Discover the Experience

Streaming Library User's Manual

Library Version 2.9.0

Streaming Library User's Manual

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 DE-Erlangen, Germany
+49-9131-7701-0
+49-9131-7701-6333
info.automotive@elektrobit.com

Technical support

EB Assist ADTF Support

Phone: +49-9131-7701-7777

http://automotive.elektrobit.com/support

Contents

1 Introduction

The *ADTF Streaming Library* is used for read and write access to files compatible to the ADTF *Harddisk Recorder*, so called *DAT Files*. The *Streaming Library* allows you loading *DAT Files*, selecting a single or all *Streams* in the *DAT File* and reading the saved data thereof. It also allows you to create new *DAT Files* an to populate them with data.

1.1 Installation

If you are using the *ADTF Streaming Library* on a Windows 7 operating system, you should not install it into the "ProgramFiles" or any other "System" directory. Because of the user account control (UAC) you have to have administrative privileges to run CMake and/or the batch-file to build the examples. Otherwise the build will fail every time.

It is not possible to use the silent installer for automated platform-mixing installations (*ADTF Streaming Library* 32 bit on 64 bit operating systems).

1.2 Dependencies

No additional libraries or programs are required apart from the ones delivered with the *Streaming Library*. No other dependencies to ADTF exist.

1.3 Further documentation

This user's manual is not a code reference but an additional description of the concepts and comment to the usage of the *Streaming Library*. The code reference is found in a seperate document called *SDK*

Documentation (see `<InstallDir>/doc/streaminglib.chm`).

The *DAT File Format* is explained in the document `DataFileFormatSpecification.pdf`.

1.4 Examples

Additionally, eleven example applications are included which demonstrate the usage of the *Streaming Library* (see `<InstallDir>/examples`):

- additionaldata
- candump
- canwriter
- copychunks
- decode_compressed_video
- dummywriter
- eds
- encode_compressed_video
- fileaccess
- mediadescription

1.5 Special text formats and symbols

This software guide uses special text formats and symbols to indicate important elements and facts, as shown here:

Windows, **Dialogs** and other **elements of the user interface**

`File names`, `directory names`, etc.

Cross reference section 1.5 Special text formats and symbols

URLs: www.url.com

`inline code`

```
Source code
```

Properties

Proper names

Warnings indicate potential error sources.
→ The arrow indicates the steps you have to take to prevent an error.

This note symbol indicates useful information.

Tips provide additional information.

2 Utilization

2.1 Approach

To utilize the *Streaming Library* you can use *CMake*. An introduction to *CMake* can be found in the *SDK Documentation*.

Include the file `adtf_streaming.h` in your project. In the directory of your application you need the following file additionally (see Table 2.1 Needed DLLs respective shared objects):

Platform	Debug	Release
Windows	`adtfstreamingD_<Ver>.dll`	`adtfstreaming_<Ver>.dll`
Linux	`libadtfstreamingD_<Ver>.so`	`libadtfstreaming_<Ver>.so`

Table 2.1: Needed DLLs respective shared objects (`<Ver>` = Library Version)

These files are delivered with the *Streaming Library*.

In principle you will work with the following classes:

- A file reader to access the *Streams* in the *DAT Files* (see section 2.2 FileReader: IADTFFileReader)
- Media descriptors which describe the *DAT File* (see section 2.3 MediaDescriptor: tADTFMediaDescriptor)
- Stream descriptors, which describe a *Stream* (see section 2.4 StreamDescriptor: tADTFStreamingDescriptor)
- Data blocks, which compose a *Stream* (see section 2.5 Data block: cADTFDataBlock)
- Chunk copy to copy video stream from the source file to the destination file (see section 2.6 Chunk Copy: cChunkCopy)

- A file writer to create and write to new *DAT Files* (see section 2.7 FileWriter: IADTFFileWriter)

The following media types and media samples are currently supported out of the box:

- adtf.core.media_type (only major and subtype information)
- adtf.type.audio (only adtf::tWaveFormat is possible for this sample)
- adtf.type.video (only adtf::tBitmapFormat is possible for this sample)
- adtf.type.video_compressed
- adtf.core.media_sample (raw data, these are used for video samples as well)
- adtf.sample.can_message_raw (only adtf::tCanMessage is possible for this sample)
- adtf.sample.can_message_ext_raw (only adtf::tCanMessageExt is possible for this sample)
- adtf.sample.can_message (only adtf_devicetb::tCANData is possible for this sample)

Note that you can add further types by loading the appropriate services via the `IADTFFileReader::AddTypeService` method, please have a look at the API documentation.

2.2 FileReader: IADTFFileReader

Typically the following steps are made when working with the `IADTFFileReader`:

1. Create an instance of the class `IADTFFileReader`:

   To read a *DAT File* you need an instance of the `IADTFFileReader`. You create one with the static method `IADTFFileReader::Create()`:

   ```
   IADTFFileReader *pFileReader = IADTFFileReader::Create();
   ```

2. Open the *DAT File*:

   A *DAT File* is opened with the method `IADTFFileReader::Open(...)`.

3. Gather information about the *DAT File* and the contained *Streams*:

   The `IADTFFileReader` provides the number of *Streams*, the length of the *DAT File* and the number of different blocks inside the *DAT File*:

   ```
   tInt nStreamCount = 0;
   pFileReader->GetStreamCount(&nStreamCount);
   tUInt64 n64DataBlockCount = pFileReader->GetDataBlockCount();
   tUInt64 n64IndexBlockCount = pFileReader->GetIndexBlockCount();
   tTimeStamp tsDuration = pFileReader->GetTimeDuration();
   ```

You can request the current position as well:

```
pFileReader->GetCurrentBlockPos();
pFileReader->GetcurrentIndexBlockPos();
```

Stream IDs are **not** null-based. If `GetStreamCount()` returns 2 then *Streams* with the IDs 1 and 2 exist.

The name of the *Streams* in a *DAT File* are unique. You get the name associated with a *Stream ID* as follows:

```
const tChar *strName = pFileReader->GetStreamName(nCounter);
```

You can access the *MediaDescription* for a *Stream*:

```
const tChar * GetStreamMediaDescription(...);
```

For further information about the *MediaDescription* see the DDL documentation inside the *SDK documentation*.

4. Execute the operation on a single or all *Streams*:

   - Reading

     Data blocks can be read with the method `Read()`. The following code fragment reads all data blocks of a *DAT File*:

     ```
     cADTFDataBlock *pDataBlock = NULL;
     while (IS_OK(pFileReader->Read(&pDataBlock)))
     {
         ...
     }
     ```

     Note that there are **two** different methods available for reading:

     a) `tResult Read(cADTFDataBlock **ppDataBlock);`

        This method returns the current data block. The returned pointer is **only** valid until the next call to `Read()`!

     b) `tResult Read(cADTFDataBlock *pDataBlock);`

        This method expects a user allocated data block of type `cADTFDataBlock` and fills it with the read data.

   - Setting the position in the *Stream*

5. Close the *DAT File*:

   To close an already opened *DAT File* use `IADTFFileReader::Close()`.

6. Release the instance of the `IADTFFileReader`.

   A created `IADTFFileReader` needs to be freed in the end as well. Use the following method to do so:

   ```
   IADTFFileReader::Release(pFileReader);
   ```

To filter the output of the logging you use the static method `IADTFFileReader::SetLogLevel`. An introduction to possible logging levels can be found in the *SDK Documentation*.

2.3 MediaDescriptor: tADTFMediaDescriptor

Information on the loaded *DAT File* is requested as follows:

```
const tADTFMediaDescriptor *pMDesc = pFileReader->GetMediaDescriptor();
```

A `tADTFMediaDescriptor` contains for instance a textual description as well as the creation date of the *DAT File*. A precise description of the elements of `tADTFMediaDescriptor` can be found in the *SDK Documentation*.

2.4 StreamDescriptor: tADTFStreamingDescriptor

The `tADTFStreamingDescriptor` serves to deliver exact information on the concerned *Stream*. It already holds according members for the *MediaTypes* which are supported by ADTF, e.g. video, audio, and CAN. It contains the *MediaType* and *MediaSubtype*, which are used to identify a *Stream* uniquely.

The stream descriptor can be obtained with the following call:

```
const tADTFStreamDescriptor *pSDesc = pFileReader->GetStreamDescriptor(nID);
```

The exact description of the elements of the `tADTFStreamDescriptor` can be found in the *SDK Documentation*.

2.5 Data block: cADTFDataBlock

A data block contains the serialized *MediaSample*, timestamps, and block indices. It is important to know to which *Stream* a read data block belongs to. Call

```
cADTFDataBlock:GetStreamID()
```

to get the *Stream ID*.

A data block has two different timestamps. On the one hand the timestamp of the *MediaSample* is available. What this timestamp exactly tells depends on the *Filter* which created the *MediaSample*, see

```
cADTFDataBlock::GetTime()
```

On the other hand there is the timestamp which tells the time the *MediaSample* was written to file by the *Harddisk Recorder*. This timestamp can be obtained using the function

```
cADTFDataBlock::GetFileTime()
```

To get the position of the data block in the *DAT File* you can request the block position of the data block, see

```
cADTFDataBlock::GetBlockPos()
```

Access to the content of the *MediaSample*, i.e. the saved data therein, is obtained by using the function

```
cADTFDataBlock::GetData()
```

2.6 Chunk Copy: cChunkCopy

The class `cChunkCopy` enables you to copy chunks from a source file to a destination file without interpreting the contained data.

First it is neccecary to create a copier instance with

```
cChunkCopy::Create()
```

After that, the source file has to be opened with

```
cChunkCopy::Open(const tChar *strFileName)
```

and the destination file has to be created with

```
cChunkCopy::CreateFile(const tChar *strFileName)
```

Now all streams can be copied to the new destination file, optionally with a new stream id and/or a different stream name. Call

```
cChunkCopy:AddStreamToCopy(const tUInt16 ui16StreamId,
    const tUInt16 ui16StreamIdNewId, const tChar* strStreamName)
```

to mark a stream for copying to the destination file.

After that, you can copy all regarding data blocks of the added streams with

```
cChunkCopy::CopyNext()
```

You can also jump to a special position and for copying only the data blocks from there with

```
cChunkCopy::Seek(tInt64 n64DataBlockPos)
```

After all, the source and destination files have to be closed with

```
cChunkCopy::Close()
```

For more informations look at the example *CopyChunks*.

2.7 FileWriter: IADTFFileWriter

The class `IADTFFileWriter` enables you to create *DAT Files* which are compatible to ADTF 2. It provides methods to create *Streams* and write data to these *Streams*.

Typically you will execute the following steps when working with the `IADTFFileWriter`:

1. Load all additionally needed factories and *Services*:

   ```
   AddClass(const tChar *strID, tVoid *pFactory);
   AddTypeService(const tChar *strServiceFilename);
   ```

   Do this before your first call to `IADTFFileWriter::Create()`!

2. Create an instance of `IADTFFileWriter`:

   ```
   IADTFFileWriter *pWriter = IADTFFileWriter::Create();
   ```

3. Open the file you want to write to *DAT File*:

   ```
   pWriter->Open(const tChar *strFileName);
   ```

4. Create all the *Streams* you need:

   ```
   pWriter->CreateStream(...);
   ```

   The *MediaTypes* contained in the basic install of ADTF can be found in the delivered header file `adtf_mediatypes.h`.

5. OPTIONAL: Set a *MediaDescription* for the *Stream*:

   ```
   pWriter->SetStreamMediaDescription(...);
   ```

   For further information about *MediaDescriptions* see the DDL documentation.

6. Write the file's properties:

```
pWriter->SetMediaDescription(...);
```

7. Write your extensions to the file:

```
pWriter->AddExtension(...);
```

8. Close the *DAT File*:

```
pWriter->Close();
```

9. Release your instance of `IADTFFileWriter`:

```
IADTFFileWriter::Release(pWriter);
```

To filter the output of the logging you use the static method `IADTFFileWriter::SetLogLevel`. An introduction to possible logging levels can be found in the *SDK Documentation*.

2.7.1 Time synchronicity

Time synchronicity is defined as follows in ADTF: every recorded data element is contained in a *MediaSample* which contains a timestamp of its creation date. However, the *Harddisk Player* itself does not provide time synchronized playback. That means the time difference between two samples played back does not have to match exactly. Furthermore, samples are played back in the order they are written to the *DAT File*.

What this means for you

The `IADTFFileWriter` does not sort incoming data by its timestamp, since that would require a very big data buffer. If you fill different *Streams*, do not do this one after the other!

Wrong

1. fill *Stream* A
2. fill *Stream* B
3. fill *Stream* C
4. ...

For clarification

If you record a video, do not firstly save the video data and then save the audio data. Instead, maintain the approximate order of the incoming data.

Right

1. receive data for *Stream* A ⇒ write data to *Stream* A
2. receive data for *Stream* B ⇒ write data to *Stream* B
3. receive data for *Stream* A ⇒ write data to *Stream* A
4. ...

Created timestamps

Please note that ADTF in principle uses two different timestamps:

- ***MediaSample* timestamp**: This timestamp marks the creation of the *MediaSample*.
- **Chunk timestamp**: This timestamp marks the reception of the *MediaSample* by the *Harddisk Recorder*. Timestamps of consecutive recorded *MediaSamples* may only rise or remain constant.

3 FAQ

- Question: How do I select a *Stream*?
  - **Select by name**: Each *Stream* possesses a unique name. To obtain this name call the function `IADTFFileReader::GetStreamName(tUInt16 nStreamId)` for the *Stream IDs* from 1 to the value returned by `GetStreamCount()` and compare the result with the *Stream* name you are looking for. Not that the *Stream ID* 0 is generally used to access **all** *Streams*.
  - **Since version 2.1.1**: Since version 2.1.1 the API of `IADTFFileReader` provides the method `GetStreamId()`. This method allows direct access to a *Stream's ID* if its name is known. The API of `IADTFFileWriter`, available since version 2.1.1, supports this method too.
  - **Select by stream descriptor**: A `cADTFStreamDescriptor` returns the information of the *Stream*. You can obtain the stream descriptor according to a received *Stream* by calling `IADTFFileReader::GetStreamDescriptor(StreamId)`. The stream descriptor contains the type of the *Stream* provided it is of a common format. Additionally the stream descriptor conatins the *MediaType* and the *MediaSubType* for precise identification of the content.
- Question: `Read()` returns the data blocks of all *Streams* Can this be filtered?

  There is no automatic filtering function. Select the *Stream IDs* of the *Streams* you care for before calling `Read()`.
- Question: I want to read data with the *Streaming Library* from a *DAT File* which is not part of the *MediaSamples* delivered with ADTF. How do I proceed?

  Register the according class factories or *Services* using `IADTFFileReader::AddClass()` or `IADTFFileReader::AddTypeService()`.
- Question: `CreateStream` expects a sample type and a *MediaType*. Is the *MediaType* the same as major and subtype in ADTF?

  No. This is an unfortunate case of naming conflict. In this case it is referred to the class

`cMediaType`, from which developers can derive their own implementation.

Defintions of media and sample type provided by the basic installation of ADTF can be found in the file `adtf_mediatypes.h`.

- Question: Well, then what is the sample type?

  Developers may derive their own *MediaSample* implementations from the default *ADTF MediaSample*. For example, ADTF contains a special *MediaSample* for CAN messages derived from `cMediaSample`. In order to write data correctly, the `IADTFFileWriter` needs to know what kind of *MediaSample* to use. ADTF basic sample definitions are provided in the file `adtf_mediatypes.h`. From a developer's viewpoint, these are simly the OIDs of the derived classes.

- Question: Can I create multiple *Streams* sharing the same name in a *DAT File*?

  No. The name of a *Stream* is unique inside a *DAT File*.

- Question: Can I assign a specific *Stream ID* to a *Stream*?

  No. Use the *Stream* name as a unique identifier for your *Stream* instead. Do not rely at the first created *Stream* having the *Stream ID* 1. Do not rely on consecutively created *Streams* having sequential IDs.

- Question: I created a *DAT File* with multiple *Streams*. When I play this file, the *Streams* seem to be played back one after the other instead of parallel. Why?

  You wrote your *Streams*—when actually ran in parallel—consecutively into your *DAT File*. The class `IADTFFileReader` does not sort your written data. Read the section about time synchronicity in order to understand why this does not work and how to do it correctly (see subsection 2.7.1 Time synchronicity).

- Question: Is it possible to repair a *DAT File* I accidentally wrote my *Streams* consecutively into?

  Currently there is no tool to convert such a file. However, the *Streaming Library* provides you with everything you need in order to write such a tool yourself.

- Question: Are *Stream Handles* and *Stream IDs* identical?

  Yes, these are historic names for the same thing.

- Question: How can I access compressed video streams?

  The *Streaming Library* itself will not perform any kind of decompression. It will only hand you raw compressed data that you can decompress on your own. The example application *compressed-video* shows how to do this for all currently available video codecs of the *ADTF Video Compression Toolbox*.

- Question: Why are all structs/classes/defines named differently than in ADTF?

This is intentional! The *ADTF Streaming Library* is designed to be used independently from ADTF and as such can not reuse the types defined in ADTF. Redefining them in the *Streaming Library* would create namespace collisions when both ADTF and the *Streaming Library* are used. Furthermore the structures are different and incompatible with those of ADTF.

- Problem: Crashes! Errors!

  Always check the returned pointers if they are `NULL`. We try our best to prevent errors. Nevertheless if you find an error which you can reproduce, please send an email to our support with the exact conditions under which the error occurs.

- Question: Is there an index entry in the index table for every chunk in a *Stream*?

  No there is only an index entry every second for a *Stream*. Otherwise the index table would be too big. Additionally when creating a *DAT File* the index table is stored in the memory.

Index