FTLight file/stream data structure documentation

Version

2015-09-26

History

2015-09-26	New work title „FTLight“ (Faster Than LIGHT)
2005-04-16	Escaping special characters changed to backslash
2005-03-20	Value data type added
2005-03-16	Framework functionality added
2005-03-09	Time data type added
2005-03-06	Data types, wrapper, arrays and high-speed binary coding added
2005-03-03	Number representation in binary format added
2005-02-26	Examples for metadata and multi-channel receiver added
2005-02-20	Number data types specified in more detail
2005-02-17	Hex digits added for number formats, Dot added to identifier structure
2004-03-10	Introduction added which explains the purposes from a high-level view
2004-01-29	Versioning added
2004-01-22	Request/Response protocol extended by flow control elements
2004-01-21	Request/Response capability added
2004-01-20	FTLight repository organization added
2004-01-19	Data integrity support added
2004-01-18	Explanation for ‘FTLight collection’ added and address usage expanded
2004-01-12	First version based on discussions in the erac-vlbi group

Credits

The following people participated in discussions about the content of this document and provided many ideas that have been implemented in the concept:

Marko Cebokli	marko.cebokli@mors.si	
Jim Sky	radiosky@radiosky.com	web: http://www.radiosky.com

Disclaimer

Content

FTLight file/stream data structure documentation 1

- Version 1
- History 1
- Credits 2
- Disclaimer 2
- Content 3
- Purpose 5
- Introduction 5
- Case study 6
  - A – Data column 6
  - B – Timestamped data set 6
  - C – Structured information 6
- Composition of structured information 6
- Mapping structured information into a file/stream 7
  - Design goals 7
  - File/stream structure 7
- Comparing FTLight structures to folders, registry entries and databases 11
- Data types 12
  - Special characters 12
  - Text data type 12
  - Number data type 12
  - Binary data type (BinX) 13
  - Number representation in binary format (NumX) 13
  - Data type representation in binary format (TypeX) 14
  - Identifier data type 16
  - Address data type 17
- Framework functionality 20
  - Optional values 20
  - Comments 20
  - Default values 20
  - Duplicate specifications 20
  - Interpretation 20
- Data integrity 21
  - Checksum calculation 21
  - Corrupted data retrieval 22
- FTLight repository 23
- Versioning 24
  - Multiple versions in the repository 24
- Request/Response procedure 25

Storing request/response streams....25
Request structure....26
Appendix A....28
A.1 Example for storing multi-level metadata in front of interferometry data....28
A.2 Example for a multi-channel receiver with random channel selection....28
A.3 Example for a multi-channel receiver with regular channel selection....28

Purpose

The FTLight file/stream data structure has been designed to facilitate the collection and exchange of huge data volumes as well as to allow for accessing subsets of those data volumes down to the very detail information items.

A special need for handling huge data volumes exists in radio astronomy, particularly when combining the data streams from distributed antenna sites in order to get them working in an integrated function, e.g. to generate sky surveys by interferometer networks.

Although the specification has been adjusted to serve the special needs of interferometer networks, it is also well suited to connect any set of distributed computers and let them work in an integrated way.

Introduction

In current computer science the layered approach for connecting computer systems with each other is widely used. A layered approach means that the information to be transferred is wrapped into a data package along with additional information that informs for example about where the information should go to, where does it come from, which categories it belongs to and others. This package is possibly wrapped into a subsequent package which serves the same purpose but on a higher level. The resulting package can in turn be wrapped by a next wrapper, and so forth.

Usually the control information is placed in a header of the new data package whereas the information to be transferred goes into the body of that package. There can also be a trailing information chunk which contains for example additional information for maintaining data integrity. The structure of the final data package that goes effectively through the transfer channel can thus become complex and the overhead can become big:

Header1 Header 2 … Header N (actual information) Trailer N … Trailer 2 Trailer 1

An advantage of the layered structure is apparently that the layers are independent from each other and that they can thus be implemented separately. Furthermore appropriate combinations of layers can be used to transfer the actual information through arbitrary heterogeneous systems.

On the other hand each layer increases the size of the transferred information and the requirement for independent layers makes it almost impossible to optimize transport across multiple layers. In an extreme case the analysis of a data stream that transfers a table with several thousand entries provided to the result that a data stream based on a SOAP protocol contained 95% overhead and only 5% actual data compared to the binary presentation of data that was fed into the transport channel.

It must be admitted that the SOAP protocol is well suited for transferring arbitrary structured data due to the usage of XML files as a basis for data presentation which other transfer protocols do not offer. Thus with existing data protocols it was always a trade-off between transfer efficiency and data structuring capabilities that decided upon which protocol to use under given circumstances.

The FTLight data specification aims at getting the best out of both worlds, to provide for highly efficient data storage and transport on the one hand and to allow for arbitrary data structures on the other hand along with other features that are related to the need of dealing with arbitrary huge data volumes.

Another goal is to be independent from transport layers that control the flow of information between two computers. For successful data transport it should be sufficient to have a connection like IP services available which allows for specifying the recipient of a data stream. Alternatively any other connection like for example a serial line will be sufficient to run a FTLight data stream. The implementation of this goal will open a broader field of application in heterogeneous environments and at the same time it is a basis for thorough and deep transport optimizations.

A special attention needs to be paid to the turn-around time of data packages that leave a system for travelling to a second system which in turn sends a response back to the first system. The more loops are needed for accomplishing a proper data transfer the more the delay time becomes important. If the delay time becomes very big like for example in case of signals travelling between space probes and stations at earth at most one complete signal loop can be applied for data transfer. All data protocols which need multiple signal loops will be rendered useless in this case.

Case study

A – Data column

A single data column represents one of the simplest data structure:

```
2602
2595
2594
..
```

The main disadvantage of a single data column is the lack of any information where the data comes from, what does it mean, which units apply to the numbers and so forth.

B – Timestamped data set

A header and timestamps provide for more information about the measurement data:

```
Time                    Flux        Temperature
[sec since 1.1.1970]    [Jy]        [°C]
1073217600.370          2602        -2.4
1073217600.390          2595        -2.4
1073217600.410          2594        -2.3
```

With header information and a timestamp the meaning of the measurement values becomes more determined. However, also in this case there is no information about the radio source being monitored or the equipment that was used.

C – Structured information

Providing information in a structured way clarifies all circumstances of the measurement data down to the very detail:

```
EKD@JO64qc.RSpectro,1073217600:FTLight,2004-01-12
,Antenna,dish 90cm
,Azimuth:degree,0
,Elevation:degree,15
,Frequency:GHz,10.600
,Bandwidth:KHz,250
,Data
:Time,Flux,Temperature
:[sec since 1.1.1970],[Jy],[°C],@
:1073217600.370,2602,-2.4
:1073217600.390,2595,-2.4
:1073217600.410,2594,-2.3
```

The above structured information block extends the previous timestamped data set with meta information which describes for example the creation time of the file/stream along with the specification that was used; it identifies the operator, the location as well as the equipment with a key (EKDJO64qc.RSpectro) and it provides for those observation details like antenna, azimuth, elevation, frequency and bandwidth.

Composition of structured information

A convenient way to pose a structure on single information items is to put related items into an ordered collection which basically assigns a sequence number to each item:

```
[0:1073217600, 1:Antenna, 2:Azimuth, 3:Elevation, 4:Frequency, 5:Bandwidth, 6:Data]
```

Subsequently the items of an ordered collection can be referenced by sequence number or by the value which represents the item, e.g. "`1:`" is equivalent to "**`Antenna`**".

There are two ways to extend an existing ordered collection, either in size or in depth. If it is extended in size this means that more items will be added to the collection. On the other hand, extending the information structure's depth provides to new collections that will be attached to an item of the current collection.

```
Example:     [0: 1073217600, .. 4:Frequency, ..]
                                  |
                                  |_____ [0:GHz, 1:10.600]
```

Allowing for extension of ordered collections in size as well as in depth provides to hierarchical data structures. Addressing an item in those hierarchical data structures requires mentioning all superior items that represent the path from the root down to the item that is addressed. Hereby the path can be expressed either by mentioning the values of all superior items or by referring to the sequence numbers that the items have been assigned in their ordered collections.

Below a comparison is listed showing the path of all items in the previously given sample data structure expressed as a chain of sequence numbers on the left hand side and the value of each item on the right hand side:

0	**EKD@JO64qc.RSpectro**
0-0	**1073217600**
0-0-0	FTLight
0-0-1	2004-01-12
0-1	**Antenna**
0-1-0	Dish 90cm
0-2	**Azimuth**
0-2-0	Degree
0-2-1	0
0-3	**Elevation**
0-3-0	Degree
0-3-1	15
0-4	**Frequency**
0-4-0	GHz
0-4-1	10.600
0-5	**Bandwidth**
0-5-0	KHz
0-5-1	250
0-6	**Data**
0-6-0	***Time***
0-6-0-0	[sec since 1.1.1970]
0-6-0-0-0	1073217600.370
0-6-0-0-1	1073217600.390
0-6-0-0-2	1073217600.410
0-6-1	***Flux***
0-6-1-0	[Jy]
0-6-1-0-0	2602
0-6-1-0-1	2595
0-6-1-0-2	2594
0-6-2	***Temperature***
0-6-2-0	[°C]
0-6-2-0-0	-2.4
0-6-2-0-1	-2.4
0-6-2-0-2	-2.3

Mapping structured information into a file/stream

Design goals

Between several possibilities to map structured information into a file/stream the preferred solution should be featured by a minimal data volume of the file/stream related to a given set of information. Therefore explicit structure elements like the tags in XML files will be replaced by implicit rules that work without the need of explicit structure information in most cases. In a small group of remaining cases the structure information will be added in the short form as a chain of sequence numbers as it was described and shown in the above example.

In addition to the already mentioned requirements a further goal is to have the possibility to review FTLight files/streams conveniently and to support adjustments of information items by a text editor.

File/stream structure

A file/stream is made from 8-bit values (bytes) which represent the numbers from 0 to 255. Some ASCII characters like 13 (CR – carriage return) and 10 (LF – line feed) are used to structure the document whereas the majority of characters is used to code information.

Lines inside a file/stream

The high-level structure of a file/stream is imposed by end-of-line (CR+LF) characters. Inside each line special delimiter characters provide for separation of information items.

Items inside a line

There are four delimiters that divide a line into information items. Additionally, each of those delimiters serves a special purpose for further structuring a line and for determining the type of information of a subsequent item:

44 (comma)	- Extending path or collection, next item is text or numerical.
59 (semicolon)	- Extending path or collection, next item is binary or special.
58 (colon)	- Starting collection, next item is text or numerical.
61 (equal sign)	- Starting collection, next item is binary or special.

Example:

```
      Frequency:GHz,10.600
```

translates into the following structure:

```
0               Frequency
0-0                GHz
0-1                10.600
```

Item replication in subsequent lines

Whenever the same item is used in a subsequent line on the same position it can be omitted:

Example:

```
EKD@JO64qc.RSpectro,1073217600:FTLight,2004-01-12
EKD@JO64qc.RSpectro,Antenna,dish 90cm
```

is equivalent to:

```
EKD@JO64qc.RSpectro,1073217600:FTLight,2004-01-12
,Antenna,dish 90cm
```

and it translates in both cases into the following structure:

```
0            EKD@JO64qc.RSpectro
0-0             1073217600
0-0-0              FTLight
0-0-1              2004-01-12
0-1             Antenna
0-1-0              Dish 90cm
```

Extending a collection's size by a subsequent line

The first different path item in a subsequent line extends the size of the collection on that level:

Example:

```
EKD@JO64qc.RSpectro,1073217600:FTLight,2004-01-12
,Antenna,dish 90cm
```

The first line generates a path:

```
0            EKD@JO64qc.RSpectro
0-0             1073217600
0-0-0              FTLight
0-0-1              2004-01-12
```

whereas the second line adds a new item "**Antenna**" to the collection on the second level that consisted of only a timestamp item "`1073217600`" before:

```
0-1             Antenna
0-1-0              Dish 90cm
```

Maintaining a current path

A path that was specified in a line will become the current path. Whenever a line starts with a text, numerical or binary item the previous path is preserved and will be used as current path. A colon at the first position of a line indicates a text/numerical data type. Otherwise the field will be evaluated as binary, address or identifier data type.

Example:

```
EKD@JO64qc.RSpectro,Data:Time,Flux,Temperature
```

is equivalent to:

```
EKD@JO64qc.RSpectro,Data
:Time,Flux,Temperature
```

and translates in both cases into the following structure:

```
0            EKD@JO64qc.RSpectro
0-0             Data
0-0-0              Time
0-0-1              Flux
0-0-2              Temperature
```

The "**EKD@JO64qc.RSpectro**" item is recognized as the root of the path because of the '@' character in the string as it will be defined later under "Data types".

Maintaining a parent collection

Whenever a collection is created on a path this collection becomes the parent collection. The items of that collection will be used as parent for subsequent parallel writing operations.

Example:

```
EKD@JO64qc.RSpectro,Data
:Time,Flux,Temperature
```

The newly created collection [Time,Flux,Temperature] will be the parent collection.

Parallel writing operations

Most often data sets consist of multiple columns. Parallel writing supports specifying one row of such a data set in one line. The items of that line will go in parallel to the collections that are attached as child collections to the items of the parent collection.

Usually multiple parallel writing operations will be performed without changing the parent collection. In case the items of a parallel writing operation need to become the new parent collection the line that contains those items will get a single '@' character item in the end of the line.

Example:

```
EKD@JO64qc.RSpectro,Data
:Time,Flux,Temperature
:[sec since 1.1.1970],[Jy],[°C],@
```

The items of the third line will be written into the collections that are attached to the items in the second line and will become the new parent collection at the same time. They now represent a virtual collection which solely serves as the parent collection for parallel writing but it has no real representation in the structured data set. The structure therefore looks like follows:

```
0           EKD@JO64qc.RSpectro
0-0            Data
0-0-0             Time
0-0-0-0              [sec since 1.1.1970]
0-0-1             Flux
0-0-1-0              [Jy]
0-0-2             Temperature
0-0-2-0              [°C]
```

Adding subsequent data rows to the previous example will fill the table with data:

Example:

```
EKD@JO64qc.RSpectro,Data
:Time,Flux,Temperature
:[sec since 1.1.1970],[Jy],[°C],@
:1073217600.370,2602,-2.4
:1073217600.390,2595,-2.4
:1073217600.410,2594,-2.3
```

Finally this translates into the following structure:

```
0           EKD@JO64qc.RSpectro
0-0            Data
0-0-0             Time
0-0-0-0              [sec since 1.1.1970]
0-0-0-0-0               1073217600.370
0-0-0-0-1               1073217600.390
0-0-0-0-2               1073217600.410
0-0-1             Flux
0-0-1-0              [Jy]
0-0-1-0-0               2602
0-0-1-0-1               2595
0-0-1-0-2               2594
0-0-2             Temperature
0-0-2-0              [°C]
0-0-2-0-0               -2.4
0-0-2-0-1               -2.4
0-0-2-0-2               -2.3
```

Excel import

The structured data from previous example can be imported into Excel and will provide to the following view. Thus the design goal to display the structured data file/stream in a reasonable way has been accomplished. Furthermore all data has become directly available for flexible processing:

EKD@JO64qc.RSpectro	Data		
Time	Flux	Temperature	
[sec since 1.1.1970]	[Jy]	[°C]	@
1073217600.370	2602	-2.4	
1073217600.390	2595	-2.4	
1073217600.410	2594	-2.3	

Comparing FTLight structures to folders, registry entries and databases

From introduction of collections the question might come up what is the difference between FTLight collections along with their hierarchical data structures on the one hand and well know information structures like folders in file systems, registries and databases on the other hand. A short answer could be that the FTLight specification aims at removing artificial limitations that have been put on other structures mostly for reasons dealing with performance and efficient resource management.

The FTLight specification is committed to performance and efficient resource usage, too. However, the highest priority takes being conceptually unlimited. The limitations will come into play when a FTLight file is written into a file system or when a FTLight stream is transferred between computers. In those practical applications the limitations will come from the size of the storage device where the FTLight file is written to respectively from the bandwidth of the communication channel which transfers a FTLight stream from one computer to another one.

In the following some examples are reviewed where the FTLight structure overcomes limitations:

Folders in file systems

Usually each file system imposes limitations on the length of the elements where a path is made from and on the set of characters that can be used for path elements. Additionally, the total length of a path is most often also limited either by the design of the file system or by the operating system that runs the file system. A commonly used limitation is for example 1024 bytes for the total length of a path along with excluding characters like slash'/' and backslash '\' from path elements.

Those limitations have gone in the FTLight specification. Path elements as well as the total length of a path are conceptually unlimited. Additionally, there are two data types available (Text and Binary), which allow for constructing the elements of a path from arbitrary single-byte characters 0..255 as well as using any bit field as a part of a path.

Registries

Basically there is no other conceptual difference between a registry entry and a file in a file system except additional limitations on the size of the value that can be assigned to an entry in a registry. The limitation of both structures is that only one value can be assigned to a file/registry name on a given path. If multiple related files or values need to be stored then the information structure has to be expanded by an additional level first.

In a registry the additional level would consist of a collection of entries where every entry would need to have a unique name. Subsequently the collection of related values can be assigned to those names. In a file system the equivalent would be storing related files using filenames with different extensions, for example .exe for the executable and .ini for initialization information while having the same basic name for both files. However, this is only a workaround for that limitation.

The FTLight specification overcomes assigning only one value to a given registry name or file name by the idea of assigning a conceptually unlimited collection of items to a given path. This allows for example for maintaining all measurement data under the name of the data source that generated those values. The advantage over storing all measurement data just into a file is that each particular measurement value continues to be a separate entity under the roof of a common specification and thus no additional rules are needed for writing/retrieving those values from file.

One can also look at the FTLight specification as unifying the traditional concept of path, folder, and files with various file formats into one concept of a collection that can be combined with other collections for the purpose of generating unlimited hierarchical information structures.

Databases

Relational database systems are very efficient for maintaining data that can be organized in tables along with a set of rules for the relations between those tables. However, most often they are not well suited for maintaining hierarchically organized data structures.

The FTLight specification extends the concept of hierarchical data structures by the ability to synchronize ordered collections for representing table like data, for example to maintain series of measurement data along with a timestamp as a key pointing to measurement data records.

Data types

The following data types allow for expressing each information item in the most appropriate way. They have been defined such that a parser can uniquely identify them when working on a FTLight file/stream:

• Text	(single-byte characters from 0 to 255, unlimited number of characters)
• Numbers	(all data formats that can be composed from “0123456789 - + . E e X x A a B b C c D d F f”)
• Binary	(all bit fields starting with a single bit up to an unlimited number of bits)
• Identifier	(containing information about the operator, location and the used equipment)
• Address	(provides for referencing items inside a file/stream and for establishing links)

Special characters

The following characters serve special purposes in the FTLight file/stream definition and must be avoided respectively escaped in all data types where they could be mixed-up:

• 10 (line feed)	- line separator
• 13 (carriage return)	- line separator
• 44 (comma)	- field separator
• 45 (minus sign)	- address component
• 58 (colon)	- field separator
• 59 (semicolon)	- field separator
• 61 (equal sign)	- field separator
• 64 (@ sign)	- identifier, operator
• 96 (back apostroph)	- QUERY operator
• 127 (del)	- DEL operator

Text data type

The text data type can handle all single-byte characters from 0 to 255. In order to avoid conflicts all special characters that where mentioned before have to be escaped. This will be done by putting a back slash in front of those characters. When parsing a Text item the back slash in front of special characters has to be removed.

The possibility to handle the complete set of single-byte characters makes the Text data type basically capable of coding any kind of files including binary files. However, when using the Text data type for this purpose one should be aware that those files can possibly not be properly opened and displayed with a text editor or spreadsheet program as it was intended because of the control characters (0..31) which will most likely be contained in those files.

Furthermore given an equal distribution of all possible characters in a binary file the doubling of all special characters provides to a coding efficiency of about 96%. Also from this point of view the Binary data type with about 97% coding efficiency is preferred. Additionally, the Binary data type avoids all control characters (0..31).

Example:

This is an example of the Text data type\: “mail\@server.com”.

Number data type

Whenever an item which is declared as text/number contains only the following characters: “0123456789 - + . E e X x A a B b C c D d F f” it will be further investigated if it constitutes a valid number. If this is the case the item will be converted to a number. The size of a number is unlimited.

The following number formats will be recognized:

- **Integer** – any combination of digits 0..9, e.g. 0, 10, 938776658832671414423574758
- **Float** – two integers connected by a dot, e.g. 123.4, 0.56, .87, 543.

- **Exponential** – a float extended by an exponent, e.g. 0.56E-2, 0.62e12, 4.283E+5
- **Hexadecimal** – ‘0x’ or ‘0X’ followed by hex digits 0..9,A..F,a..f, e.g. 0x0023BAFC

Binary data type (BinX)

The coding of binary data is based on a set of symbols which represent the values from 0..215. Those symbols are mapped to characters in a FTLight file/stream in such a way that no control character (0..31) and no special character is needed to express a symbol.

In the direction from a symbol to an extended ASCII character the conversion works as follows:

symbol = 12	→ character = 248	(avoid 44 - comma)
symbol = 13	→ character = 249	(avoid 45 - minus sign)
symbol = 26	→ character = 250	(avoid 58 - colon)
symbol = 27	→ character = 251	(avoid 59 - semicolon)
symbol = 29	→ character = 252	(avoid 61 - equal sign)
symbol = 32	→ character = 253	(avoid 64 - @ sign)
symbol = 64	→ character = 254	(avoid 96 - back apostroph)
symbol = 95	→ character = 255	(avoid 127 - del)
others	→ character = symbol + 32	

In the backwards direction from a character to a symbol the conversion works as follows:

character 32..247	→ symbol = character – 32
character 248	→ symbol = 12
character 249	→ symbol = 13
character 250	→ symbol = 26
character 251	→ symbol = 27
character 252	→ symbol = 29
character 253	→ symbol = 32
character 254	→ symbol = 64
character 255	→ symbol = 95

Four symbols received from four subsequent characters in a binary data field are combined with a radix of 216 to compose a 31-bit field. If more than 31 bits are contained in the bit field a multitude of four characters is used to compose the overall bit field based on 31 bit fields respectively one, two or three more characters are added to reach the bit field length. In this way the coding of bit fields of arbitrary size like for example also graphics, sound and video files can be performed.

Example:

Let the binary data field be the string ‘ABCD’ which translates into the symbols:

A → 65-32=**<u>33</u>**   B → 66-32=**<u>34</u>**   C → 67-32=**<u>35</u>**   D → 68-32=**<u>36</u>**

The value of the bit field will be:

(((**<u>33</u>** * 216 + **<u>34</u>**) * 216) + **<u>35</u>**) * 216 + **<u>36</u>** = **<u>334157868</u>**

what is equivalent to a bit field with a content of: **0010011111010101101100000101100**

The chosen coding scheme represents a 31-bit field by four characters of the extended ASCII table. This is equal to a 31/32 efficiency or about 97% compared to a binary format.

The available range for a group consisting of four symbols is

216^4 - 1 = 2,176,782,335.

Compared to the maximal 31-bit value of 2,147,483,647 there is an excess of 29,298,688 combinations.

To be done:

The excess combinations offer a good opportunity to add more features like a boundary possible on every bit or compression of zero-bit or one-bit fields as well as improved compression of more complex repetitive structures.

Number representation in binary format (NumX)

Representation of numbers requires the following elements:

- Integers
- Negative numbers
- Fraction of integers (rational numbers)
- Exponent
- Base of an exponent
- Negative exponent

The basic binary data type (BinX) is used to represent arbitrary huge integer numbers. The back apostroph (96) will be introduced to structure a binary data field into multiple components which will have the following meaning dependent on the topology:

BinX	- Integer, e.g. 12345
`BinX	- **-BinX**, negative number (integer in this case), e.g. -12345
BinX`	- **1 / BinX**, fraction, e.g. 1/12345
BinX1`BinX2	- **BinX1 * BinX2**, product of integers, e.g. 1234*100=123400
BinX1`BinX2`	- **BinX1 / BinX2**, fraction of integers, e.g. 12345/100=123.45
BinX1`BinX2`BinX3	- **BinX1 * BinX2 exp BinX3**, scientific, e.g. 123E+45
BinX1`BinX2``BinX3	- **BinX1 * BinX2 exp -BinX3**, scientific, e.g. 123E-45
BinX1`BinX2`BinX3`BinX4	- **BinX1 / BinX2 * BinX3 exp BinX4**, scientific, e.g. 1.3E+45
BinX1`BinX2`BinX3``BinX4	- **BinX1 / BinX2 * BinX3 exp –BinX4**, scientific, e.g. 1.3E-45

Data type representation in binary format (TypeX)

It is common that each software application uses a specific data format for making data persistent by writing it to a data storage medium respectively for transferring information from one instance of an application to a second instance of an application. Those data formats have been created by the developers of the software application in order to store and transfer data in the most efficient way based on the requirements that the specific application is addressing. Often developers hit barriers when they want to introduce new features into an existing application due to constraints that they established in the initial specification which resulted from the initial requirements. From the user point of view the application goes through multiple versions during its life cycle where compliance between the data formats of subsequent software versions is one of the biggest challenges that developers are faced with. In past there was often no way to provide for backward compatibility and thus the only choice that remained was to go for a new format specification and try to solve backward compatibility by data converters.

Although the FTLight specification has been designed as an open container for storing and transferring arbitrary data structures in an efficient way, it might nevertheless be useful to wrap existing data files by a FTLight container in order to take advantage e.g. of the flexible metadata representation features. At the same time the versioning problem can be solved in a transparent way when taking advantage of the specific FTLight elements provided for this purpose like unique identifiers and internal references in combination with the update procedure. On the other hand it might be recommended to introduce different binary formats were each particular format addresses specific features like processing speed or data compression in a better way than the general FTLight built-in representations can do. The following format elements will allow to wrap other (outside FTLight) data formats as well as to extend the built-in format features in a transparent manner.

Type identifiers

The first four BinX symbols in a binary FTLight field serve as a type identifier. Once a type is specified in a binary field it will be forwarded also to all descendent binary FTLight fields.

The type identifier uses those combinations of BinX symbols which are not used when representing a 31-bit field by four numbers with a 216 radix. If the values of the first four symbols represent instead a valid 31-bit number then the field constitutes a usual BinX binary format field.

The following TypeX identifiers are defined based on BinXmax = 216^4 – 1 = 2,176,782,335:

FTLightOpen	= BinXmax - 0	= 2,176,782,335	- Open format based on FTLight symbols
FTLightWrap	= BinXmax - 1	= 2,176,782,334	- Arbitrary format wrapped by BinX
BinMCL	= BinXmax - 2	= 2,176,782,333	- BinMCL coded array of bit fields
BinXbinary	= BinXmax - 3	= 2,176,782,332	- BinX coded array of bit fields
BinXstring	= BinXmax - 4	= 2,176,782,331	- BinX coded array of strings
BinXvalue	= BinXmax - 5	= 2,176,782,330	- BinX coded value
BinXtime	= BinXmax - 6	= 2,176,782,329	- BinX coded time

Type controls (ControlX)

A TypeX identifier can be followed by type control fields. The meaning of the type control fields are specific and will be described individually for each type. A control is a non-limited sequence of bytes which are coded accordingly to the BinX rules. The controls are separated by back apostroph (96) characters:

TypeX`BinX(n)`ControlX1`…`ControlXn-1`FTLight(binary data)

The first control field always contains the total number of all control fields. Therefore it has been called BinX(n) = ControlX0.

If the character after the TypeX identifier is not a back apostrophe character then the following binary block of data is a homogenous bit field with no further explicit structure information, at least not known to the FTLight container. In this case a BinX coding is assumed for the binary data.

FTLightOpen data type

The FTLightOpen data type is an invitation to develop highly specific data formats based on the radix 216 scheme and using only valid FTLight characters. It is recommended to describe those data formats in the Internet and to provide a URL to the appropriate web page in ControlX1 = URL:

FTLightOpen`BinX(n)`BinX(URL)`ControlX2`…`ControlXn-1`FTLight(binary data)

The number of the controls following the URL is specific to the format.

FTLightWrap data type

The FTLightWrap data type is a wrapper for existing data formats from outside the FTLight specification respectively for complete FTLight repositories. It is recommended to provide a URL to a web page in ControlX1 = URL for the application that can handle the encapsulated data file or alternatively the name of the application if a URL is not available.

FTLightWrap`BinX(n)`BinX(URL/application)`ControlX2`…`ControlXn-1`BinX(binary data)

The number of the controls following the URL is specific to the format. If there are no controls available after the FTLightWrap type identifier (no back apostrophe after the type identifier) then the binary data is assumed to be a complete FTLight repository.

BinMCL data type

The BinX data type requires several arithmetic operations for calculating subsequent BinX characters. Dependent on the type of processor this may provide to a considerable time effort for generating the BinX data stream. For extreme situations a coding is recommended which can be generated solely by shifting bits and applying a lookup table without involving any time consuming arithmetic operations. This format can be applied for example on computers with a low processor speed when nevertheless a high throughput is needed, however, it can also be applied on high performance computers when extreme requirements regarding the data rate have to be fulfilled.

A coding scheme that meets the described requirements has been developed by Marko Cebokli as a 15/16 format. It operates on 30-bytes fields or 15 times 16-bit fields. First the most sign bit (MSB) from every 16 bit value is shifted to a 16-bit register. Next each of the 16-bit values with the stripped MSB is looked-up as a 15-bit value in a pre-calculated table. That table has been filled with the proper FTLight characters for a total of 32768 15-bit values before. Finally the register which collected all the MSBs and which is adjusted such that the MSB of the last 16-bit value is at the LSB position of the 16-bit register is also looked-up in the pre-calculated table.

An implementation of a 15/16 algorithm provides for a coding efficiency of 93.8% in comparison to the 96.9% of a 31/32 algorithm like BinX. The advantage of the 15/16 algorithm is that when implemented with a lookup table it needs only about 42 processor cycles per byte on Pentium 4 processors whereas a 31/32 algorithm needs about 135 processor cycles per byte and thus the 15/16 algorithm will run more than 3 times faster than the 31/32 algorithm.

The layout of BinMCL is similar to BinXbinary:

BinMCL`BinX(n)`BinX(dimension_1)`…` BinX(dimension_n-1)`BinX(binary data)

For example, an array of two channels from a single-bit interferometer containing data from two interferometer devices would look like follows:

BinMCL`BinX(4)`BinX(1)`BinX(2)`BinX(2)`BinMCL(binary data)

One measurement point will consist of four bits, where two bits constitute the I and Q channels of the first device and the other two bits constitute the I and Q channels of the second device.

It becomes obvious that in this case even each bit of the binary data field has got its own address inside the FTLight stream and can thus be referred to by that address. This can become essential when combining data streams from several observation sites into one result on the correlator site.

BinXbinary data type

The BinXbinary data type equals the BinX binary format if no controls are present. In case of more than one control those values will be the sizes of the dimensions of an array of bit fields and the first control will be the total number of dimensions in that array:

BinXbinary`BinX(n)`BinX(dimension_1)`…` BinX(dimension_n-1)`BinX(binary data)

For example, an array of samples from a 100-channel spectrometer with 12-bit sample values will look like follows:

BinXbinary`BinX(3)`BinX(12)` BinX(100)`BinX(binary data)

This binary field will be evaluated as a 3-dimensional bit field with the first two dimensions set to 12 bits for a basic value and 100 values for a row of measurement data. The size of the third dimension, that is the number of measurement rows or frequency sweeps will follow from the size of the binary data field.

BinXstring data type

The BinXstring data type allows for storing n-dimensional arrays of zero-terminated strings.

BinXstring`BinX(n)`BinX(dimension_1)`…` BinX(dimension_n-1)`BinX(sequence of strings)

For example, an array of translations from a first language to a second and a third language will be stored as follows:

BinXstring`BinX(2)`BinX(3)`BinX(term11 term12 term13 … termN1 termN2 termN3)

Each term is followed by a zero which indicates the end of a zero-terminated string. Each term in a tripel of strings will have the same meaning but in different languages.

BinXvalue data type

The BinXvalue data type allows for specifying a fraction or a multitude of a unit or any ratio of both, e.g. as a base value for measurement data.

BinXvalue`BinX(2)`BinX(multiplier)`BinX(value)

BinXvalue`BinX(3)`BinX(multiplier)``BinX(neg_value)

BinXvalue`BinX(3)`BinX(multiplier)`BinX(devisor)`BinX(value)

BinXvalue`BinX(4)`BinX(multiplier)`BinX(devisor)``BinX(neg_value)

BinXvalue`BinX(4)`BinX(multiplier)`BinX(devisor)`BinX(exponent)`BinX(value)

BinXvalue`BinX(5)`BinX(multiplier)`BinX(devisor)`BinX(exponent)``BinX(neg_value)

BinXvalue`BinX(5)`BinX(multiplier)`BinX(devisor)``BinX(neg_exponent)`BinX(value)

BinXvalue`BinX(6)`BinX(multiplier)`BinX(devisor)``BinX(neg_exponent)``BinX(neg_value)

BinXvalue`BinX(5)`BinX(multiplier)`BinX(devisor)`BinX(base)`BinX(exponent)`BinX(value)

BinXvalue`BinX(6)`BinX(multiplier)`BinX(devisor)`BinX(base)`BinX(exponent)``BinX(neg_value)

BinXvalue`BinX(6)`BinX(multiplier)`BinX(devisor)`BinX(base)``BinX(neg_exponent)`BinX(value)

BinXvalue`BinX(7)`BinX(multiplier)`BinX(devisor)`BinX(base)``BinX(neg_exponent)``BinX(neg_value)

A relative measurement value which is expressed as an integer will be combined with a multiplier, a divisor, a base and an exponent in order to get to the actual measurement value. For example, a relative measurement value=1 which represents a 1E-26 part of a base unit will be coded as follows:

BinXvalue`BinX(5)`BinX(1)`BinX(1)``BinX(26)`BinX(value)

BinXtime data type

The BinXtime data type allows for specifying a fraction or a multitude of a second or any ratio of both as a time basis.

BinXtime`BinX(2)`BinX(multiplier)`BinX(time)

BinXtime`BinX(3)`BinX(multiplier)`BinX(devisor)`BinX(time)

BinXtime`BinX(4)`BinX(multiplier)`BinX(devisor)`BinX(exponent)`BinX(time)

BinXtime`BinX(5)`BinX(multiplier)`BinX(devisor)``BinX(neg_exponent)`BinX(time)

BinXtime`BinX(5)`BinX(multiplier)`BinX(devisor)`BinX(base)`BinX(exponent)`BinX(time)

BinXtime`BinX(6)`BinX(multiplier)`BinX(devisor)`BinX(base)``BinX(neg_exponent)`BinX(time)

All time data refers to January, 1st 1970. The time increment to be used in order to get from that base time to an arbitrary point in time is specified accordingly to the rules of numbers as it was determined before. For example, a time increment of a millisecond will be coded as follows:

BinXtime`BinX(3)`BinX(1)`BinX(1000)`BinX(time)

Identifier data type

An identifier is defined as a string that contains a @ sign. When an identifier occurs as the first item in a line of a FTLight file/stream then it will become the root of any subsequent information. An identifier represents the highest level in any information hierarchy.

The purpose of an identifier is to make all information items unique in a universal sense. “In a universal sense” means that the existence of a duplicate key is highly improbable and that this is true for all data that is collected on earth, on other planets, in other solar systems or at any other place in the universe.

This specification will not solve the problem how to make a key universally unique. Nevertheless the following recommendation aims at generating unique keys when collecting radio astronomy data on earth. The following story serves as a guide for the key generation recommendation:

“A person **A** born in **B** and collecting data in **C** with the equipment **D**”

The proposed generator rule will use for example the uppercase first letters from A and B, a locator specified accordingly to the locator system that is used in ham radio and it will take a shortcut for the equipment D. The components will be combined as follows:

AB@C.D

Example:

The identificator “EKD@JO64qc.RSpectro” consists of the following components:

• E	- Eckhard, first name
• K	- Kantz, last name
• D	- Dambeck, place of birth
• @	- makes the string an identifier
• JO64qc	- location of the radio astronomy equipment
• .	- separator for convenient reading

- RSpectro - shortcut for “Radio Frequency Spectrometer”, the equipment

Besides the proposed key generation rule any other string of arbitrary length will suffice as long as it contains an @ sign like for example an email address.

Furthermore any digital signature can be taken as a component of the identificator. However, it has to be taken care for special characters. Whenever a special character is used as part of an identifier it has to be escaped by a preceding back slash character. This does not apply to an @ sign which has to occur alone since this makes the string an identifier.

Address data type

The purpose of an address is to avoid replication of items inside a FTLight file/stream by establishing a reference to already existing items. The scope of an address is limited to the file/stream where it is contained in.

Theoretically it would be possible to refer to items in other FTLight files/streams from inside a current file/stream because of the uniqueness of information based on identifiers on the root level. Despite of this possibility it is not recommended to do so since this would result in relying on a fixed, hard-wired structure of information.

When looking into the history of multiple decades of information representation in structures it becomes visible that each structure definition gets outdated after some time and will be adjusted, extended or completely redefined in order to satisfy changing needs. Thus continuing with a hard-wired structure definition provides to relatively short-term data structures.

In opposite the FTLight specification aims at long-term data structures that can easily be adjusted, extended or even partly redefined without breaking links to previously defined structures. However, this requires using full path information instead of addresses when referring to external data.

Address layout

An address is composed of integer numbers connected by a “– “ (minus sign) where the sequence of integer numbers refers to an item’s position in a file/stream information hierarchy. Addresses represent links to the items to which they point to.

Example:

0-0-1-0

An address can be used for example as the first item in a line whenever the items in that line have to be connected to a parent item that could not be reached by implicit FTLight rules.

Example:

```
EKD@JO64qc.RSpectro,1073217600:FTLight,2004-01-12
,Antenna,dish 90cm
```

is equivalent to

```
EKD@JO64qc.RSpectro,1073217600
,Antenna,dish 90cm
0-0:FTLight,2004-01-12
```

and translates in both cases into the following structure

```
0           EKD@JO64qc.RSpectro
0-0            1073217600
0-0-0             FTLight
0-0-1             2004-01-12
0-1            Antenna
0-1-0             Dish 90cm
```

Change management

Addresses can also be of advantage if an existing structure needs to be expanded without breaking down running applications that work with and rely on those structures, e.g.

Old format:

```
Frequency:GHz,10.600
```

New format:

```
Frequency:GHz,10.600,start,step,end,default
0-2,10.500
0-3,0.00025
0-4,12.750
0-5,0-1
```

The structure of the new format would look like follows:

```
0                Frequency
0-0                GHz
0-1                10.600
0-2                start
0-2-0                10.500
0-3                step
0-3-0                0.00025
0-4                end
0-4-0                12.750
0-5                default
0-5-0                10.600
```

The last value (0-5-0) would be the same as (0-1) since a link is pointing to the latter. This allows for staying compliant with the previous structure definition which had the default frequency on position (0-1). All applications which take the default frequency from that position will continue to work also with the new information structure. Additionally, new applications can take advantage of the extended set of information which provides for a start, step and end frequency in addition to the default frequency.

Thus the example shows a second method how to stay compliant with older definitions where the new structure definition takes care about not breaking the link to existing applications. Together with the previously explained approach of using full path names instead of links wherever this is possible, this provides to a practical solution for changing data structures. Those can be maintained over long periods of time and for multiple generations of applications which work with structures that are adjusted, extended or partly redefined whenever those requirements evolve.

The ability to adjust, extend and partly redefine structures is considered to be of extreme importance in radio astronomy because of the long-term character of the data that this area is dealing with. A database entry showing business details of a company might be of low or no interest at all after a couple of decades since that company may have disappeared at that time. The opposite is the case with astronomical data. Having data from a radio source that was collected a couple of decades ago might be key for establishing dynamic models of that radio source. Thus the data has to be maintained in a readable form over a very long time. The FTLight specification is committed to that goal.

Framework functionality

Although the main purpose of a data collection is assembling real data it might be useful to have additional information available that describes the content of particular data fields. Those data are called metadata. Since a FTLight file itself usually contains various metadata, the description of those fields could be called meta-metadata. Instead, in the context of the FTLight specification that information will be referred to as Framework.

Optional values

Framework information will be started by empty data fields as they are generated for example by a double colon '::' after a path specification. The items which follow that double colon will become the optional values which can be filled into the empty data field at the superior level.

For example:

```
source::Sun,Crab nebula,3C353
```

The values Sun, Crab nebula and 3C353 are the optional values that can be assigned to the subordinate collection of the 'source' item.

Comments

A comment will be identified by two empty data fields being at adjacent levels, for example:

```
source:::This is the name of the observed source.
```

Keywords as comment can be used to specify recommendations for the target field, for example:

```
source:::limit:chars:80
```

The recommendation for the target field is to contain at most 80 characters.

Default values

Whenever an item is attached to a chain of optional values that item becomes the default value and the chain of optional values will be inserted as a hierarchy of real data items into the target location. However, the default value can later be overwritten by inserting another optional value as a real data field into the FTLight file, for example:

```
observation::source:Sun
observation::source::Sun,Crab nebula,3C353
…
observation:source:Crab nebula
```

First a default value 'Sun' as well as three options are specified for the 'source' item. Later the default value is overwritten by 'Crab nebula'.

Duplicate specifications

In case of multiple uniform data structures it might be useful to specify framework data only once. For example, if multiple channels exist that have the same data structure then the item that contains the channel number can be left empty which extends the following specifications to all channels:

```
Channel,::sample rate:Hz
```

Interpretation

A Framework will usually be defined in a special framework file which can be included into other FTLight files by reference. The content of a Framework is intended for supporting data collection, e.g. by providing useful default and optional values as well as by explaining the meaning of data fields and how to fill them. It will never constrain information on any level although it might contain certain recommendations for constraining items e.g. for the number of characters as in

```
source:::limit:chars:80
```

However, It is up to the user of the Framework to adhere to those constrains or not.

Data integrity

The purpose of a data integrity support is to allow for detecting data corruption on the one hand and to provide a concept for retrieving as much as possible valid data from a corrupted file/stream on the other hand.

The data integrity support in a FTLight file/stream is based on a line as the smallest unit where data corruption can be detected. Therefore a checksum can be added to the end of a line which is calculated from all characters in front of the checksum plus a line number which replaces the checksum temporarily.

The checksum will be written as a binary field starting with a semicolon as field separator in front of it. Whenever there is a binary field in the end of a line it has to be the checksum.

On the other hand it is not forced to include a checksum. However, in this case the last field in a line must not contain a Binary item.

Checksum calculation

The checksum is based on one or multiple symbols of the Binary data format. Dependent on the number of binary symbols in the last field of a line the base of the checksum will be 216 to the power of that number, e.g. 216*216 = 46656 in case of two symbols or 10077696 in case of three.

The number of symbols should be chosen based on the size of a line. It can vary from one line to the other. For short lines with text only one symbol might be sufficient. In case of many Gigabytes of data in a single item, e.g. from an audio/video file or when wrapping another huge FTLight file as binary data in an item, it might be recommended to take multiple symbols as checksum. There is no conceptual limit on the number of symbols in a checksum.

After establishing the number of symbols the checksum will be calculated as the remainder when dividing the complete line by the base of that checksum. Before performing that calculation the checksum field will be filled with the line number beginning with one for the line which contains an identifier at the first position.

Example: Single symbol checksum

```
,Data;7
```

Character	Value	Remainder
,	44	44 % 216 = 44
D	68	(44*256+ 68) % 216 = 100
a	97	(100*256+ 97) % 216 = 209
t	116	(209*256+116) % 216 = 52
a	97	(52*256+ 97) % 216 = 17
;	59	(17*256+ 59) % 216 = 91
7	55	(91*256+ 55) % 216 = **23**

The character corresponding to symbol 23 will be 23 + 32 = 55 accordingly to the previously described rules for generating the Binary data type. That value is equivalent to the ‘7‘ character. Thus the line with checksum would look like follows (be unchanged in this special case):

```
,Data;7
```

The receiver of that line will do the same calculations and will be able to compare the result to the checksum that is written in the end of the line. Before performing the calculations the receiver needs to replace the checksum with the line number generated on the receiver’s site. Thus the line number will also be checked without being actually transferred.

Corrupted data retrieval

Occasionally it may happen that a single byte or a series of bytes in a file/stream get corrupted. While the checksum allows for detecting corrupted lines it will not help to reconstruct the previous content. Therefore it will depend on the valid lines if some information can be restored. Given the current path in the FTLight file/stream was not changed by the corrupted line all other data will keep their positions in the information hierarchy and thus can be used further. Only the corrupted data lines will be lost in this case.

However, if the corrupted line changed the current path in the FTLight file/stream all subsequent information may go to the wrong place if the line after the corrupted one used the current path. In this case only human editing can help to restore the valid data lines properly by opening the FTLight file/stream and editing it in a text editor as far as the intended structure is known or can be derived.

FTLight repository

When a computer generates a FTLight file or receives a FTLight stream from another computer the content of the FTLight file/stream will usually be stored on hard disk. The following repository structure allows for storing FTLight files on hard disk without worrying that any previous files get overwritten.

The key for getting to unique path/file names is the creation timestamp of a FTLight file/stream. As a general rule the creation timestamp on position "0-1" always follows an identifier on position "0-0" in each FTLight file/stream. This is one of the predefined characteristics of a FTLight file/stream where functionality is based on and thus it must be preserved forever.

The folder hierarchy in a FTLight repository is defined as follows:

1 – FTLight
2 – Year
3 – Month
4 – Day
5 – Hour
6 – Minute
7 – Seconds
8 – Milliseconds
9 – …

Dependent on the time period that is spanned by a FTLight file the actual file will be for example in a 'Seconds' folder or on a higher or lower level. The file names are constructed as follows:

'Year'-'Month'-'Day'_utc'Hour'h'Minute'm'Seconds's.'Milliseconds'_Identifier.csv

Under the assumption that for any given identifier there will only be one file at a time the above filename will be universally unique. The extension .csv allows for opening that file conveniently in an Excel spreadsheet for review.

Example: FTLight repository with a 6 minute and a millisecond time period file

```
FTLight
  2004
    Jan
      20th
        utc06
          06m
          12m
          18m
          24m
            2004-01-20_utc06h24m_EKD@JO64qc.RSpectro.csv
          31m
            47s
              719ms
                2004-01-20_utc06h31m47s719ms_EKD@JO64qc.RSpectro.csv
```

In addition to the uniqueness of each file in the above structure another advantage is that information can be found and reviewed easily. Furthermore information can be taken out in any portion and send over by conventional means like diskette, CD, DVD, ftp file or any other medium or channel to another party.

Example:

Sending over all observation data from a given day to another party would require to zip the folder of that day and to send it over by any file exchange medium or channel. The receiver would drop it into the FTLight repository since the newly received files will be unique and no danger exist that any existing files get overwritten.

Version tracking

Information can be incorrect or uncompleted at creation time or it can change with time. Therefore a method is needed that allows for implementing corrections, extensions and changes. Since the creation timestamp of existing information must not change an additional requirement emerges from maintaining multiple versions of a given FTLight file in the repository.

The main tool to add versions of existing files to the FTLight repository is to assign a new timestamp to the zero position in the sub-collection of the current timestamp. Every subsequent version will do the same and will assign a new timestamp to the zero position in the sub-collection of the current timestamp. The identifier of the operator who introduced the change will also be written to the new sub-collection which contains already the new timestamp at the first position.

Example:

```
EKD@JO64qc.RSpectro,1073217600
,Frequency:GHz,10.600
,Bandwidth:KHz,250
```

The frequency information will be changed from 10.6 GHz to 10.55 GHz by an operator with the identifier kantz@wegalink.com about two weeks later:

```
EKD@JO64qc.RSpectro,1073217600,1075123807
,Frequency:GHz,10.600
,Bandwidth:KHz,250
0-0:kantz@wegalink.com,0-2,Frequency:GHz,10.55
```

The second information block can be read as follows:

- This information belongs to the identifier EKD@JO64qc.RSpectro
- There is an initial information available with the timestamp 1073217600
- A changed version has been created with the timestamp 1075123807
- The initial information had a Frequency: GHz, 10.600 and a Bandwidth: KHz,250
- The operator who introduced a change has the identifier kantz@wegalink.com
- The changed version has an unchanged bandwidth (address 0-2)
- The changed version has a Frequency: GHz, 10.55

Summarizing the version handling procedure one can say that a new version of existing information will be created by extending the path to the current timestamp with a new timestamp. Subsequently the identifier of the operator who introduced the change will be added. Further all unchanged items as well as the changed items will be added to the added collection.

Multiple versions in the repository

A change to an existing file can be made using the same identifier but also using any other identifier. This means that any operator can change any information for any identifier.

The storage of changed information will be done same as before using the timestamp and the identifier of the operator who generated the changed information. Additionally a link is added to the position where the original file is stored. In order to avoid any conflicts the links to changed versions of a file will be written into a subdirectory with the same name as the file but without using the extension, like e.g. .csv.

The decision to use the original information or any of the changed versions is up to the user of that information. A user being interested in the most recent information will have to look first for a folder with the same name as the requested file but without the extension, like e.g. .csv. In case there is such a folder the links found in that folder will be searched for the most recent one.

Example:

```
FTLight
  2004
    Jan
      12th
        utc06
          18m
            2004-01-12_utc06h18m_EKD@JO64qc.RSpectro.csv
            2004-01-12_utc06h18m_EKD@JO64qc.RSpectro
              Link to: FTLight/2004/Jan/26th/utc13/31m/47s/
                           2004-01-26_utc13h31m47s_kantz@wegalink.com.csv
      26th
        utc13
          31m
            47s
              2004-01-26_utc13h31m47s_kantz@wegalink.com.csv
```

The example contains an original file which was generated on January, 12th by an operator's equipment with the identifier EKD@JO64qc.RSpectro. On January, 26th it was changed by an operator with the identifier kantz@wegalink.com. A link to the changed information was established in a subfolder at the position of the original file.

For determining if a given file/stream contains the original information or changes to previous information the timestamp at position 0-0 will be examined. If it has a sub-collection attached and if there is another timestamp on position 0 of the sub-collection it is changed information. Otherwise it is the original information.

Request/Response procedure

First the event of sending a request and receiving a response will be described for which subsequently a support will be defined:

- An initiating party issues a request for a subset of data that belongs to a given identifier
- A responding party has the desired information and sends it over to the initiator
- The initiator adds the received information to its FTLight repository
- The initiator can send requests and receive responses with a high frequency
- A response will always be the original file with an unchanged timestamp or a subset of it
- For different requests multiple responses can arrive for the same original file
- Multiple original files respectively their subsets can arrive in response to a single request

Storing request/response streams

The procedure of sending requests and receiving responses has been defined as a high frequency process. Usually there can be multiple requests/responses per second dependent on the bandwidth between the computers. Therefore storing request/response streams need to be done with a small time slot, for example one millisecond.

Furthermore it has been defined that multiple responses can arrive for the same original file from another computer, for example when first requesting only a small subset of data and later asking for the complete file. All of those copies will have the same identifier and the same timestamp since they were derived from the same original file. Therefore multiple copies need to be maintained in order not to overwrite the file at the position which the original file occupies.

This will be achieved by storing all responses in the same directory as the request file. That file will have a unique location since the time slot for those files has been chosen based on that assumption. When a new response was received and stored subsequently a link will be established in the file system at the position that was occupied by the original file and that link will point to the last received copy of the original file respectively a subset of it.

The described procedure will not make problems as long as each subsequent response will contain more data out of the original file than the previous responses. Otherwise some data will be lost on the file's original position. However, it will be preserved in the copies of that file. Since this is a special case which can not be solved by the described procedure the initiator is responsible for handling that situation. One possibility is to send requests for an ever growing subset of data or otherwise the initiator could choose to merge the received response files.

A FTLight repository with some request/response data in it could look like follows:

```
FTLight
  EKD@JO64qc.RSpectro
    2004
      Jan
        20th
          utc06
            00m
              Link to: FTLight/2004/Jan/20th/utc17/31m/47s/719ms/
                           2004-01-20_utc06h00m_EKD@JO64qc.RSpectro.csv
            10m
              Link to: FTLight/2004/Jan/20th/utc17/31m/47s/719ms/
                           2004-01-20_utc06h10m_EKD@JO64qc.RSpectro.csv
          utc17
            31m
              47s
                719ms
                  2004-01-20_utc17h31m47s719ms_EKD@JN58ve.Lyra.csv
                  2004-01-20_utc06h00m_EKD@JO64qc.RSpectro.csv
                  2004-01-20_utc06h10m_EKD@JO64qc.RSpectro.csv
```

In the above repository example a request file from an initiator with the identifier EKD@JN58xf.Lyra has been sent out in order to get measurement data from the identifier EKD@JO64qc.RSpectro. The response consisted of two files from the information hierarchy below the EKD@JO64qc.RSpectro identifier and those files have been put into the same directory where the request file is located. Subsequently, links were established in the file system in order to make the files available on the original position where they are supposed to be based on their timestamps.

Thus the request/response procedure offers writing copies of remote FTLight repositories to the local FTLight repository same as they would be transferred by a conventional storage medium. All communication to remote computers is based on FTLight streams which are preserved in the FTLight repository and can thus easily be reviewed.

Request structure

Requesting information from another party will consist of general elements that identify a request and it will contain additional elements that describe the information which is wanted in detail. The detail request information has to be agreed between communicating parties whereas the general elements will always apply.

General request elements

The QUERY item that was specified in the scope of the FTLight specification (96, back apostrophe) will be defined as the general element for a request as soon as it appears at the position 0 in any collection. In this case the initiator wants to receive the complete collection which is available in the FTLight repository of the responding party along with all sub-levels. The identification of the initiator will be at position 0 in the subordinate collection to the general request item.

Example:

```
EKD@JO64qc.RSpectro,`,EKD@JN58xf.Lyra
```

The initiator with the identifier EKD@JN58xf.Lyra wants to receive all information which is available under the EKD@JO64qc.RSpectro identifier. Since this might be a huge data volume some restrictions related to the time period could be added to the request in order to limit the amount of data for the expected answer. However, this belongs to the additional elements that have to be used in agreement by both communicating parties.

A general element for constraining a request is to put wanted items into the collection with the QUERY item at the first position. This constrains the request to the subset of data that is below those items in the information hierarchy.

Example:

```
EKD@JO64qc.RSpectro:`,Frequency,Bandwidth
0-0,EKD@JN58xf.Lyra
```

In the above example only the frequency and the bandwidth are requested. This would exclude the huge amount of measurement data which is organized for example under the Data item, however, all files would be returned where a Frequency and a Bandwidth item are available on the second level just below the identifier. This still might be a huge amount of files and thus also in this case a time constraint would be useful.

A running response stream can be stopped at any time by the initiator. The general element to stop a response stream is to put a QUERY item in the place of the initiator with the initiator's identifier on the 0 position of the subordinate collection to the Stop element.

Example:

```
EKD@JO64qc.RSpectro,`,`,EKD@JN58xf.Lyra
```

The above example stops the data stream coming from the responder's FTLight repository with the identifier EKD@JO64qc.RSpectro and going to the initiator's site with the identifier EKD@JN58xf.Lyra.

Request for identifiers

A single back apostrophe followed by an initiator's identifier requests a list of all identifiers. The response will consist of one entry for each identifier along with the earliest available timestamp.

Example:

```
`,EKD@JN58xf.Lyra
```

The answer to the above request for identifiers could look like follows:

```
EKD@JO64qc.RSpectro,1073212000
```

That response informs the initiator about the availability of data for the EKD@JO64qc.RSpectro identifier beginning at a time given by the timestamp 1073212000 in seconds after January 1st, 1970 in UTC.

Time constraint

A time constraint belongs to the additional elements in the request/response procedure which has to be agreed between both communicating parties. A structure could be as follows:

```
EKD@JO64qc.RSpectro,`,EKD@JN58xf.Lyra
0-0:begin:utc,1073217400
:end:,1073217800
:step:sec,20
```

In the above example data from a 400 second time period is requested and samples are expected to follow each other with a 20 second time step. Those data could be used e.g. as preview data.

Appendix A

A.1 Example for storing multi-level metadata in front of interferometry data

```
MCL@JN76ec.SIDI,1108598400:FTLight,2005-03-03
,global metadata tag1:item1,item2,...
,global metadata tag2:item1,item2,...
,global metadata tagN:item1,item2,...
,metadata for channel1,metadata tag1: item1, item2,...
,,metadata tag2:item1, item2,...
,,metadata tagN: item1, item2,...
,metadata for channel2,metadata tag1: item1,item2,...
,,metadata tag2: item1,item2,...
,,metadata tagN: item1, item2,...
,metadata for channelN,metadata tag1: item1, item2,...
,,metadata tag2: item1,item2,...
,,metadata tagN: item1, item2,...
,correlation values for baseline 1
:Time,Correlation
:1108598400.123,0.9876
:1108598400.124,0.9875
:1108598400.125,0.9877
...
,correlation values for baseline 2
:Time,Correlation
:1108598400.123,0.9836
:1108598400.124,0.9835
:1108598400.125,0.9837
...
,correlation values for baseline 3
:Time,Correlation
:1108598400.123,0.3476
:1108598400.124,0.3475
:1108598400.125,0.3477
...
```

A.2 Example for a multi-channel receiver with random channel selection

Frequency channels are sampled randomly and in non-regular time intervals. In this case the full time and frequency information has to be provided along with each signal value. This could be done as follows for a first baseline and in a similar way for other baselines:

```
EKD@JN58ve.RSpectro,1108598400:FTLight,2005-03-03
,Data,baseline1:[m],12.35
:Time,Frequency,Signal strength
:[Seconds since 1970-01-01],[GHz],[0..4095],@
:1109462400.111,10.610,2745
:1109462400.239,10.670,2745
:1109462400.377,10.655,2745
```

A.3 Example for a multi-channel receiver with regular channel selection

A pre-defined set of frequencies is sampled in regular time intervals. In this case it is sufficient to provide the used frequency vector only once and to store only the start time of a blok of samples. This could be done as follows:

```
EKD@JN58ve.RSpectro,1108598400:FTLight,2005-03-03
,Data,baseline1:[m],12.35
:Time,Freq1,Freq2,Freq3,Freq4,Freq5
:[Seconds since 1970-01-01],[GHz],[GHz],[GHz],[GHz],[GHz]
:equal sample time intervals,10.630,10.635,10.640,10.645,10.650,@
:1109462400.100,2736,2850,2473,2945,2791
:1109462500.100,2335,2457,2272,2628,2437
:1109462400.100,2533,2593,2311,2593,2692
```