for future use, 0 in version 130

## 2.3 Chunk tags

Every data segment in a timeline file (we call it "chunk") except the main header starts with a 8 byte "Chunk tag". Because this tag contains the type and length of the chunk you can easily skip unwanted or unknown chunks. A chunk tag structure has a total size of 8 bytes.

Length	Name	Value	Description
4 x UCHAR	cSignature[ 4 ]		Chunks are given a four letter case sensitive ASCII signature (FourCC). This signature always starts with a ' ' character (124) to identify the chunk start easily in a file editor and because this character isn't allowed to use in a filename. The second character is 'I' for Index, 'C' for chunk and 'G' for garbage. Here a list of possible chunks:   III – Index chunk,  TLC – Timeline control data  GEC – Garbage entry chunk  CYC – Year chunk,  IYI – Year index  CMC – Month chunk,  IMI – Month index  CDC – Day chunk,  CEC – Entry chunk
WORD	wChunkLen		Length of chunk including the leading chunk tag.
WORD	wReserved	0	Reserved 16 bit unsigned integer for future use, 0 in version 130

## 2.4 Main index

After the header comes the so called "Main index" which includes the important pointer to the year tree queue and additional pointers. A index header structure has a total size of 120 bytes.

Length	Name	Value	Description
8 BYTE	xChunkTag	III	Description see "Chunk tag"
UINT32	uEntries		This unsigned 32 bit integer contains the amount of entry chunks in this timeline file.
INT64	i64Lower		This 64 bit pointer points to the first year tree or points to NULL if the file contains no (valid) data.
INT64	i64Garbage		This 64 bit pointer points to the garbage queue or points to NULL if no garbage has been produced.
8 BYTE	xPITlastAccess		Contains a PIT (Point in Time) structure which specifies the date and time of the last access to this file.
INT64	i64TLCvalues		Points to an internal structure used to store values of the Timeline Control dialog. This structure has a total size of 460 bytes. If not used this pointer points to NULL.
INT64	i64Reserved1	0	Reserved 64 bit signed integer for future use, 0 in version 130
INT64	i64Reserved2	0	"
INT64	i64Reserved3	0	"
INT64	i64Reserved4	0	"
INT64	i64Reserved5	0	"
INT64	i64Reserved6	0	"
INT64	i64Reserved7	0	"
INT64	i64Reserved8	0	"
UINT32	uReserved	0	Reserved 32 bit unsigned integer for future use, 0 in version 130
INT64	i64Reserved	0	Reserved 64 bit signed integer for future use, 0 in version 130

## 2.5 Year / Month / Day- chunks

The trees are build using this chunks for a year, month or day date. They have a total size of 38 bytes.

Length	Name	Value	Description
8 BYTE	xChunkTag	CYC,  CMC,  CDC	Description see "Chunk tag"
WORD	wID		The ID is the year for a year chunk, the month (1=January) for a month and the day for a day chunk.
INT64	i64Next		This 64 bit pointer points to the next chunk (if any) or points to NULL if none. This pointer is used for year chunks only.
INT64	i64Lower		This 64 bit pointer points to the next (lower) layer in the tree. For example if the chunk is a year chunk it points to the year index chunk.
UINT32	uReserved	0	Reserved 32 bit unsigned int for future use, 0 in version 130
INT64	i64Reserved	0	Reserved 64 bit signed integer for future use, 0 in version 130

## 2.6 Year index

Every year chunk is followed by a year index chunk. This chunk contains pointers to all twelve month of a year to speed up search operations. The year chunk structure has a total size of 126 bytes.

Length	Name	Value	Description
8 BYTE	xChunkTag	Signature  YI	Description see "Chunk tag"
WORD	wID		The year value in wID of the year chunk is repeated
13 x INT64	i64Month[ 13 ]		Array of thirteen 64 bit values containing pointers to the month chunks or a NULL pointer if no data are present. Please note: Because the first month January has a value of 1 - not 0 - there are 13 entries (0 + 1 to 12). The first (0) array element contains data that belongs to the year but could not be assigned to a specific month.
UINT32	uReserved	0	Reserved 32 bit unsigned integer for future use, 0 in version 130
INT64	i64Reserved	0	Reserved 64 bit signed integer for future use, 0 in version 130

## 2.7 Month index

Every month chunk is followed by a month index chunk. This chunk contains pointers to all 31 days of a month to speed up search operations. The month index chunk has a total size of 278 bytes.

Length	Name	Value	Description
8 BYTE	xChunkTag	IMI	Description see "Chunk tag"
WORD	wID		The year value in wID of the month chunk is repeated
32 x INT64	i64Days[ 32 ]		Array of 64 bit values containing pointers to the day chunks or a NULL pointer if no data are present. Please note: Because the first day of a month has a value of 1 - not 0 - there are 32 entries (0 + 1 to 31). The first (0) array element contains data that belongs to the month but could not be assigned to a specific day.
UINT32	uReserved	0	Reserved 32 bit unsigned integer for future use, 0 in version 130
INT64	i64Reserved	0	Reserved 64 bit signed integer for future use, 0 in version 130

## 2.8 Entry chunks

This chunks are the leaves of the data trees containing the file data (file name and path). Because there can be more entries for one day the entries are organized in a chain, the i64Next pointer points to the next entry or to NULL if the end of the chain is reached. The root and the filename of the file are stored direct after the fixed data, please note that the wChunkLen value in xChunkTag for an entry chunk is defined as the length of the fixed data + the length of the root + the length of the filename to make it as easy as possible to jump over an entry chunk. The fixed part of an entry chunk has a total size of 80 bytes.

Length	Name	Value	Description
8 BYTE	xChunkTag	CEC	Description see "Chunk tag"
WORD	wYear		Year value from the year chunk in a higher level.
WORD	wMonth		Month value from the month chunk in a higher level.
WORD	wDay		Day value from the day chunk in a higher level.
INT64	i64Next		This 64 bit pointer points to the next entry for the current day or to NULL if the end of the chain is reached.
INT64	i64DayChunk		Pointer back to the parent day chunk.
WORD	wFileType		The detected file type, for a complete list of all possible values please visit <a href="http://www.finefiles.com/download/filetypes.h.html">http://www.finefiles.com/download/filetypes.h.html</a>
WORD	wRootLen		Length of the root data following this fixed data
WORD	wFileNameLen		Length of the filename following the root data in bytes
WORD	wMD5pos		Byte-position of a found 32 character MD5 hash sequence in the file name starting at 0 or a value of 999 if not found. Note: a value of 0 means the first byte of the filename, not of the complete path.
22 x UCHAR	cReserved[ 22 ]	0	22 bytes reserved for future use, 0 in version 130
INT64	i64FileSize		The size of the file in bytes
UINT32	uReserved	0	Reserved 32 bit unsigned integer for future use, 0 in version 130
INT64	i64Reserved	0	Reserved 64 bit signed integer for future use, 0 in version 130

Please visit finefiles.com to [download a C/C++ example to access timeline files](#) if you would like to write your own routines.

## 2.9 Garbage

finefiles tries to recycle deleted entry chunks when creating new entries. The garbage queue is indexed from the i64Garbage pointer in the main index. Only entry chunks are recycled, all other chunks can't be deleted or recycled at this time. Deleted entries become the signature "|GEC", the wChunkLen is untouched, the i64Next pointer points to the next garbage entry or to NULL. All other data is reset to 0, the root- and filename data is overwritten with the character '#'. You can switch off the recycling of entry chunks in the finefiles settings if you encounter any problems.

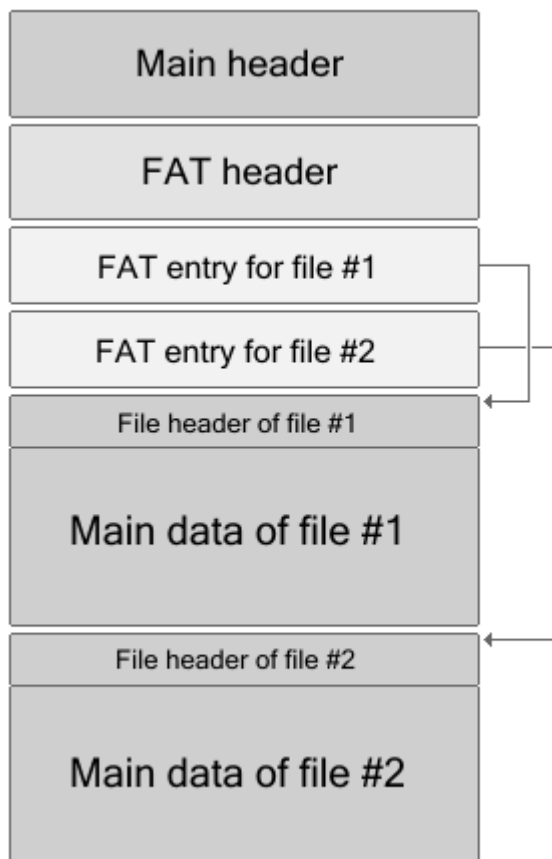
### 3. Sidecar file format specification

---

Here you find the complete file specification for sidecar files which finefiles generates to realize the sidecar features. The specifications may be useful if you would like to write your own routines to read out the data the sidecars contains.

#### 3.1 Basics about sidecars

Sidecars are designed as a "container" to embed files of any type. They provide a directory (we call it "FAT") for the files in the beginning of the sidecar to obtain a fast access. The user can view the content of internal files, add or delete files - all done using the "Sidecar Explorer" which provides simple drag & drop operations. If new files are "embedded" inside the sidecar a new FAT entry is created and appended to the FAT, the main data of the file is appended to the sidecar. Before embedding finefiles tries to classify the new file by the extension - the assignment depends on extension lists in the extensions tab in finefiles settings. If a file could not be classified the sidecar explorer shows it as "binary" or, if selected, it is rejected. Please note that if files are deleted manually the sidecar is not rewritten afterwards, instead the file attribute (value of cAttribute in the file FAT entry structure, see below) is set to SCFAT\_ATT\_DELETED which makes the file to be ignored in all following operations. If a new file is embedded all previous deleted files are skipped during the sidecar rebuild process.



Beside files manually added to the sidecar by the user finefiles manages a special file called "finefiles.Registers". This file, as the name implies, contains data gathered during process and stored in the internal registers. The "finefiles.Registers" file is created and maintained automatically when using the CARRY IN SIDECAR action. You cannot modify the values stored inside this system file and you cannot delete it except you delete the whole sidecar.

#### 3.2 Sidecar structure

On the left side you see a schematic diagram of the structure of a sidecar containing two embedded files. A sidecar always starts with the main header which was designed to identify the file as a sidecar when viewed in a text editor for example. Directly after the main header the FAT header follows. This is the most important header inside the sidecar because it contains the number of valid (and deleted) entries which defines the dimension of the rest of the sidecar file.

The FAT header is followed by a variable number of FAT entries, one for each valid or deleted file the sidecar contains. Inside a FAT entry there is an offset

(pointer) to the file header which is followed by the main data.

#### 3.3 Accessing (readout) a file

If you would like to access a file inside a sidecar first please check the main header to verify that the file you are performing is really in sidecar file format. After that read the following FAT header, the number of FAT entries is calculated by adding the wEntries and wDelEntries value of this header, see

below. Then start to read one FAT entry structure after another and compare the cFileName value the structures contains with the filename you are looking for. Note that if the cAttribute value has the SCFAT\_ATT\_DELETED bit (bit 6 / 64) set the current entry have been deleted before and should be skipped. Continue that read loop until the file is found or all FAT entries are performed.

If the file is found follow the i32Offset pointer inside the FAT entry and read the fixed part of the file header i32Offset is pointing to first. This header contains the size of the complete file header including the variable original file name and an optional padding byte. So add the value of wHeaderLen to i32Offset from the FAT entry structure to get a pointer to the main file data. The size of the main file data is stored in the i32FileSize, don't forget to subtract File header::wHeaderLen and a possible padding byte (FAT entry::i8PaddingBytes) from this value to gain the size of the main data only.

Please note that if the cAttribute value in FAT entry has the SCFAT\_ATT\_SYSTEM bit (bit 2 / 4) set the file is a system file and has NO file header, the i32Offset pointer points directly to the main data in this case.

### 3.4 The main header

A sidecar starts with an 40 byte header to identify the file as valid. The design of this header is based on the file header of a [PNG graphic file](#). Each of the header bytes is there for a specific reason, it contains no pointer.

Length	Name	Value	Description
UCHAR	cStart	0x89	Has the high bit set to detect transmission systems that do not support 8 bit data and to reduce the chance that a text file is mistakenly interpreted as a TLF, or vice versa.
34 x UCHAR	cSignature[ 34 ]	ffSCS-101 finefiles sidecar stream	In ASCII, the signature "ffSCS-101 finefiles sidecar stream", allowing to identify the format clearly if it is viewed in a text editor. Please note that the 101 is a version marker and might be changed (higher) in future versions.
UCHAR	cCRLF[ 2 ]	0x0D 0x0A	A DOS-style line ending (CR/LF) to detect DOS-Unix line ending conversion of the data.
UCHAR	cEOF	0x1A	The end-of-file (EOF) character stops display of the file under DOS when the command type has been used.
UCHAR	cLF	0x0A	A Unix-style line ending (LF) to detect Unix-DOS line ending conversion.
UCHAR	cReserved	0	Reserved byte for future use, 0 in version 100

### 3.5 FAT header

Directly after the main header there is always a FAT header containing basic information about the files the sidecar contains. Even if the sidecar contains no files there is such a FAT header present. This header has a total size of 50 bytes.

Length	Name	Value	Description
12 x UCHAR	cSignature[ 12 ]	ffSC-FAT- 100	In ASCII, the signature "ffSC-FAT-100". Please note that the 100 is a version marker and might be changed (higher) in future versions.
WORD	wEntries		Number of valid entries (files).
WORD	wDelEntries		Number of deleted entries (files).

UINT32	u32BytesAllFiles		Sum of file size of all valid files in the sidecar. Please note that the size of a single file means the file header + the file main data.
UINT32	u32BytesAllDelFiles		Sum of file size of all deleted files in the sidecar. Please note that the size of a single file means the file header + the file main data.
PIT	xPITdateCreated		Date and time when the sidecar was initially created as point in time structure.
PIT	xPITlastAccess		Date and time of the last (writing) access to the sidecar as point in time structure.
UINT8	u8PrefThumbnailNr		Number of the image entry to use as thumbnail in Tiles Window starting at 1 or 0 if no preferred image is defined.
9 x UCHAR	cReserved[ 9 ]	0	Reserved bytes for future use, all 0 in version 100.

### 3.6 FAT entry

For each file (valid or deleted) there is a corresponding FAT entry structure. The number of FAT entries is calculated by adding the wEntries and wDelEntries values of the FAT header. The FAT entry structure has a total size of 50 bytes.

Length	Name	Value	Description
UCHAR	cMagic	#	This ASCII "magic" value is used to identify the start of a FAT entry and to detect read / write errors.
20 x UCHAR	cFileName[ 20 ]		Every file in the sidecar has his own filename usually build out of the original file name, extensions are cut. The sidecar file name is limited to 20 bytes, filled up with spaces if lower. The file name have to be unique in the sidecar.
UCHAR	cAttribute		This value contains the file attributes as bit flags:  SCFAT__ATT__WRITEPROT (bit 0 / 1) SCFAT__ATT__HIDDEN (bit 1 / 2) SCFAT__ATT__SYSTEM (bit 2 / 4) SCFAT__ATT__VOLUME (bit 3 / 8) SCFAT__ATT__FOLDER (bit 4 / 16) SCFAT__ATT__ARCHIVE (bit 5 / 32) SCFAT__ATT__DELETED (bit 6 / 64)
WORD	wFileType		This value contains the file type as a 16 bit integer value, for a complete list of all possible values please visit <a href="http://www.finefiles.com/download/filetypes.h.html">http://www.finefiles.com/download/filetypes.h.html</a>
UINT32	u32Offset		Offset (pointer) to the file data. For regular files this pointer points to a file header structure.
UINT32	u32FileSize		The file size, this means the size of the leading file header + the main file data.
INT8	i8PaddingBytes		This value is 0 if no padding byte was necessary. If this is 1 there have been one padding byte appended at the end of the file main data. Use the following calculation to get the real file size:  Real file size = (i32FileSize - i8PaddingBytes) - size of file header

			You find the size of the file header in wHeaderLen in the file header structure.
7 x UCHAR	cReserved[ 7 ]	0	Reserved bytes for future use, 0 in version 100.
PIT	xPITadded		Date and time when the original file was embedded as point in time structure.
WORD	wReserved	0	Reserved word for future use, 0 in version 100.

### 3.7 File header

Each file, except system files with the attribute SCFAT\_ATT\_SYSTEM (bit 2 / 4) set, led by a file header. This header keep some information about the original file like the original file name and original file attributes which are used for the case that the original file should be restored. The fixed part of the file header structure has a total size of 30 bytes.

Length	Name	Value	Description
UCHAR	cMagic	F	This ASCII "magic" value is used to identify the start of a file header and to detect read / write errors.
UCHAR	cReserved1	0	Reserved byte for future use, 0 in version 100.
WORD	wHeaderLen		The length of the complete header. This means the fixed part (30 bytes) + the variable part (the original file name including a possible padding byte).
DWORD	dwOrgAttribute		A copy of the original file attribute.
WORD	wOrgFileNameLen		The length of the original file name (without a possible padding byte).
PIT	xPITorgFileCreated		Date and time when the original file was created as point in time structure.
12 x UCHAR	cReserved2[ 12 ]	0	Reserved bytes for future use, all set to 0 in version 100.

### 3.8 finefiles Registers file

finefiles uses this file to store internal register data like the calculated MD5 hash. Please note that this kind of files do not have a leading file header. The finefiles registers file structure has a total size of 128 bytes.

Length	Name	Value	Description
10 x UCHAR	cSignature[ 10 ]	ff-REG-100	In ASCII, the signature "ff-REG-100". Please note that the 100 is a version marker and might be changed (higher) in future versions.
WORD	wReserved1		Reserved word for future use. Set to 0 in version 100.
UINT32	uValidFlags		This flag field indicates which registers are defined. The following flags are used (a set bit means that the register is valid):  SCREG_VALID_MD5 (bit 0 / 1) SCREG_VALID_DATESTAMP (bit 1 / 2) SCREG_VALID_TIMESTAMP (bit 2 / 4) SCREG_VALID_FILETYPE (bit 3 / 8) SCREG_VALID_VIDEOSPECS (bit 4 / 16) SCREG_VALID_AUDIOSPECS (bit 5 / 32) SCREG_VALID_IMAGESPECS (bit 6 / 64)
34 x CHAR	sMD5[ 34 ]		This field contains the 32 byte MD5 hash and a

			possible EOS (end of string) following the hash if SCREG_VALID_MD5 is set in iValidFlags.
DATETAG	xDateTag		The value of the datestamp register if SCREG_VALID_DATESTAMP is set in iValidFlags.
TIMETAG	xTimeTag		The value of the timestamp register if SCREG_VALID_TIMESTAMP is set in iValidFlags.
WORD	wFileType		The detected file type, for a complete list of all possible values please visit <a href="http://www.finefiles.com/download/filetypes.h.html">http://www.finefiles.com/download/filetypes.h.html</a>  Note: valid if SCREG_VALID_FILETYPE is set in iValidFlags only.
WORD	wReserved2	0	Reserved word for future use. Set to 0 in version 100.
2 x WORD	wVideoWidth wVideoHeight		Value of video width and height if SCREG_VALID_VIDEOSPECS is set in iValidFlags.
DOUBLE	dVideoFPS		Value of the frames per second register if SCREG_VALID_VIDEOSPECS is set in iValidFlags.
3 x WORD	wVideoHour wVideoMin wVideoSec		Value of the video duration register if SCREG_VALID_VIDEOSPECS is set in iValidFlags.
3 x WORD	wAudioHour wAudioMin wAudioSec		Value of the audio duration register if SCREG_VALID_AUDIOSPECS is set in iValidFlags.
2 x WORD	wImageWidth wImageHeight		Value of the image dimensions register if SCREG_VALID_IMAGESPECS is set in iValidFlags.
34 x UCHAR	cReserved[ 34 ]		Reserved chars for future use. All set to 0 in version 100.

Please visit [finefiles.com](http://finefiles.com) to [download a C/C++ example for sidecar access](#)