RTSA File Format

Features and Purpose

The RTSA file format is used to store files generated by the RTSA Suite. The file format is binary a chunk based, similar to e.g. the PNG file format.

The file format offers the following distinguishing features:

- Binary, compact and optionally compressed storage of measurement data
- Meta data storage (location, time, format etc.)
- Multi streams
- Interleaving of streams
- Sequential stream read and write
- Random access of complete files
- Preview storage of power spectra and power histogram
- Extension of existing streams

File Structure

Chunk Structure

The file is composed of individual and optionally recursive chunks. Each chunk starts with a chunk header:

```
struct DSPStreamFileChunk
{
        quint32         mChunkID, mChunkSize, mChunkFlags;
        quint16         mVersion, mHeaderSize;
};
```

The chunkID is a 32 bit code comprised of four ASCII letters. The chunk size (which includes the header) can be used to skip unknown or ignored chunks and progress quickly through the file. A chunk is comprised of a header and actual data, which may be either another series of chunks or binary data, based on the type of chunk.

The version field indicates incompatible of versions of chunks. If the header size in a chunk is larger than the known size of the reading application, the superfluous data may be safely skipped – if it is smaller, the additional fields can be safely assumed to have a default value (usually zero). Incompatible layout of a chunk is indicated by a different version number. With this scheme, it is painless to add new fields to chunks without creating incompatible versions.

General Data Types

Data is stored in little endian format.

Times are always stored in 64bit floating point doubles, relative to the start of the Unix epoch (January first 1970, 12am) or the start of the stream.

All offsets in the file are 64 unsigned integers.

Strings are stored as UTF8 and padded with zeros.

Generic File Layout

An RTSA file starts with a DSFH (File Head) chunk and ends with a DSFT (File Tail) chunk. Extending an existing file will result in multiple DSFH/DSFT pairs. Chunks in the file can be read forward using the chunk size or with random access using offsets stored in other chunks. All offsets are 64bit, relative to the file start and are backwards. There are no forward references, thus one can easily stream an RTSA file.

Basic file structure could be:

DSFH File Header
- **STRM** Stream Head
  - **ANTA** Antenna
  - **SSTR** Sub Stream
  - **SAMP** Samples
  - **SAMP** Samples
  - **SAMP** Sample
- **STRT** Stream Tail

DSFT File Tail

Other references are based on 64 bit IDs, eg. Stream ID, Sub Stream ID or Antanna IDs. All objects bearing an ID are placed in the file before they are used. They are also part of backward linked chains for retrieval during random access.

Chunks

File Head DSFH

The DSFH chunk starts a new independent segment in an RTSA file. All IDs are considered invalid a the start of a new file header chunk.

```
struct DSPStreamFileChunkHead : public DSPStreamFileChunk
{
        double          mCreationTime;
};
```

mCreationTime	File creation time relative to the epoch

File Tail DSFT

The DSFT terminates a file segment. This is usually the second chunk visited during random access or the last chunk during streaming.

```
struct DSPStreamFileChunkTail : public DSPStreamFileChunk
{
        double          mCompletionTime;
        qint64          mStreamOffset;
        quint32         mNumStreams;
```

```
};
```

mCompletionTime	File completion time relative to the epoch
mStreamOffset	Offset of the tail of the last stream in the file
mNumStreams	Number of streams in the file

Stream Head STRM

The stream head chunk indicates the start of a new stream in the file.

```
struct DSPStreamFileChunkStreamHead : public DSPStreamFileChunk
{
        quint64         mStreamID;
        double          mStartTime;
        qint64          mStreamOffset;
};
```

mStreamID	Unique 64bit ID for this stream
mStartTime	Start time of this stream relative to the epoch
mStreamOffset	Offset of the tail of the previous stream in the file

Stream Tail STRT

The stream tail chunk ends a stream. It includes offsets to the start of the stream and the other stream meta data elements such as sub streams and antennas.

```
struct DSPStreamFileChunkStreamTail : public DSPStreamFileChunk
{
        qint64          mStreamOffset, mSubStreamOffset, mPreviewOffset;
        quint64         mNumSamples, mPayloadSize;
        quint32         mPreviewLevels, mNumPreviews, mNumPreviewSegments;
        double          mEndTime;
        qint64          mAntennaOffset;
        qint64          mMetaDataOffset;
};
```

mStreamOffset	Offset of the stream head chunk
mSubStreamOffset	Offset of the last sub stream chunk
mPreviewOffset	Offset of the last preview chunk
mNumSamples	Number of samples in this stream
mPayloadSize	Total payload size in bytes for this stream
mPreviewLevels	Number of preview hierarchy levels
mNumPreviews	Total number of previews
mNumPreviewSegments	Total number of preview segments
mEndTime	End time of this stream relative to the stream start time (aka stream duration)
mAntennaOffset	Offset of the last antenna chunk
mMetaDataOffset	Offset of the last meta data type chunk

Sub Stream SSTR

A sub stream chunk contains the common meta data for a series of samples, such as frequency bounds, rates, types, orientations etc. A stream may contain any number of sub streams, due to e.g. antenna movement or multi segment antennas.

```
struct DSPStreamFileChunkSubStream : DSPStreamFileChunk
{
        quint64         mStreamID;
        quint32         mSubStreamID;

        qint64          mSubStreamOffset

        double          mFrequencyStart;
        double          mFrequencyStep;
        double          mFrequencySpan;

        double          mValueMinimum;
        double          mValueMaximum;

        double          mDirection;
        quint32         mAntennaIndex;
        quint32         mNumCategories;

        char            mName[128];

        quint64         mAntennaID;
        quint64         mMetaDataID;
};
```

mStreamID	The ID of the parent stream
mSubStreamID	The stream unique ID of this sub stream
mSubStreamOffset	Offset of the previous sub stream of this stream
mFrequencyStart	Start of the frequency range
mFrequencyStep	Sample rate or bin step
mFrequencySpan	Size of frequency range
mValueMinimum	Lowest value
mValueMaximum	Highest value
mDirection	Simple directional indicator
mAntennaIndex	Index of multi segment antenna
mNumCategories	Number of categories, if this is a sub stream with name indexed samples
mName	Name of this sub stream
mAntennaID	The ID of the antenna used to create this sub stream
mMetaDataID	Meta data type ID, if this is a structured data sub stream

The payload of a sub stream chunk contains the sub stream category chunks.

Sub Stream Category SSCA

A single category in a category sub stream. A category is a named scalar measurement, e.g. a channel power or a detection probability.

```
struct DSPStreamFileChunkSubStreamCategory : public DSPStreamFileChunk
{
	char		mName[128];
	quint32		mFlags;
	quint8		mRed, mGreen, mBlue, mAlpha;
	double		mStartFrequency, mEndFrequency;
};

const quint32 DSSCF_FREQUENCY_VALID	=	0x00000001U;
const quint32 DSSCF_COLOR_VALID		=	0x00000002U;
```

mName	Name of the category
mFlags	Category flags DSSFC_*
mRed	Red color value
mGreen	Green color value
mBlue	Blue color value
mAlpha	Alpha color value
mStartFrequency	Start frequency
mEndFrequency	End frequency

Antenna ANTA

Antenna chunks combine information of the physical and logical properties of the antenna used as well as geo information, such as location and direction.

```
struct DSPStreamFileChunkAntenna : public DSPStreamFileChunk
{
	quint64		mAntennaID;
	qint64		mAntennaOffset;
	char		mName[128];
	double		mLatitude, mLongitude;
	quint32		mFlags;
	quint32		mNumSegments;
	float		mTransform[4][4];
	char		mAntennaUUID[16];
};


static const quint32 DSPAF_LOCATION_VALID	=	0x00000001U;
static const quint32 DSPAF_TRANSFORM_VALID	=	0x00000002U;
static const quint32 DSPAF_DIRECTION_VALID	=	0x00000004U;
static const quint32 DSPAF_ROTATION		=	0x00000008U;
```

mAntennaID	Unique ID of the antenna
mAntennaOffset	Offset of previous antenna chunk in the stream
mName	Name of the antenna
mLatitude	Latitude of base antenna location
mLongitude	Longitude of base antenna location
mFlags	Antenna flags DSPAF_*

mNumSegments	Number of antenna segments
mTransform	Antenna transformation (e.g. rotation)
mAntennaUUID	Global unique ID of the antenna

The 64 bit antenna ID is used to indicate one antenna chunk, whereas the antenna UUID indicates the physical antenna. Thus moving an antenna would change the antenna ID but not the UUID.

The payload of an antenna chunk contains the antenna segment chunks.

Antenna Segment ANTS

A multi segment antenna contains a series of antenna segment chunks in its payload section.

```
struct DSPStreamFileChunkAntennaSegment : public DSPStreamFileChunk
{
        char            mName[128];
        float           mOrientation[4];
        quint32         mID;
};
```

mName	Name of the segment
mOrientation	Orientation of the segment in the antenna coordinate system
mID	ID of the segment

Meta Data Type MDTT

Structured data is stored in the file using a binary compression mode based on a meta data type. These types are stored themselves as binary compressed data in the payload section of meta data type chunks.

```
struct DSPStreamFileChunkMetaDataType : public DSPStreamFileChunk
{
        quint64         mMetaDataID;
        qint64          mMetaDataOffset;
};
```

mMetaDataID	Unique ID of this meta data type
mMetaDataOffset	Offset of previous meta data chunk

The structured data and its meta data type system is explained in its own chapter.

Preview SPRV

A preview chunk contains one histogram and several preview spectra, as well as offsets into the file for fast seeking. They are organized in a tree, where the tree height is determined by the number of previews in the file.

```
struct DSPStreamFileChunkStreamPreview : public DSPStreamFileChunk
{
        static const quint32 HistogramWidth   =       48;
        static const quint32 HistogramHeight  =       32;
        static const quint32 WaterfallWidth           =       128;
        static const quint32 SegmentsShift            =       4;
        static const quint32 Segments                 =       16;
```

```
        static const quint32 Samples                =       4096;

        quint8          mPreviewLevel, mPreviewCount;
        qint64          mPreviewOffsets[Segments];
        double          mPreviewTimes[Segments];
        quint64         mPreviewSamples[Segments];
};
```

mPreviewLevel	Level of this preview chunk in the hierarchy. Leave chunks of the tree have level zero
mPreviewCount	Number of preview elements in this chunk
mPreviewOffsets	Offsets of child preview chunks or the sample chunks for leave preview chunks
mPreviewTimes	Start times of the child preview chunks relative to the stream start time
mPreviewSamples	Start sample index numbers of the child preview chunks

The payload of the preview chunk may contain the preview information in unitless eight bit values.

```
struct DSPStreamFileChunkStreamPreviewData
{
        quint8          mHistogram[HistogramHeight][HistogramWidth];
        quint8          mWaterfall[Segments][WaterfallWidth];
};
```

Samples SAMP

Actual measurement data is stored in sample chunks.

```
struct DSPStreamFileChunkSamples : public DSPStreamFileChunk
{
        quint64                 mStreamID;
        quint32                 mSubStreamID;
        DPSStreamSampleType     mSampleType     : 8;
        DSPStreamSampleUnit     mSampleUnit     : 8;
        DSPStreamPayloadType    mPayloadType    : 8;
        qint32                  mCompression    : 8;
        double                  mPacketStartTime, mPacketEndTime;
        quint32                 mPacketFlags;
        quint32                 mSampleSize, mSampleDepth, mNumSamples;
};
```

mStreamID	ID of the parent stream
mSubStreamID	ID of the sub stream for this data
mSampleType	Datatype of the individual data elements
mSampleUnit	Unit used for the samples
mPayloadType	General payload type
mCompression	Compression or zero for lossless
mPacketStartTime	Start time of this chunk relative to stream start
mPacketEndTime	End time of this chunk relative to stream start
mPacketFlags	Packet flags DSPPF_*
mSampleSize	Size of an individual sample

mSampleDepth	Depth of a sample
mNumSamples	Number of samples in the packet

```
enum DPSStreamSampleType
{
        DSST_U8,
        DSST_U16,
        DSST_S16,
        DSST_U32,
        DSST_S32,
        DSST_F32,
        DSST_U8N,
        DSST_U16N,
        DSST_S16N,
        DSST_U32N,
        DSST_S32N,
        DSST_F32N
};
```

The sample type describes a single data element with its size (8, 16 or 32) its signed-ness (U or S) and whether it is integer or float (U/S or F). The extension N denotes packet storage, whereas all others are stored on 16 byte boundaries.

```
enum DSPStreamSampleUnit
{
        DSSU_GENERIC,
        DSSU_DBM,
        DSSU_PERCENTAGE,
        DSSU_DBM_HZ,
        DSSU_DBM_M2,
        DSSU_INDEX,
        DSSU_PHASE,
        DSSU_SIGNED_1,
        DSSU_UNSIGNED_1
};
```

The sample unit describes the physical unit and value range for an individual data element.

GENERIC	Generic floating point value
DBM	Decibel milliwatt
PERCENTAGE	Percentage 0..1
DBM_HZ	Decibel milliwatt per Hz
DBM_M2	Decibel milliwatt per square meter
INDEX	Integer index
PHASE	Phase from -π to +π
SIGNED_1	Signed floating point in the range – 1 to 1
UNSIGNED_1	Unsigned floating point in the range 0 to 1

The payload type specifies the high level sample data structure.

```
enum DSPStreamPayloadType
{
        DSPT_GENERIC,
        DSPT_AUDIO,
        DSPT_IQ,
```

```
        DSPT_SPECRTA,
        DSPT_DETECTION,
        DSPT_HISTOGRAM,
        DSPT_ENERGY,
        DSPT_VECTOR3,
        DSPT_STRUCTURED,
        DSPT_IQ_SLICE,
        DSPT_IMAGE
};
```

GENERIC	Generic numeric data
AUDIO	Audio samples
IQ	IQ samples, two values per sample
SPECTRA	Power spectra
DETECTION	Detection probability
HISTOGRAM	Histogram
ENERGY	Energy
VECTOR3	3D Vectors
STRUCTURED	Structured data using meta data types
IQ_SLICE	Slices of IQ samples
IMAGE	Grey scale image

Structured Data using Meta Data Type

Type of Types

The structured data types use a hierarchical type system with a small set of base types and three type constructors: fixed sized vectors, variable sized arrays and objects.

```
enum Type
        {
                MT_NONE,
                MT_BOOL,
                MT_INTEGER,
                MT_FLOAT,
                MT_STRING,
                MT_VECTOR,
                MT_ARRAY,
                MT_OBJECT
        };

static const quint32 DSSMTF_8BIT      =       0x00000001;
static const quint32 DSSMTF_16BIT     =       0x00000002;
static const quint32 DSSMTF_64BIT     =       0x00000004;

static const quint32 DSSMTF_SIGNED    =       0x00000010;

static const quint32 DSSMEF_RECURSIVE =       0x00000020;
```

A type object itself has five fields:

id	U64	ID of this type
type	U8	Type enum for basic type or type constructor

flags	U32	Flags for this type DSMTF_*
count	U32	Number of elements or bitmask
elements	Array	Member elements

The elements array is used with objects and has the following type:

name	String	Name of the element
flags	U32	Flags for the element DSMEF_*
type	Object	The type of the element

This type of types has the type ID zero and forms the root of the type system.

A C type definition would look like this:

```
struct MetaType
{
	quint64		mID;
	Type		mType;
	quint32		mFlags;
	quint32		mCount;
	struct Element
	{
		QString		mName;
		quint32		mFlags;
		MetaType	mType;
	}	mElements[];
};
```

Storage Format

A simple numeric type is stored in little endian format, using the number of bytes denoted by its type flag (8, 16, 32 or 64).

The string type is stored using a 32bit number for the number of characters followed by a sequence of UTF8 characters.

Vectors are stored as a packed sequence of elements.

Arrays are stored with a 32bit size, followed by a sequence of elements.

Objects are stored with a 32bit mask, indicating non zero elements (starting with bit zero) followed by a sequence of non zero elements.

Examples

The examples assume type IDs starting at one (1)

Array of 16bit signed Integers

An array of simple types, stores the type of the base element in a single element child.

```
MetaType	ArrayOfInt = {1, MT_ARRAY, 0, 0, {{"", 0, {2, MT_INTEGER,
DSSMTF_16BIT | DSSMTF_SIGNED, 0, {}} }}};
```

The resulting binary sequence would thus be:

01 00 00 00 00 00 00 00 : mID 1
06 : mType MT_ARRAY
00 00 00 00 : mFlags 0
00 00 00 00 : mCount 0
00 00 00 01 : mElements size 1
00 00 00 00 : mName ""
00 00 00 00 : mFlags
00 00 00 1F : mType mask 11111
02 00 00 00 00 00 00 00 : mID 2
02 : mType MT_INTEGER
12 00 00 00 : mFlags 16Bit and signed
00 00 00 00 : mCount 0
00 00 00 00 : mElements size 0

Object of a 3D vector

Objects can store up to 32 named data elements. The child elements are stored in an array of objects, including the types. Types that have been defined before are stored with their ID only. This example uses a vector of 32bit floats with the elements x, y and z.

MetaType Vector3D = {1, MT_OBJECT, 0, 0,
{{"x", 0, {2, MT_FLOAT, 0, 0, {}}
{{"y", 0, {2, MT_FLOAT, 0, 0, {}}
{{"z", 0, {2, MT_FLOAT, 0, 0, {}} }};

The resulting binary sequence would thus be:

01 00 00 00 00 00 00 00 : mID 1
07 : mType MT_OBJECT
00 00 00 00 : mFlags 0
00 00 00 00 : mCount 0
00 00 00 03 : mElements size 3
01 00 00 00 78 : mName "x"
00 00 00 00 : mFlags
00 00 00 1F : mType mask 11111
02 00 00 00 00 00 00 00 : mID 2
03 : mType MT_FLOAT
12 00 00 00 : mFlags 16Bit and signed
00 00 00 00 : mCount 0
00 00 00 00 : mElements size 0
01 00 00 00 79 : mName "y"
00 00 00 00 : mFlags
00 00 00 01 : mType mask 00001
02 00 00 00 00 00 00 00 : mID 2
01 00 00 00 7A : mName "z"

```
00 00 00 00 : mFlags
   00 00 00 01 : mType mask 00001
      02 00 00 00 00 00 00 00 : mID 2
```

Seeking and Preview Data

All preview chunks of a stream form a tree of stream segments. Each node has up to 16 references to nodes in the next lower level. The lowest level references individual sample chunks.

Seeking by time or sample number is thus a three step process:

1. Read the stream tail and extract the preview root offset and the stream end time
2. Traverse the tree using the preview times, samples and offset fields of the nodes, starting from the root until you reach a leaf node
3. Linearly read and scan the sample chunks using the packet start and end time

The preview data in each preview chunk consists of a series of up to 16 power spectra and one histogram. The cover range of each spectra is given by the preview times, the cover range of the histogram is the full range of the preview chunk.

The preview data is comprised of eight bit unsigned integer values spanning the complete range from 0 to 255. It has no unit or scale and is intended for visual presentation of the stream content without the need to actually read the file. The recursive structure of the preview allows a quick presentation of the full stream or sections even for very large files.

Compression of Spectrum Data

Uncompressed spectra are stored as 32bit floating point numbers. The common unit is dBm (decibel milliwatt). Spectrum data can be compressed using a compression factor indicator from 1 to 31. The algorithm is based on wavelets, quantization and variable symbol lengths.

Wavelet Conversion

The first compression step is a trivial wavelet transform. It is performed on up to 16 spectra in one block.

It alternates between a compression step in time direction and a step in the frequency direction, until both are not divisible by two anymore. Only the low pass coefficients are recursively filtered.

The wavelet transform replaces the even indexed numbers with the sum of and the odd indexed numbers with the difference between the two samples. The results are multiplied by the square root of one half in each step to keep the numbers in range.

Quantization

All coefficients are then uniformly quantized using a quantization factor derived from the compression factor.

```
float   quant = 0.1f * (1 << (chunk.mCompression - 1));
```

Bit Packing

Bit packing uses a variant of the Rice Code to store the integer portion of the quantized coefficients. The number of leading zero bits indicates the size of the code, each leading zero bit increases the size of the residual by three bits. The remaining bits provide the residual values. Codes are therefore multiples of four bits, which simplifies parsing.

Code	Value	Code	Value
1000	+0	1001	-0
1010	+1	1011	-1
1100	+2	1101	-2
1110	+3	1111	-3
0100 0000	+4	0100 0001	-4
0100 0010	+5	0100 0011	-5
0100 0100	+6	0100 0101	-6
0111 1110	+35	0111 1111	-35
0010 0000 0000	+36	0010 0000 0001	-36
0010 0000 0010	+37	0010 0000 0011	-37
0011 1111 1110	+291	0011 1111 1111	-291
0001 0000 0000 0000	+292	0001 0000 0000 0001	-292
0001 0000 0000 0010	+293	0001 0000 0000 0011	-293
0001 1111 1111 1110	+2343	0001 1111 1111 1111	-2343

Decompression

Decompression is performed in the inverse order of compression:

1. Unpacking the required number of coefficients from the bitstream
2. Dequantization
3. Inverse wavelet transform

Sample Code

This section provides some sample code for the decompression step.

The `WaveTransformStep` performs one decompression step in the frequency (x/colums) or time dimension (y/rows). The coefficients are assumed to be compact. The sx and sy parameters specify the step size, the dxy parameter the offset between the two sections (half the step size in the appropriate direction).

```
void WaveTransformStep(quint32 sx, quint32 sy, quint32 dxy)
{
   for (quint32 y = 0; y < NumRows; y += sy)
   {
      for (quint32 x = 0; x < NumColumns; x += sx)
      {
         float  s = WaveBuffer[x + y * NumColumns];
         float  t = WaveBuffer[x + y * NumColumns + dxy];

         WaveBuffer[x + y * NumColumns       ] = SQRTHALF * (s + t);
         WaveBuffer[x + y * NumColumns + dxy] = SQRTHALF * (s - t);
      }
   }
```

```
}
```

`WaveDecompress` first determines the starting step size, then iterates alternatively in time and frequency domain.

```
void WaveDecompress(void)
{
  quint32step = 1;
  while ((NumRows & (2 * step - 1)) == 0) step *= 2;
  while ((NumColumns & (2 * step - 1)) == 0) step *= 2;

  while (step > 1)
  {
    step >>= 1;
    if ((NumColumns & (2 * step - 1)) == 0)
    {
      WaveTransformStep (2 * step, step, step);
    }
    if ((NumRows & (2 * step - 1)) == 0)
    {
      WaveTransformStep (step, 2 * step, step * NumColumns);
    }
  }
}
```

The transform step is the same in compression and decompression mode, only the order and step sizes are different.

Sample Files Analyzed

File Header

```
DSFH
44 53 46 48               mChunkID
18 00 00 00               mChunkSize
00 00 00 00               mChunkFlags;
01 00                     mVersion
18 00                     mHeaderSize
E0 96 2A ED 3A 1C 15 43   mCreationTime
```

Stream Header

```
STRM
53 54 52 4D               mChunkID
28 00 00 00               mChunkSize
00 00 00 00               mChunkFlags
01 00                     mVersion
28 00                     mHeaderSize
07 00 00 00 00 00 00 00   mStreamID
29 5C FF EC BE 22 D6 41   mStartTime
00 00 00 00 00 00 00 00   mStreamOffset - no prior stream packet, thus terminating
                          offset 0
```

Antenna

```
ANTA
41 4E 54 41               mChunkID
F8 00 00 00               mChunkSize
00 00 00 00               mChunkFlags
```

```
01 00                      mVersion
F8 00                      mHeaderSize
....
```

Sub Stream

```
SSTR
53 53 54 52                mChunkID
E8 00 00 00                mChunkSize
00 00 00 00                mChunkFlags
01 00                      mVersion
E8 00                      mHeaderSize
07 00 00 00 00 00 00 00    mStreamID
03 00 00 00                mSubStreamID
00 00 00 00 00 00 00 00    mSubStreamOffset - no prior substream packet
...
```

Sample packet

```
SAMP
53 41 4D 50                mChunkID
40 70 00 00                mChunkSize
00 00 00 00                mChunkFlags
01 00                      mVersion
40 00                      mHeaderSize
07 00 00 00 00 00 00 00    mStreamID
03 00 00 00                mSubStreamID
05                         mSampleType - DSST_F32
01                         mSampleUnit - DSSU_DBU
03                         mPayloadType - DSPT_SPECRTA
00                         mCompression - uncompressed
D9 39 6A D2 6D DD 50 40    mPacketStartTime
1D E7 BD Fa 7F DD 50 40    mPacketEndTime
00 00 00 00                mPacketFlags;
80 03 00 00                mSampleSize - 896 bins
01 00 00 00                mSampleDepth
08 00 00 00                mNumSamples - 8 spectra in the packet
91 C8 9A C2                 - first data sample
...
```

Stream Tail

```
STRT
53 54 52 54                mChunkID
58 00 00 00                mChunkSize
00 00 00 00                mChunkFlags
01 00                      mVersion
58 00                      mHeaderSize
18 00 00 00 00 00 00 00    mStreamOffset - offset of the STRM chunk in the file
38 01 00 00 00 00 00 00    mSubStreamOffset - offset of the last SSTR chunk in the
                           stream
30 D8 0B 01 00 00 00 00    mPreviewOffset
10 13 00 00 00 00 00 00    mNumSamples
00 E0 0A 01 00 00 00 00    mPayloadSize
01 00 00 00                mPreviewLevels - small file, thus single level preview tree
06 00 00 00                mNumPreviews - 6 preview chunks in the bottom level
58 00 00 00                mNumPreviewSegments - 88 preview segments in the bottom
                           level
xx xx xx xx                 - padding
78 F8 D9 3D 29 08 51 40    mEndTime
40 00 00 00 00 00 00 00    mAntennaOffset - offset of the last ANTA chunk in the
                           stream
```

File Tail

```
DSFT
44 53 46 54                mChunkID
28 00 00 00                mChunkSize
00 00 00 00                mChunkFlags
01 00                      mVersion
28 00                      mHeaderSize
```

```
A0 FE 52 ED 3A 1C 15 43     mCompletionTime
01 00 00 00                 mNumStreams
xx xx xx xx                  - padding
```

Command Line File Utility

The RTSAFileTool command line utility can be used to inspect, repair or export rtsa files. It is part of the installation and can be found in the applications install folder.

Inspecting Files

The command line for inspecting files is:

```
RSTAFileTool info
   [-start=<starttime>] [-end=<endtime>]
   [-histo] [-preview[=<lines>]]
   file.rtsa
```

start	HH:mm:ss.zzz	Optional start time in the file
end	HH:mm:ss.zzz	Optional end time in the file
histo		Show histogram for selected range
preview	U32	Show preview lines of spectras
file.rtsa		Filename of source file

Repairing Files

The command line for repairing files is:

```
RSTAFileTool repair
   [-compress=<factor>]
   file.rtsa target.rtsa
```

compress	U32	Compression factor to be used in target file
file.rtsa		Filename of source file
target.rtsa		Filename of target file

Exporting Data from Files

The command line for exporting data from files is:

```
RSTAFileTool export
   [-start=<starttime>] [-end=<endtime>]
   [-compress=<factor>]
   [-format=<csv|rtsa|xml|excel|dat|xml|json|wv|asc|mat|iq>]
   file.rtsa [target.csv]
```

start	HH:mm:ss.zzz	Optional start time in the file
end	HH:mm:ss.zzz	Optional end time in the file
compress	U32	Compression factor for target files
format		Output format
file.rtsa		Filename of source file
target.csv		Filename of target file, or stdout if no filename is provided