USER MANUAL

Tagged Stream Format - TSF V1.1.19

1 Tagged Stream Format - TSF V1.1.19

The *Tagged Stream Format* enables programmers to read and write binary documents with nested, sorted and unambiguously tagged objects efficiently even in restrictive environments like microcontrollers. These documents can also be converted to XML files, or JSON files, or be documented expressively, e. g.:

#1	#2	#3	#4	Data Type / Object Name	Description / Comment	Group
1				StreamInfo	Unspecified tags of the root tag collection are reserved	
	2			UINT RegisteredId	(Reserved)	id
	3			BYTE[] UnregisteredUuid	*Uuid* of the document type (consisting of 16 bytes)	id
	4			UINT VersionMajor	Downward **in**compatible major version. The meaning of the tags has been changed, or tags have been removed with an increasing *VersionMajor* value.	
	5			UINT VersionMinor	Optional: Nonzero downward compatible minor version. Tags have been added with an increasing *VersionMinor* value. This field merely indicates the target device's understanding of the stream format.	
	11			Streams		
		3		Stream		
			9	BYTE[] Uuid	Optional: Stream *Uuid* (consisting of 16 bytes if defined)	
3				ExampleCollections[]		
	3			BYTE[] ExampleString	Array of BYTE	
	5			UINT[] ExampleArray	Array of UINT	
5				UINT16 ExampleUInt		

If a group is defined then exactly one tag of a group may be defined.

The binary representation of an example TSF document of the predefined type looks like:

```
0000: 01 E1 74 73 0D 1C 10 B1  B6 FE AE 31 C7 48 49 BA
0010: 42 C7 39 88 B9 75 73 5D  1D 4C 10 07 2C 7D A8 13
0020: 35 43 EA 88 33 4C EB 46  B0 61 4F 00 00 00 1E 02
0030: 1A 01 02 03 04 2F 11 02  03 03 00 02 00 01 00 00
0040: 1C 05 01 02 03 04 05 2F  12 03 02 D2 04 01 03 04
0050: 4E 61 BC 00 00 29 D2 04  00
```

The corresponding raw XML representation looks like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tsf>
  <obj id="1">
    <obj id="3">b1 b6 fe ae 31 c7 48 49  ba 42 c7 39 88 b9 75 73
      <!-- "....1.HI.B.9..us" --></obj>
    <obj id="11">
      <obj id="3">
        <obj id="9">07 2c 7d a8 13 35 43 ea  88 33 4c eb 46 b0 61 4f
          <!-- ".,}..5C..3L.F.aO" --></obj>
      </obj>
    </obj>
  </obj>
  <obj id="3">
    <obj>
      <obj id="3">01 02 03 04
        <!-- "...." / 67305985 / 1.5399896e-36f --></obj>
      <obj id="5">
        <data>03 00  <!-- ".." / 3 --></data>
        <data>02 00  <!-- ".." / 2 --></data>
        <data>01 00  <!-- ".." / 1 --></data>
      </obj>
    </obj>
    <obj>
      <obj id="3">01 02 03 04 05  <!-- "....." --></obj>
      <obj id="5">
        <data>d2 04  <!-- ".." / 1234 --></data>
        <data>03  <!-- "." / 3 --></data>
        <data>4e 61 bc 00
          <!-- "Na.." / 12345678 / 1.729998e-38f --></data>
      </obj>
    </obj>
  </obj>
  <obj id="5">d2 04  <!-- ".." / 1234 --></obj>
</tsf>
```

The corresponding JSON representation looks like:

```json
{
    "root": {
        "uuid": "b1b6feae-31c7-4849-ba42-c73988b97573",
        "streams": {
            "stream": {
                "uuid": "072c7da8-1335-43ea-8833-4ceb46b0614f"
            }
        }
    },
    "exampleCollections": [ {
        "exampleString": "\u0001\u0002\u0003\u0004",
        "exampleArray": [ 3, 2, 1 ]
      }, {
        "exampleString": "\u0001\u0002\u0003\u0004\u0005",
        "exampleArray": [ 1234, 3, 12345678 ]
    } ],
    "exampleUInt": 1234
```

```
}
```

The C# program to generate and read these bytes is as follows:

```csharp
using System;
using System.Diagnostics;
using Tellert.Format;

namespace test_cs
{
    class Program
    {
        const string tsfUuid =
          "b1b6feae-31c7-4849-ba42-c73988b97573";
        const uint tsfMajorVersion = 0;
        const uint tsfMinorVersion = 0;

        struct ExampleCollection // ExampleCollection
        {
            public byte[] String;
            public uint[] Array;
        }

        struct Document
        {
            public Uuid Uuid;
            public ExampleCollection[] ExampleCollections;
            public ushort ExampleUInt16;
        }

        static bool WriteTsfFile(ref Document doc, string fileName)
        {
            Uuid uuid = new Uuid(tsfUuid);

            Tsf tsf = new Tsf(1024);
            if (tsf.WriteArray(1, 1))
            {
                tsf.Write(3, uuid);
                tsf.Write(4, tsfMajorVersion);
                tsf.Write(5, tsfMinorVersion);
                if (doc.Uuid.Good() && tsf.WriteArray(11, 1))
                {
                    if (tsf.WriteArray(3, 1))
                    {
                        tsf.Write(9, doc.Uuid);
                        tsf.WriteEnd();
                    }
                    tsf.WriteEnd();
                }
                tsf.WriteEnd();
            }
            if (tsf.WriteArray(3, doc.ExampleCollections.Length))
            {
                for (int i = 0; i < doc.ExampleCollections.Length;
                  i++)
                {
                    ExampleCollection c = doc.ExampleCollections[i];
                    tsf.Write(3, c.String);
                    if (i == 0)
                    {
                        if (tsf.WriteFixedArray(5, c.Array.Length,
                          sizeof(ushort)))
                        {
                            for (int j = 0; j < c.Array.Length; j++)
                            {
```

```csharp
                        tsf.WriteFixedArray(c.Array[j]);
                    }
                }
            }
            else
            {
                if (tsf.WriteVarArray(5, c.Array.Length))
                {
                    for (int j = 0; j < c.Array.Length; j++)
                    {
                        tsf.WriteVarArray(c.Array[j]);
                    }
                }
            }
            tsf.WriteEnd();
        }
    }
    tsf.Write(5, doc.ExampleUInt16);
    tsf.WriteEnd();

    return tsf.TryWriteFile(fileName);
}

static bool ReadTsfFile(ref Document doc, string fileName)
{
    Tsf tsf = new Tsf();
    Uuid uuid = Uuid.Empty, refUuid = new Uuid(tsfUuid);
    uint majorVersion = 0;
    uint minorVersion = 0;
    bool result = true;

    doc.Uuid = Uuid.Empty;
    doc.ExampleUInt16 = 0;

    if (!tsf.TryOpen(fileName)) return false;
    if (tsf.SelectArray(1))
    {
        if (tsf.Select(3)) tsf.Read(ref uuid);
        if (tsf.Select(4)) tsf.Read(ref majorVersion);
        if (tsf.Select(5)) tsf.Read(ref minorVersion);
        if (tsf.SelectArray(11))
        {
            if (tsf.SelectArray(3))
            {
                if (tsf.Select(9)) tsf.Read(ref doc.Uuid);
                tsf.SkipArray();
            }
            tsf.SkipArray();
        }
        tsf.SkipArray();
    }
    doc.ExampleCollections = new ExampleCollection[tsf.
      ReadArraySize(3)];
    for (int i = 0; i < doc.ExampleCollections.Length; i++) {
        ExampleCollection c = new ExampleCollection();
        if (tsf.Select(3)) tsf.Read(ref c.String);
        if (tsf.Select(5))
        {
            c.Array = new uint[tsf.DataCount];
            for (int j = 0; j < c.Array.Length; j++)
            {
                tsf.Read(ref c.Array[j]);
                tsf.SelectNextItem();
            }
        }
        doc.ExampleCollections[i] = c;
```

```csharp
        tsf.SkipCollection();
    }
    if (tsf.Select(5)) tsf.Read(ref doc.ExampleUInt16);
    if (tsf.Bad) result = false;


    if (uuid != refUuid || majorVersion > tsfMajorVersion)
      result = false;


    return result;
}


static void Main(string[] args)
{
    Document doc = new Document(), doc2 = new Document();

    doc.Uuid = Uuid.NewUuid();
    doc.ExampleCollections = new ExampleCollection[2];
    doc.ExampleCollections[0].String =
      new byte[] { 1, 2, 3, 4 };
    doc.ExampleCollections[0].Array = new uint[3];
    for (uint i = 0; i < 3; i++)
      doc.ExampleCollections[0].Array[i] = 3 - i;
    doc.ExampleCollections[1].String =
      new byte[] { 1, 2, 3, 4, 5 };
    doc.ExampleCollections[1].Array = new uint[3];
    doc.ExampleCollections[1].Array[0] = 1234;
    doc.ExampleCollections[1].Array[1] = 3;
    doc.ExampleCollections[1].Array[2] = 12345678;
    doc.ExampleUInt16 = 1234;


    Debug.Assert(WriteTsfFile(ref doc, "test.tsf"));
    Debug.Assert(ReadTsfFile(ref doc2, "test.tsf"));


    // to be compared outside with "test.tsf"
    Debug.Assert(WriteTsfFile(ref doc2, "test2.tsf"));
    }
  }
}
```

The C++ program to generate and read these bytes is as follows:

```cpp
#include "tsfclass.h"


#define TSF_UUID "b1b6feae-31c7-4849-ba42-c73988b97573"
#define TSF_MAJOR_VERSION 0
#define TSF_MINOR_VERSION 0


struct ExampleCollection {
    TsfBytes ExampleString;
    std::vector<unsigned> ExampleArray;
};
inline bool operator == (const ExampleCollection &lhs,
  const ExampleCollection &rhs)
{
    return lhs.ExampleString == rhs.ExampleString &&
      lhs.ExampleArray == rhs.ExampleArray;
}


struct Document {
    Tsf::Uuid DocumentUuid;
    std::vector<ExampleCollection> ExampleCollections;
    TsfUInt16 ExampleUInt16;
};


bool WriteTsfFile(const Document &doc, const TSF_TCHAR * fileName)
{
    Tsf::Uuid uuid(TSF_UUID);
    size_t i, j;


    Tsf tsf(1024);
    if (tsf.WriteArray(1, 1)) {
        tsf.Write(3, uuid);
        if (doc.DocumentUuid.Good() && tsf.WriteArray(11, 1)) {
            if (tsf.WriteArray(3, 1)) {
                tsf.Write(9, doc.DocumentUuid);
                tsf.WriteEnd();
            }
            tsf.WriteEnd();
        }
        tsf.WriteEnd();
    }
    if (tsf.WriteArray(3, doc.ExampleCollections.size())) {
        for (i = 0; i < doc.ExampleCollections.size(); i++) {
            const ExampleCollection &c = doc.ExampleCollections[i];
            tsf.Write(3, c.ExampleString);
            if (i == 0) {
                if (tsf.WriteFixedArray(5, c.ExampleArray.size(),
                  sizeof(TsfUInt16))) {
                    for (j = 0; j < c.ExampleArray.size(); j++) {
                        tsf.WriteFixedArray(c.ExampleArray[j]);
                    }
                }
            }
            else {
                if (tsf.WriteVarArray(5, c.ExampleArray.size())) {
                    for (j = 0; j < c.ExampleArray.size(); j++) {
                        tsf.WriteVarArray(c.ExampleArray[j]);
                    }
                }
            }
            tsf.WriteEnd();
        }
```

```
    }
    tsf.Write(5, doc.ExampleUInt16);
    tsf.WriteEnd();


    return tsf.WriteFile(fileName);
}


bool ReadTsfFile(Document &doc, const TSF_TCHAR *fileName)
{
    Tsf tsf;
    Tsf::Uuid uuid, refUuid(TSF_UUID);
    unsigned majorVersion = 0;
    unsigned minorVersion = 0;
    size_t i, j;
    bool result = true;

    doc.DocumentUuid.Empty();
    doc.ExampleUInt16 = 0;

    if (!tsf.Open(fileName)) return false;
    if (tsf.SelectArray(1)) {
        if (tsf.Select(3)) tsf.Read(uuid);
        if (tsf.Select(4)) tsf.Read(majorVersion);
        if (tsf.Select(5)) tsf.Read(minorVersion);
        if (tsf.SelectArray(11)) {
            if (tsf.SelectArray(3)) {
                if (tsf.Select(9)) tsf.Read(doc.DocumentUuid);
                tsf.SkipArray();
            }
            tsf.SkipArray();
        }
        tsf.SkipArray();
    }
    doc.ExampleCollections.resize(tsf.ReadArraySize(3));
    for (i = 0; i < doc.ExampleCollections.size(); i++) {
        ExampleCollection &c = doc.ExampleCollections[i];
        if (tsf.Select(3)) tsf.Read(c.ExampleString);
        if (tsf.Select(5)) {
            c.ExampleArray.resize(tsf.GetDataCount());
            for (j = 0; j < c.ExampleArray.size(); j++) {
                tsf.Read(c.ExampleArray[j]);
                tsf.SelectNextItem();
            }
        }
        tsf.SkipCollection();
    }
    if (tsf.Select(5)) tsf.Read(doc.ExampleUInt16);
    if (tsf.Bad()) result = false;

    if (uuid != refUuid || majorVersion > TSF_MAJOR_VERSION)
      result = false;


    return result;
}


int main(int argc, char *argv[])
{
    static Document doc, doc2;
    int i;

    doc.DocumentUuid.New();
    doc.ExampleCollections.resize(2);
    ExampleCollection &c0 = doc.ExampleCollections[0];
    c0.ExampleString = { 1, 2, 3, 4 };
```

```
c0.ExampleArray.resize(3);
for (i = 0; i < 3; i++) c0.ExampleArray[i] = 3 - i;
ExampleCollection &c1 = doc.ExampleCollections[1];
c1.ExampleString = { 1, 2, 3, 4, 5 };
c1.ExampleArray.resize(3);
c1.ExampleArray[0] = 1234;
c1.ExampleArray[1] = 3;
c1.ExampleArray[2] = 12345678;
doc.ExampleUInt16 = 1234;

if (!WriteTsfFile(doc, TSF_TEXT("test.tsf"))) return 1;
if (!ReadTsfFile(doc2, TSF_TEXT("test.tsf"))) return 2;
if (doc.DocumentUuid != doc2.DocumentUuid ||
  doc.ExampleCollections != doc2.ExampleCollections ||
  doc.ExampleUInt16 != doc2.ExampleUInt16) return 3;

return 0;
}
```

The C program to generate and read these bytes is as follows:

```c
#include "tsf.h"

#define TSF_UUID "b1b6feae-31c7-4849-ba42-c73988b97573"
#define TSF_MAJOR_VERSION 0
#define TSF_MINOR_VERSION 0
static char tsfData[1024];
static TsfObject tsfObject;


#define MAX_STRING 255
#define MAX_TOTAL_STRING (MAX_STRING+1) /* +1 for terminating NUL */
#define MAX_ARRAY 32
#define MAX_ITEMS 16


typedef struct {
    size_t ExampleStringSize;
    TsfByte ExampleString[MAX_TOTAL_STRING];
    size_t ExampleArraySize;
    unsigned ExampleArray[MAX_ARRAY];
} ExampleCollection;


typedef struct {
    TsfUuid DocumentUuid;
    size_t ExampleCollectionSize;
    ExampleCollection ExampleCollections[MAX_ITEMS];
    TsfUInt16 ExampleUInt16;
} Document;


TsfBool WriteTsfFile(const Document *doc, const TSF_TCHAR * fileName)
{
    TsfObject *tsf = &tsfObject;
    TsfUuid uuid;
    size_t i, j;
    TsfBool result = TSF_TRUE;

    TsfUuidFromString(uuid, TSF_UUID);

    TsfCreate(tsf, tsfData, sizeof(tsfData), TSFH_NORMAL, TSF_TRUE);
    if (TsfWriteArray(tsf, 1, 1)) {
        TsfWriteUuid(tsf, 3, uuid);
        TsfWriteUInt(tsf, 4, TSF_MAJOR_VERSION);
        TsfWriteUInt(tsf, 5, TSF_MINOR_VERSION);
        if (TsfUuidIsGood(doc->DocumentUuid) &&
          TsfWriteArray(tsf, 11, 1)) {
            if (TsfWriteArray(tsf, 3, 1)) {
                TsfWriteUuid(tsf, 9, doc->DocumentUuid);
                TsfWriteEnd(tsf);
            }
            TsfWriteEnd(tsf);
        }
        TsfWriteEnd(tsf);
    }
    if (TsfWriteArray(tsf, 3, doc->ExampleCollectionSize)) {
        for (i = 0; i < doc->ExampleCollectionSize; i++) {
            const ExampleCollection *c = &doc->ExampleCollections[i];
            TsfWriteObject(tsf, 3, c->ExampleString,
              c->ExampleStringSize);
            if (i == 0) {
                if (TsfWriteFixedArray(tsf, 5, c->ExampleArraySize,
                  sizeof(TsfUInt16))) {
                    for (j = 0; j < c->ExampleArraySize; j++) {
```

```c
                        TsfWriteFixedArrayUInt(tsf,
                            c->ExampleArray[j]);
                    }
                }
            }
            else {
                if (TsfWriteVarArray(tsf, 5, c->ExampleArraySize)) {
                    for (j = 0; j < c->ExampleArraySize; j++) {
                        TsfWriteVarArrayUInt(tsf,
                            c->ExampleArray[j]);
                    }
                }
            }
            TsfWriteEnd(tsf);
        }
    }
    TsfWriteUInt(tsf, 5, doc->ExampleUInt16);
    TsfWriteEnd(tsf);
    if (!TsfWriteFile(tsf, fileName)) result = TSF_FALSE;
    TsfClose(tsf);


    return result;
}


TsfBool ReadTsfFile(Document *doc, const TSF_TCHAR *fileName)
{
    TsfObject *tsf = &tsfObject;
    TsfUuid uuid, refUuid;
    unsigned majorVersion = 0;
    unsigned minorVersion = 0;
    size_t i, j;
    bool result = TSF_TRUE;


    TsfUuidEmpty(uuid);
    TsfUuidFromString(refUuid, TSF_UUID);
    memset(doc, '\0', sizeof(Document));


    if (!TsfOpenFile(tsf, fileName, tsfData, sizeof(tsfData)))
      return TSF_FALSE;
    if (TsfSelectArray(tsf, 1)) {
        if (TsfSelect(tsf, 3)) TsfReadUuid(tsf, &uuid);
        if (TsfSelect(tsf, 4)) TsfReadUInt(tsf, &majorVersion);
        if (TsfSelect(tsf, 5)) TsfReadUInt(tsf, &minorVersion);
        if (TsfSelectArray(tsf, 11)) {
            if (TsfSelectArray(tsf, 3)) {
                if (TsfSelect(tsf, 9)) TsfReadUuid(tsf,
                    &doc->DocumentUuid);
                TsfSkipArray(tsf);
            }
            TsfSkipArray(tsf);
        }
        TsfSkipArray(tsf);
    }
    doc->ExampleCollectionSize = TsfReadArraySize(tsf, 3);
    for (i = 0; i < doc->ExampleCollectionSize; i++) {
        ExampleCollection *c = &doc->ExampleCollections[i];
        if (i < MAX_ITEMS) {
            if (TsfSelect(tsf, 3)) {
                c->ExampleStringSize = TsfGetDataSize(tsf);
                TsfReadObject(tsf, c->ExampleString, MAX_STRING);
            }
            if (TsfSelect(tsf, 5)) {
                c->ExampleArraySize = TsfGetDataCount(tsf);
                for (j = 0; j < c->ExampleArraySize; j++) {
                    if (j < MAX_ARRAY) {
                        TsfReadUInt(tsf, &c->ExampleArray[j]);
```

```c
                    }
                    else {
                        TsfSkipData(tsf);
                    }
                    TsfSelectNextItem(tsf);
                }
            }
        }
        TsfSkipCollection(tsf);
    }
    if (TsfSelect(tsf, 5)) TsfReadUInt16(tsf, &doc->ExampleUInt16);
    if (TsfIsBad(tsf)) result = TSF_FALSE;
    TsfClose(tsf);

    if (!TsfUuidIsEqual(uuid, refUuid) || majorVersion >
      TSF_MAJOR_VERSION) result = TSF_FALSE;

    return result;
}


int main(int argc, char* argv[])
{
    static Document doc, doc2;
    ExampleCollection *c;
    int i;

    memset(&doc, '\0', sizeof(doc));
    TsfUuidNew(doc.DocumentUuid);
    doc.ExampleCollectionSize = 2;
    c = &doc.ExampleCollections[0];
    c->ExampleStringSize = 4;
    memcpy(c->ExampleString, "\1\2\3\4", 4);
    c->ExampleArraySize = 3;
    for (i = 0; i < 3; i++) c->ExampleArray[i] = 3 - i;
    c = &doc.ExampleCollections[1];
    c->ExampleStringSize = 5;
    memcpy(c->ExampleString, "\1\2\3\4\5", 5);
    c->ExampleArraySize = 3;
    c->ExampleArray[0] = 1234;
    c->ExampleArray[1] = 3;
    c->ExampleArray[2] = 12345678;
    doc.ExampleUInt16 = 1234;

    if (!WriteTsfFile(&doc, TSF_TEXT("test.tsf"))) return 1;
    if (!ReadTsfFile(&doc2, TSF_TEXT("test.tsf"))) return 2;
    if (memcmp(&doc, &doc2, sizeof(Document)) != 0) return 3;

    return 0;
}
```

1.1   Light Version

A tagged stream format is a binary stream representation with a *NUL* terminated collection of nested, sorted and unambiguously tagged data items. The data of a missing item is set to zero (or an empty string).

A *flexible number* is defined with a single byte. If this byte equals $0ff_{16}$, then the next two bytes represent the flexible number. If these next two bytes equal $0ffff_{16}$, then the next four bytes represent the flexible number, and so on. Flexible numbers are prefixed with *num*.

An *id* represents a strictly monotonically increasing non-zero identification number within a *NUL* terminated object collection sequence. The *id* with number 31 is reserved.

Identified objects consist of an *id* byte and object data. An *id* byte consists of the *id* field (higher significant 5 bits) and of the *type* field (lower significant 3 bits):

id = 0	type = 0	Zero tag
id = 0	type = 1	Header (little endian): 01 e1 74 73 Header (big endian): 01 e1 73 74
id = 0	type > 1	Reserved
id > 0	type = 0	Data size: 1 byte
id > 0	type = 1	Data size: 2 bytes
id > 0	type = 2	Data size: 4 bytes
id > 0	type = 3	Data size: 8 bytes
id > 0	type = 4	Data size is defined by *numDataSize*; subsequent items: *<numDataSize>*
id > 0	type = 5	Object collection vector of length 1 until zero tag; subsequent items: *<objs[0]>*
id > 0	type = 6	Object collection vector; subsequent items: *<numCount><objs[0]><objs[1]>...*
id > 0	type = 7	Reserved

1.2   Full Version

A tagged stream format is a binary stream representation with a *NUL* terminated collection of nested, sorted and unambiguously tagged data items. The data of a missing item is set to zero (or an empty string).

A *flexible number* is defined with a single byte. If this byte equals $0ff_{16}$, then the next two bytes represent the flexible number. If these next two bytes equal $0ffff_{16}$, then the next four bytes represent the flexible number, and so on. Flexible numbers are prefixed with *num*.

An *id* represents a strictly monotonically increasing non-zero identification number within a *NUL* terminated object collection sequence. If all *id* bits are set, then these bits are ignored and the *id* is the directly following flexible number after the *id* byte.

id 1 should be reserved for the root level. Extended array data types should be omitted if possible. Numbers should be kept as little as possible, and the identifiers should be limited to 30 if possible.

Identified objects consist of an *id* byte and the object data. An *id* byte consists of the *id* field (higher significant 5 bits) and of the *type* field (lower significant 3 bits):

id = 0	type = 0	Zero tag
id = 0	type = 1	Header: 01 { id:28, type:1 } "ts"|"st" Header (little endian): 01 e1 74 73 Header (big endian): 01 e1 73 74
id = 0	type = 2	NOP: Byte without further meaning
id = 0	type > 2	Reserved
id > 0	type = 0	Data size: 1 byte
id > 0	type = 1	Data size: 2 bytes
id > 0	type = 2	Data size: 4 bytes
id > 0	type = 3	Data size: 8 bytes
id > 0	type = 4	Data size is defined by *numDataSize*; subsequent items: *<numDataSize>* Reserved: *numDataSize* = 0|1|2|4|8
id > 0	type = 5	Object collection vector of length 1 until zero tag; subsequent items: *<objs[0]>*
id > 0	type = 6	Object collection vector; subsequent items: *<numCount><objs[0]><objs[1]>*... Reserved: *numCount* < 2

id > 0	type = 7	Extended data type; Subsequent byte: *<dim:4 edType:4>*
		
		  edType = 0: no subsequent items / *dim* = 0: *default* (0 or *empty string*)
		  dim = 1: *null* / *dim* = 2: *undefined* (redefined by a later id tag)
		
		  edType = 1: Fixed-sized jagged data array; subsequent items:
		   <numFixedItemSize><numCount1>[...<numCount#dim>]
		   <item0>[<item1>...]
		
		  edType = 2: Variable-sized jagged data array; subsequent items:
		   <numCount1>[...<numCount#dim>]
		   <numSizeofItem0><item0>[<numSizeofItem1><item1>...]
		
		  Reserved - *edType* = 3: Jagged object array; subsequent items:
		   <numCount1>[...<numCount#dim>]<objs0>[<objs1>]

1.3　License

Tagged Stream Format V1.1.19
Copyright (c) 2017-2019 Rudy Tellert Elektronik

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.4　Internet

The *Tagged Stream Format* homepage is located at
https://www.tellert.de/?product=tsf