DTAPI

Audio/Video FIFO Interface (AvFifo)

REFERENCE

Oct 2023

Table of Contents

1. Introduction

1.1. What is the AvFifo Interface?

AvFifo, short for Audio/Video FIFO Interface, is a component of the DekTec Application Programming Interface (DTAPI). While DTAPI offers multiple programming models, including the "input/output channel" model and the “Matrix API” model, AvFifo introduces a third, FIFO-based model.

Designed specifically for software-based digital-TV processing applications, like video encoders and decoders, AvFifo facilitates efficient interfacing with SMPTE 2110 and SMPTE 2022-5/6 streams through the DTA-2110 10GbE NIC (Network Interface Card).

For a comprehensive overview of DTAPI and installation instructions, refer to the "DTAPI Manual – Overview and Data Formats", included with the DTAPI installation.

This manual focuses on the details of the AvFifo API, providing examples to help you understand and implement the AvFifo interfacing model within your applications.

1.2. The FIFO-Based Interfacing Model

In the AvFifo interfacing model, the unit of data is a "frame," which represents a timestamped unit from a video, audio, or ancillary data stream. The model simplifies the process of data reception and transmission by efficiently handling the decoding, encoding, and scheduling of frames.

Reception: The AvFifo implementation decodes frames from incoming IP streams and stores them, along with their timestamps, in a FIFO. The application then reads frames from the FIFO as needed.

Transmission: The application assembles frame data, computes a timestamp indicating when the frame should be transmitted, and writes the frame to a FIFO. The AvFifo implementation reads frames from the FIFO and schedules them for transmission at their respective timestamps.

1.3. Timing Model

1.3.1. PTP Agent

The AvFifo interface relies on the Precision Time Protocol (PTP) for timing, ensuring accurate synchronization across various system components. All streams use the PTP network clock as their reference clock.

When you install the drivers for Windows or Linux, a PTP agent is automatically installed and runs within the DTAPI service. This agent synchronizes a clock counter in the hardware with PTP time.

Note: Although the PTP agent is installed by default, it must be enabled manually, as outlined in the subsequent sections.

All timestamps in the AvFifo interface are relative to PTP time, ensuring precise synchronization for audio, video, and ancillary data streams. The AvFifo implementation handles all time-sensitive operations, eliminating the need for users to perform any manual time-sensitive tasks. This simplifies the process and improves overall efficiency of the system.

The PTP agent has several settings available for configuration.

Setting	Range	Description
Enable	false, true	Turns the PTP agent on or off..
DelayMechanism	0 .. 2	0=auto, 1=EndToEnd, 2=PeerToPeer
NetworkProtocol	0, 1	0=IPv4, 1=IPv6
DomainNumber	0 .. 127	
IpV6Scope	1 .. 14	
PeerUnicastAddress:	IPv4/IPv6 address	

1.3.2. Configuring the PTP Agent in Windows

To enable synchronization with PTP in Windows, users must activate the PTP feature in the DtInfo application. A dialog enables configuring the PTP settings.

1.3.3. Configuring the PTP Agent in Linux

In Linux, the PTP settings are maintained in a configuration file, located in the following directory: /var/lib/DekTec/Service/PtpClockSlave/.

The filename follows the pattern: "CardSerialNumber_Port0_Inst0.xml", e.g. "2110000018_Port0_Inst0.xml".

Here's an example of the file layout::

```
<DekTec>
  <PtpClockSlave Cnt="6">
    <ParStrVal S="IpV6Scope" VT="2" VV="5"/>
    <ParStrVal S="DelayMechanism" VT="2" VV="0"/>
    <ParStrVal S="NetworkProtocol" VT="2" VV="0"/>
    <ParStrVal S="PeerUnicastAddress" VT="5" VV="0.0.0.0"/>
    <ParStrVal S="DomainNumber" VT="2" VV="127"/>
    <ParStrVal S="Enable" VT="4" VV="true"/>
  </PtpClockSlave>
</DekTec>
```

The 'VV' attribute contains the value of the parameter with the attribute name 'S'. Only the 'VV' attribute may be changed.

Changes made to the settings will come into effect following a restart of the DTAPI service.

1.4. AvFifo Code Examples

Section 2 of this manual introduces the basics of utilizing the AvFifo API to receive or transmit an SMPTE 2110 stream. The tutorial focuses primarily on fundamental concepts and intentionally omits details such as error handling for simplicity.

For a more comprehensive exploration that includes aspects like error handling, expanded source code examples can be found on the DekTec website at the following address: www.dektec.com/downloads/SDK/#sdkexamples. These AvFifo code examples can be run on both Linux and Windows.

At present, two example codes are available specifically for DekTec NICs that support the AvFifo API. As of the time of writing, the DTA-2110 is the only supported device, although additional compatible devices will be introduced shortly. The provided examples are as follows:

1. **AvFifo_VideoRx** – This code serves as a SMPTE 2110 video receiver that displays the stream using SDL.
2. **AvFifo_VideoTx** – This example is a SMPTE 2110 video test generator.

For updates and additions to these examples, we recommend regularly checking the DekTec website.

2. Tutorial

2.1. Introduction

In this tutorial, we will explore a straightforward reception and transmission scenario. We will demonstrate the process by implementing a basic receive loop using an **RxFifo** object and a basic transmission loop using a **TxFifo** object.

Note: Error handling is omitted for simplicity and clarity; ensure to include it in your production code.

2.2. Basic Receive Example

The following code example demonstrates how to set up DTA-2110 with an **RxFifo** object for receiving data in a simple reception scenario.

```
// 1. Declare the DTA-2110 device object and attach to the hardware.
DtDevice Dta2110{};
Dta2110.AttachToType(2110);

// Here is a good place to check NIC status with Dta2110.IsNetworkCardOperational(1).

// 2. Declare the RxFifo object, attach to port 1 of the DTA-2110, and configure for reception.
AvFifo::RxFifo RxFifo{};
RxFifo.Attach(Dta2110, 1, HwOrSwPipe::PreferHwPipe);
RxFifo.SetIpPars({…});
RxFifo.Configure({…});

// 3. Start the RxFifo.
RxFifo.Start();

// 4. Keep receiving until the stop condition is met.
while (!StopCondition())
{
    // Here is a good place to check the RxFifo status with VideoRxFifo.GetStatus().

    // 5. Read and process a frame from the RxFifo if one is available; otherwise, sleep briefly.
    if (RxFifo.GetFifoLoad() > 0)
    {
        AvFifo::Frame* Frm{RxFifo.Read()};
        ProcessReceivedFrame(Frm);
        RxFifo.ReturnToMempool(Frm);
    } else
        std::this_thread::sleep_for(10ms);
}

// 6. Clean up.
RxFifo.Stop();
RxFifo.Detach();
Dta2110.Detach();
```

Note This code example is for illustrative purposes only and is not suitable for production use. In a production environment, it is important to handle errors appropriately by checking return values and using try-catch blocks around sections of code that may throw exceptions, to ensure that DTAPI operates as expected.

2.2.1. Step 1: Attach to the Hardware

```
// 1. Declare the DTA-2110 device object and attach to the hardware.
DtDevice Dta2110{};
Dta2110.AttachToType(2110);
```

To use the A/V FIFO classes, instantiate a **DtDevice** object and attach it to the DTA-2110, following the instructions of the "DTAPI Manual – Overview and Data Formats".

Note

- Several **DtDevice** methods are available to attach to the hardware. The example uses **DtDevice::AttachToType()** to attach to the first available DekTec adapter with the specified type number.

2.2.2. Step 2: Initialize Receive FIFO

```
// 2. Declare the RxFifo object, attach to port 1 of the DTA-2110, and configure for reception.
AvFifo::RxFifo RxFifo{};
RxFifo.Attach(Dta2110, 1, HwOrSwPipe::PreferHwPipe);
RxFifo.SetIpPars({…});
RxFifo.Configure({…});
```

After attaching the **DtDevice** object to the hardware, instantiate an **RxFifo** (Receive FIFO) object and attach it to the device object. A parameter of the **RxFifo::Attach()** call allows specifying a preference for a hardware or software pipe.

2.2.3. Step 3. Start the Receive FIFO

```
// 3. Start the RxFifo.
RxFifo.Start();
```

Starting the **RxFifo** instructs the hardware to begin receiving IP packets on the configured IP address and port. When a frame (audio/video/ ancillary data unit) is received, it is written to the **RxFifo** and its load is incremented.

2.2.4. Step 4. Main Loop.

```
// 4. Keep receiving until the stop condition is met.
while (!StopCondition())
```

The main loop continues until a stop condition is met.

Note

- The stop condition will have to be set from another thread.

2.2.5. Step 5. Read and Process Frame

```
    // 5. Read and process a frame from the RxFifo if one is available; otherwise, sleep briefly.
    if (RxFifo.GetFifoLoad() > 0)
    {
        AvFifo::Frame* Frame{RxFifo.Read()};
        ProcessReceivedFrame(Frame);
        RxFifo.ReturnToMempool(Frame);
    } else
        std::this_thread::sleep_for(10ms);
```

In the main loop, check the **RxFifo's** load to determine if one or more frames are available.

- If a frame is available, read it from the **RxFifo**, process it and return it to the Frame Memory Pool for recycling.
- If no frame is available, sleep for a short period of time and try again.

2.2.6. Step 6. Clean Up

```
// 6. Clean up.
RxFifo.Stop();
RxFifo.Detach();
Dta2110.Detach();
```

Once the stop condition is met, stop the **RxFifo**, detach it from the **DtDevice** object, and detach the **DtDevice** object from the hardware if no further actions are needed.

2.3. Basic Transmit Example

The following code example demonstrates how to set up DTA-2110 with a **`TxFifo`** object for transmitting data in a simple transmission scenario.

```
// 1. Declare DTA-2110 device object and attach to the hardware.
DtDevice Dta2110{};
Dta2110.AttachToType(2110);

// 2. Declare TxFifo object, attach to port 1 of the DTA-2110, and configure for transmission.
AvFifo::TxFifo TxFifo{};
TxFifo.Attach(Dta2110, 1, HwOrSwPipe::PreferHwPipe);
TxFifo.SetIpPars({…});
TxFifo.Configure({…});
size_t FrameSize{…};          // Compute frame size matching configuration parameters.

// 3. Initialize ToD to current time. Add 100ms to have time to create the first few frames.
DtTimeOfDay ToD{};
Dta2110.GetTimeOfDay(ToD);
ToD += 100'000'000;           // 100,000,000ns = 100ms

// 4. Start the TxFifo.
TxFifo.Start();

// 5. Keep transmitting until the stop condition is true.
while (!StopCondition())
{
    // Here is a good place to check the TxFifo status with VideoRxFifo.GetStatus().

    // 6. We only write a new Frame to the TxFifo if it has space, otherwise we sleep.
    if (TxFifo.GetFifoLoad() < TxFifo.GetMaxSize())
    {
        // 7. Ensure frame is transmitted at the correct time by aligning to the video media grid.
        ToD = AvFifo::Tod2Grid_Video(ToD, {50, 1});

        // 8. Obtain a Frame from the memory pool, and assemble a video frame in it.
        AvFifo::Frame* Frame{TxFifo.GetFrameFromMemPool(FrameSize)};
        Frame->ToD = ToD;
        Frame->RtpTime = AvFifo::St2110::Tod2Rtp_Video(ToD);
        CreateVideoFrame(Frame);

        // 9. Write the newly assembled Frame to the TxFifo for transmission.
        TxFifo.Write(Frame);

        // 10. Advance the time-of-day (ToD) to the next frame time, which is 20ms later.
        ToD += 20000000;       // 20,000,000ns = 20ms
    } else {
        std::this_thread::sleep_for(10ms);
    }
}

// 11. Break down.
TxFifo.Stop();
TxFifo.Detach();
Dta2110.Detach();
```

Note This code example is for illustrative purposes only and is not suitable for production use. In a production environment, it is important to handle errors appropriately by checking return values and using try-catch blocks around sections of code that may throw exceptions, to ensure that DTAPI operates as expected.

Let us proceed by examining the example in a step-by-step manner.

2.3.1. Step 1: Attach to the Hardware

```
// 1. Declare DTA-2110 device object and attach to the hardware.
DtDevice Dta2110{};
Dta2110.AttachToType(2110);
```

Before utilizing the A/V FIFO classes, declare a **DtDevice** object and attach it to the hardware, in this case to a DekTec SMPTE 2110 NIC. This step corresponds to the instructions detailed in Section 3.1 of the "DTAPI Manual – Overview and Data Formats."

Several **DtDevice** methods are available for attaching to the hardware. In this example, we use **DtDevice::AttachToType()**, which establishes a connection with the first available DekTec adapter that matches the specified type number.

2.3.2. Step 2: Initialize Transmit FIFO

```
// 2. Declare TxFifo object, attach to port 1 of the DTA-2110, and configure for transmission.
AvFifo::TxFifo TxFifo{};
TxFifo.Attach(Dta2110, 1, HwOrSwPipe::PreferHwPipe);
TxFifo.SetIpPars({…});
TxFifo.Configure({…});
size_t FrameSize{…};          // Compute frame size matching configuration parameters.
```

Once a **DtDevice** object attached to the hardware is available, the next step is to create a **TxFifo** (Transmit FIFO) object and attach it to the **DtDevice** object. During the Attach call, a parameter is available to specify a preference for a hardware or a software pipe.

2.3.3. Step 3. Initialize Time-of-Day

```
// 3. Initialize ToD to current time. Add 100ms to have time to create the first few frames.
DtTimeOfDay ToD{};
Dta2110.GetTimeOfDay(ToD);
ToD += 100’000’000;           // 100,000,000ns = 100ms
```

When writing frames to the **TxFifo**, timestamps are required to indicate the desired transmission time for each frame. To ensure smooth transmission and accommodate potential OS scheduling jitter, frames should be prepared and buffered in the **TxFifo** ahead of time. In this example, we work 100ms ahead (which could easily be increased to, for instance, 200ms). To calculate the timestamp for the first frame to be transmitted, we obtain the current time and add 100ms.

2.3.4. Step 4. Start the TxFifo

```
// 4. Start the TxFifo.
TxFifo.Start();
```

With the configuration complete, the **TxFifo** can be started. Note that starting with an empty TxFifo is valid and will not lead to an exception. Once the first frame has been written to the **TxFifo**, it will be read by the AvFifo-internal internal transmit thread and scheduled.

2.3.5. Step 5. Main Loop.

```
// 5. Keep transmitting until the stop condition is true.
while (!StopCondition())
{
```

The main loop continues until a stop condition is detected. This condition can only be set from another thread.

2.3.6. Step 6. Check for Sufficient Space

```
    // 6. We only write a new Frame to the TxFifo if it has space, otherwise we sleep.
    if (TxFifo.GetFifoLoad() < TxFifo.GetMaxSize())
    {
```

We only write a new Frame to the **TxFifo** if it has space, otherwise we sleep briefly and check again.

2.3.7. Step 7. Compute Timestamp

```
        // 7. Ensure frame is transmitted at the correct time by aligning to the video media grid.
        ToD = AvFifo::Tod2Grid_Video(ToD, {50, 1});
```

To generate a SMPTE 2110 compliant stream, the transmission timestamp must be aligned to the video media grid, aka "Media Clock". The **`AvFifo::Tod2Grid_Video()`** function can perform this alignment. To be able to do so, it needs the frame rate as an exact fraction.

2.3.8. Step 8. Assemble a Frame

```
        // 8. Obtain a Frame from the memory pool, and assemble a video frame in it.
        AvFifo::Frame* Frame{TxFifo.GetFrameFromMemPool(FrameSize)};
        Frame->ToD = ToD;
        Frame->RtpTime = AvFifo::St2110::Tod2Rtp_Video(ToD);
        CreateVideoFrame(Frame);
```

At this stage, the application needs to create a Frame with a transmission timestamp. To achieve this, first obtain a Frame containing an embedded frame BLOB from the **`TxFifo`**'s Frame Memory Pool. Next, set the transmission and RTP timestamps and fill the BLOB with frame data using the **`CreateVideoFrame()`** function. The Frame is now prepared and ready to be written to the **`TxFifo`** in the subsequent step.

Note

- In **`CreateVideoFrame()`**, the application needs to set **`Frame->NumValidBytes`** to the number of valid data bytes in the Frame, which may be less than the Frame's size.
- The Frame obtained from the Frame Memory Pool will be returned to the AvFifo implementation in the next step.

2.3.9. Step 9. Write Frame to TxFifo

```
        // 9. Write the newly assembled Frame to the TxFifo for transmission.
        TxFifo.Write(Frame);
```

`TxFifo.Write()` writes the Frame to the **`TxFifo`**. The transmit scheduling thread running in the AvFifo implementation examines the Frames written to the TxFifo and transmits a Frame when the current time matches the Frame's timestamp.

Note

- After a Frame is transmitted, it is returned to the Frame Memory Pool for recycling.

2.3.10. Step 10. Advance Time-of-Day

```
        // 10. Advance the time-of-day (ToD) to the next frame time, which is 20ms later.
        ToD += 20000000;        // 20,000,000ns = 20ms
```

Advance the time-of-day (ToD) to the next frame time, which is 20ms later for the 50Hz frame rate used in this example.

2.3.11. Step 11. Clean Up.

```
// 11. Clean up.
TxFifo.Stop();
TxFifo.Detach();
Dta2110.Detach();
```

Once the stop condition is met, stop the **`TxFifo`**, detach it from the **`DtDevice`** object, and detach the **`DtDevice`** object from the hardware if no further actions are needed.

3. AvFifo Concepts

3.1. Frame Memory Pool

The Frame Memory Pool is designed to optimize performance and minimize memory overhead in real-time audio/video applications. Each **RxFifo** and **TxFifo** has its own dedicated Frame Memory Pool, ensuring efficient resource management. The memory pool acts as a dynamic buffer, allocating memory for frames on demand, and focuses on recycling frames to prevent constant memory allocation and deallocation.

Frames can only be allocated after the **RxFifo** or **TxFifo** is configured, so that the frame BLOB size is known. In the transmit scenario, users request a Frame from the memory pool, fill it with data, and write it to the **TxFifo**. Once transmitted, the Frame returns to the pool for reuse. In the receive scenario, the **RxFifo** acquires a Frame from the memory pool when a frame is received. Users read the Frame from the **RxFifo** and return it to the pool after processing.

The AvFifo API offers users an "observing pointer" to the frames while the memory pool retains ownership. The memory pool releases allocated frames when the **RxFifo** or **TxFifo** is cleared, reconfigured, or detached from the hardware.

3.2. Exceptions in AvFifo

3.2.1. General Philosophy

The AvFifo API uses C++ exceptions to signal exceptional conditions that may occur during operation. This approach to error handling eliminates the need for applications to examine result codes after each API call and allows API functions to return values in a more intuitive way.

All exception classes within AvFifo inherit from the **std::exception** class, providing a familiar and standardized interface for managing errors. The **std::exception** class includes the **what()** member function, which is also used by AvFifo exceptions to provide clear and concise error messages tailored to the specific error condition encountered.

3.2.2. Exception Categories

AvFifo exceptions can be classified into several categories, with each category being encoded in the class from which the exception is derived.

Derived from	Meaning
std::logic_error	Category of exceptions indicating that an AvFifo function has been invoked in a state where the function may not be called, or the function receives arguments that violate its logical preconditions. Encountering a UsageError typically indicates a programming mistake.
std::runtime_error	Category of exceptions denoting a runtime error condition.
std::bad_alloc	Indicates excessive memory usage, typically resulting from allocating too much memory for the Frame Memory Pools in the **Rx/TxFifo**s.
std::invalid_argument	Indicates that a function received an argument with a value that is outside the expected range.

3.3. Using Raw Mode

The AvFifo API features a "raw mode" that allows developers to directly access all payload bytes when receiving or transmitting frames with **RxFifo** or **TxFifo**. This ability to interact with raw data is particularly useful when dealing with SMPTE 2110 substandards not yet supported by AvFifo, or when experimenting with new or proprietary standards.

It's important to note that IP packets are transmitted using big endian byte order, while Intel CPUs typically process data in little endian byte order. Understanding the implications of this difference is crucial, and this section aims to discuss the ramifications of this contrast.

3.3.1. Network Byte Order

Network communications use big-endian byte order for the transmission of IP packets, commonly referred to as network byte order. When the AvFifo API receives IP packets, it stores the payload bytes in frames in the order of receipt. When handling big-endian data on a little-endian CPU, byte order conversion may sometimes be needed to correctly process the IP packet data.

RFC 791 provides a detailed description and illustration of the transmission order. It uses a type of diagram that is used in several other RFC's and in standards that extensively rely on RFC's such as SMPTE 2110. The following diagram, adapted from RFC 791, shows the bytes in order of transmission.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    byte 0     |    byte 1     |    byte 2     |    byte 3     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    byte 4     |    byte 5     |    byte 6     |    byte 7     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In this type of diagram, the leftmost bit, labeled 0, is the most significant bit. When a multi-byte integer is transmitted, the most significant byte is transmitted first.

3.3.2. Processing IP Packets in Network Byte Order with a Little-Endian CPU

In PCs, which use little-endian CPUs, bit 0 is typically the least significant bit. By retaining the layout and relabeling the bit numbers in the ASCII diagrams (0→7, 7→0, 8→15, 15→8, etc.), the RFC 791 diagram style can still be used.

For instance, the diagram above could be represented as follows on a little-endian CPU:

```
|7              0|15             8|22           16|31           23|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    byte 0     |    byte 1     |    byte 2     |    byte 3     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    byte 4     |    byte 5     |    byte 6     |    byte 7     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Labeling the bits in little endian order enables clarity in bit numbering. However, the byte ordering remains big endian. If, for example, byte 1 and 2 constitute a 16-bit integer in network byte order, the high and low order bytes should be swapped to get the same 16-bit value in little endian.

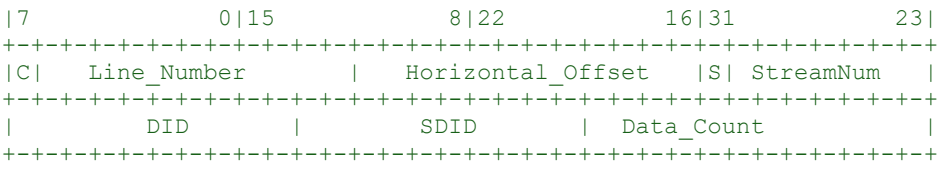
```
uint8_t Byte0, Byte1;
int16_t Qty16;

Qty16 = (int16_t)(Byte0 | (Byte1 << 8));
```

3.3.3. Example – Receiving SMPTE 2110-40 in Raw Mode

SMPTE 2110-40 addresses the transport of ancillary data, encompassing subtitles, time codes, and other metadata. While the AvFifo API does not currently offer built-in support for receiving or transmitting SMPTE 2110-40 data, these operations can be accomplished using raw mode.

In this section, we illustrate the utilization of raw mode for receiving SMPTE 2110-40 formatted ancillary data. A similar approach can also be employed for transmitting ancillary data.

As an example, let's extract the line number from the ancillary header defined in SMPTE 2110-40. This header is composed of big-endian fields and will appear in memory as follows, adopting the little-endian bit numbering scheme discussed earlier:



Bits 6 to 0 of byte 0 contain bits 10 to 4 of the Line Number, while byte 1 holds bits 3 to 0. The line number can be reconstructed using the following code:

```
uint8_t Byte0 = PayloadData[0];
uint8_t Byte1 = PayloadData[1];
int16_t LineNumber;

LineNumber  = ((int16_t)(Byte0 & 0x7F)) << 4;
LineNumber |= ((int16_t)(Byte1 & 0xF0)) >> 4;
```

This methodology can be extended to all fields, and also to transmission.

For the conversion of 32-bit integers between network byte order and little endian, the functions **`std::ntohl`** and **`std::htonl`** can be used.

3.3.4. Using Bit Fields

Alternatively, C/C++ bit fields can be employed. However, note that this method is somewhat less portable due to non-standardized allocation of bit fields, although it is compatible with both Visual Studio and gcc. Fields must be defined per byte, starting from the least significant to the most significant bit.

```
struct Smpte2110_40_Header
{
    unsigned LineNumberH : 7;
    unsigned C : 1;
    unsigned HorizontalOffsetH : 4;
    unsigned LineNumberL : 4;
    unsigned HorizontalOffsetL : 8;
    unsigned StreamNum : 7;
    unsigned S : 1;

    // Etc.
};

Smpte2110_40_Header* Hdr = (Smpte2110_40_Header*)PayloadData;
int16_t LineNumber = (Hdr.LineNumberH << 4) | Hdr.LineNumberL;
```

This approach is considered cleaner as it reduces the need for explicit bitwise operations and value shifting, resulting in more readable code. However, the portability of code utilizing bit fields may be compromised due to differences in how compilers or platforms handle them.

Consider the specific requirements of your project and any portability concerns when deciding whether to use bit fields or alternative bitwise operations.

enum class FifoStatus

Describes the status of an **RxFifo** or **TxFifo**.

```
enum class FifoStatus
{
   Ok,                        // No error, FIFO is operating correctly.
   DstMacResolveFail,         // Failed to resolve the destination MAC address.
   LinkDown,                  // Network link is down.
   MulticastJoinFail,         // Failed to register the multicast address.
   Unplugged                  // The network cable is disconnected or unplugged.
};
```

Values

Ok
The IP link is operational and functioning correctly.

LinkDown
The network link is completely down, possibly due to a disabled network port.

MulticastFail
There has been a failure in registering the multicast address.

ResolveFail
There has been a failure in resolving the destination MAC address, which may be due to a non-existent IP address.

Unplugged
The network cable is disconnected or unplugged.

Remarks

- Use function **FifoStatusToMessage()** for translating a **FifoStatus** value into a human-readable string,. This function will return a short descriptive string corresponding to the **FifoStatus** value.
- Always check the status of the IP link and handle different statuses appropriately in your code to ensure a smooth network operation.

enum class HwOrSwPipe

Indicates the user's preference for a software or hardware pipe.

Hardware pipes generally provide better performance and are more suitable for high-resolution video processing. Each hardware pipe has its own dedicated DMA (Direct Memory Access) controller, which allows for efficient and fast data transfers between the board and the system memory. In contrast, software pipes use a single DMA controller for a multiplex of all received or transmitted streams, which may result in reduced performance.

The number of available hardware pipes depends on the board type number. The DTA-2110, currently the only board that supports SMPTE 2110 hardware pipes, supports 3 hardware pipes and 3 software pipes.

```
enum class HwOrSwPipe
{
   Auto,                        // DTAPI chooses a suitable pipe type.
   ForceHwPipe,                 // Force hardware pipe.
   PreferHwPipe,                // Prefer hardware pipe, accept software pipe.
   UseSwPipe                    // Force a software pipe.
};
```

Values

Auto
DTAPI chooses a suitable pipe type.

ForceHwPipe
Forces the usage of a hardware pipe.

PreferHwPipe
Prefers the usage of a hardware pipe, but accepts a software pipe if no hardware pipe is available.

UseSwPipe
Use software pipe, bypassing any available hardware pipes.

Remarks

For handling audio and ancillary data, it is recommended to use a software pipe, as these types of data generally have lower bandwidth requirements compared to high-resolution video data. By delegating audio and ancillary data processing to software pipes, you can reserve the hardware pipes for more demanding video processing tasks, thus ensuring efficient utilization of the available resources on the board.

enum IpProtocolVersion

Enumerates the available IP protocol versions supported by the A/V FIFO interface.

```
enum class IpProtocolVersion
{
   IPv4,                        // IP protocol version 4 - 32-bit addresses.
   IPv6                         // IP protocol version 6 - 128-bit addresses.
};
```

Values

IPv4
IP protocol version 4. IPv4 uses 32-bit addresses.

IPv6
IP protocol version 6. IPv6 uses 128-bit addresses. In addition to a larger address space, IPv6 also includes improvements to security, quality of service (QoS), and other enhancements compared to IPv4.

Remarks

enum IpTransportProtocol

Selects between RTP-over-UDP and plain UDP without RTP.

```
enum class IpTransportProtocol
{
   Rtp,                         // RTP-over-UDP.
   Udp                          // UDP. Data is encapsulated in UDP without RTP.
};
```

Values

Rtp
RTP-over-UDP transport protocol.

Udp
Plain UDP transport protocol without RTP.

Remarks

enum RxFifoOverflowStrategy

Enumerates different strategies to manage situations in which the AvFifo API receives a frame while the **RxFifo** is already at maximum capacity, unable to store additional frames.

```
enum class RxFifoOverflowStrategy
{
   DropFrame,                  // When the RxFifo is full: Drop the received frame.
   ThrowException              // When the RxFifo is full: Throw exception.
};
```

Values

DropFrame

Indicates a 'strategy' to discard the most recently received frame when the **RxFifo** is full. This strategy is suitable for applications that can withstand occasional loss of frames. For instance, it could be useful in viewer or multiviewer applications, where occasional frame loss does not significantly disrupt the user experience.

ThrowException

Indicates a 'strategy' to throw an exception when the **RxFifo** is full while a new frame has been received. The exception is not immediately thrown at the time of overflow, but rather at the first subsequent user call to the **RxFifo::GetStatus()** method. This strategy is intended for real-time applications where dropping a frame could disrupt processing and lead to synchronization issues. Normally, overflow should not occur in such applications. If an overflow does occur, this strategy allows for immediate detection and suggests that the application should be restarted to regain synchronized processing.

Remarks

- The **DropFrame** strategy is appropriate for applications where the occasional loss of frames is acceptable. The **ThrowException** strategy is more suitable for real-time applications that can't afford to lose frames.
- In either case, it is crucial to implement measures to mitigate the impact of the selected strategy on your application. If the **DropFrame** strategy is used, consider notifying the user when a frame has been dropped. With the **ThrowException** strategy, ensure your application has sufficient exception handling mechanisms in place to prevent abrupt terminations and to handle the restart process smoothly.

enum St2022::FecMode

Specifies the operating mode of SMPTE 2022-1 Forward Error Correction (FEC) for a stream, indicating whether it is enabled or disabled. FEC appends error-correction bytes to the transmitted data, enabling the receiver to detect and correct transmission errors, thus enhancing communication reliability and robustness.

```
enum class FecMode
{
   Disable,                  // FEC is disabled.
   Enable                    // FEC is enabled: Tx adds FEC bytes; Rx: decodes FEC
                             // bytes if present and corrects data as needed.
};
```

Values

Disabled
: Indicates FEC is disabled. No SMPTE 2022-1 FEC bytes will be added during transmission. On reception, FEC data, even if present, will not be decoded.

Enabled
: Indicates FEC is enabled, allowing the stream to utilize SMPTE 2022-1 FEC's error detection and correction capabilities. With FEC enabled, FEC bytes are added during transmission, and during reception FEC bytes are decoded and data is corrected as needed.
Note that enabling SMPTE 2022-1 FEC is not supported in the current version of DTAPI.

Remarks

enum St2022::LinkMode

Enumerates the SMPTE 2022-7 link modes, allowing the selection between a single link and dual-link mode. The dual-link mode provides “seamless protection switching”, ensuring high reliability and redundancy in the transmission of professional media streams over IP networks.

```
enum class LinkMode
{
   Single,                    // Single link mode, uses a single network path.
   Dual,                      // SMPTE 2022-7 dual-link mode, provides redundancy.
};
```

Values

Single
: Single link mode, where the stream is transmitted over a single network path. This mode offers simplicity and lower overhead but lacks redundancy, making it more vulnerable to network failures and other issues.

Dual
: SMPTE 2022-7 dual-link mode, which uses two separate network paths for data transmission. This mode offers redundancy and seamless protection switching, allowing the receiver to switch between the two paths without interrupting the media stream. This ensures high reliability and resilience to network issues.

Remarks

enum St2110::AudioFormat

Enumerates the supported SMPTE 2110 audio formats, providing options for different bit depths and byte orders of Pulse Code Modulation (PCM) audio, as well as raw audio data.

```
enum class AudioFormat
{
   L16BE,                    // 16-bit PCM, big endian byte order.
   L24BE,                    // 24-bit PCM, big endian byte order.
   Raw
};
```

Values

L16BE
: Indicates 16-bit PCM audio with a big endian byte order.

L24BE
: Indicates 24-bit PCM audio with a big endian byte order. This format offers higher audio quality and dynamic range compared to 16-bit PCM, at the cost of increased data size.

Raw
: Indicates raw audio data without any specific formatting, providing flexibility for handling custom or non-standard audio formats.
When working with raw audio data, additional processing or conversion may be necessary to interpret or process the audio correctly.

Remarks

enum St2110::PackingMode

Enumerates the SMPTE 2110 video packing modes.

```
enum class PackingMode
{
   General,                    // Only pgroup size restriction.
   Block                       // Each packet must contain a multiple of 180 bytes.
};
```

Values

General

In the **General** packing mode, there are no specific restrictions on the packet size, except that it must adhere to the packet group (pgroup) size limitation. This mode offers flexibility in packet size.

Block

In the **Block** packing mode, each packet must contain a multiple of 180 bytes. This mode ensures a consistent and predictable packet size.

Remarks

enum St2110::RxFrameFormat

Enumerates the supported pixel formats for incoming SMPTE 2110 video.

Used to specify the pixel format on the line when configuring the **AvFifo::RxFifo**, so that AvFifo can assume that the video has the specified format.

```
enum class RxFrameFormat
{
   Raw,                        // Raw underlying format without any conversion.
   Uyvy422_8b,                 // 8-bit UYVY packed pixel format.
   Uyvy422_10b,                // 10-bit packed UYVY pixel format.
   Uyvy422_10b_to_8b,          // 10-bit packed UYVY converted to 8-bit UYVY.
   Yuv422p_8b                  // 8-bit planar YUV (3 planes: Y,U,V).
};
```

Values

Raw
Specifies that the raw video data without any conversion should be put in the **RxFifo**.

Uyvy422_8b
Specifies an 8-bit UYVY packed pixel format, where the chroma and luma components are interleaved.

Uyvy422_10b
Specifies a 10-bit packed UYVY pixel format with higher color depth compared to the 8-bit format.

Uyvy422_10b_to_8b
Represents a 10-bit packed UYVY pixel format on the line. When the video data is received by the **RxFifo**, it is first converted to an 8-bit UYVY format by DTAPI, for faster and more convenient processing.

Yuv422p_8b
Specifies an 8-bit planar YUV format with separate planes for Y, U, and V components.

Remarks

The **St2110::RxFrameFormat** enumeration specifies the pixel format of incoming SMPTE 2110 video streams. This ensures that **AvFifo::RxFifo** processes the video data using the correct format.

enum St2110::Scheduling

Enumerates the SMPTE 2110 packet scheduling method.

```
enum class Scheduling
{
   Linear,                    // Packets are transmitted evenly spaced in time.
   Gapped                     // Packets are sent with variable gaps between them.
};
```

Values

Linear

In the **Linear** scheduling method, packets are transmitted at evenly spaced time intervals. This approach ensures a consistent and predictable transmission rate.

Gapped

In the **Gapped** scheduling method, packets are transmitted with variable gaps between them. This approach allows for more flexibility in packet transmission.

Remarks

enum St2110::TxFrameFormat

Enumerates the supported pixel formats for transmitting SMPTE 2110 video. The **`AvFifo::TxFifo`** class assumes that the video frame data supplied by the application adheres to one of these formats.

```
enum class VideoScanning
{
   Uyvy422_8b,                // 8-bit UYVY pixel format.
   Uyvy422_10b                // 10-bit packed UYVY pixel format.
};
```

Values

`Uyvy422_8b`
Specifies an 8-bit UYVY packed pixel format, where the chroma and luma components are interleaved.

`Uyvy422_10b`
Specifies a 10-bit packed UYVY pixel format with higher color depth compared to the 8-bit format.

Remarks

enum St2110::VideoScanning

Enumerates the supported video scanning modes for received and transmitted video.

```
enum class VideoScanning
{
   Progressive,               // Full frames, no separation in odd and even lines.
   Interlaced,                // Alternating odd lines of a frame are placed in a
                              // field, followed by a field with the even lines of
                              // the next frame.
   PsF                        // Each frame is split into 2 fields, one containing
                              // odd lines, another containing even lines.
};
```

Values

Progressive
Each frame is transmitted in its entirety.

Interlaced
The odd lines of a frame are placed in a field, followed by placing the even lines of the next frame in the next field. This alternates between odd and even lines of consecutive frames.

PsF
Progressive video is transmitted as Interlaced video, but with double the field rate. Each frame is split into two fields, one containing the odd lines, and another containing the even lines.

Remarks

Exceptions

The AvFifo exception classes are all derived from the exceptions defined in the C++ standard library (**std**).

```
class DriverError        : public std::runtime_error { … };
class HardwarePipeUnavailable: public std::runtime_error { … };
class InvalidFormatError: public std::runtime_error { … };
class OverflowError      : public std::runtime_error { … };
class SchedulingError    : public std::runtime_error { … };
class UsageError         : public std::logic_error { … };

class bad_alloc          : public std::exception { … };
class invalid_argument   : public std::logic_error { … };

const char* Exception::what() const noexcept;
```

Exceptions

Exception	Derived from std::	Meaning
DriverError	**runtime_error**	Occurs when the AvFifo implementation calls the network or PCIe driver, signaling an issue with the driver. To resolve, check driver version and update if necessary.
HardwarePipe Unavailable	**runtime_error**	Signals that a hardware pipe is unavailable, potentially because all hardware pipes are currently in use by this or other processes.
InvalidFormatError	**runtime_error**	Occurs when a user writes a Frame to the **TxFifo** with a format that doesn't comply with the configured format. This points to an error in the code creating Frames.
OverflowError	**runtime_error**	Occurs when the AvFifo receive thread has a new Frame available, but the **RxFifo** has reached its maximum capacity. Since the receive thread cannot throw exceptions, this error is signaled from **RxFifo::GetStatus()**.
SchedulingError	**runtime_error**	Occurs when the AvFifo transmit thread encounters an invalid timestamp that is too far in the future or the past. Since the transmit thread cannot throw exceptions, this error is signaled from **TxFifo::GetStatus()**.
UsageError	**logic_error**	Occurs when an AvFifo function is called in an inappropriate state or receives arguments that violate its preconditions, typically indicating a programming mistake.
Standard Library Exceptions		
bad_alloc	**exception**	Occurs when a memory allocation request fails, typically caused by using too much memory for the Frame Memory Pools in the **Rx/TxFifo**s.
invalid_argument	**logic_error**	Occurs when an AvFifo method receives an argument with an inappropriate or invalid value.

Members

what()
Returns a C-style string representing a human-readable error message describing the exception.

Remarks

- All AvFifo exceptions are derived from **std::exception** and support the **what()** function.

struct BlobMetadata

Encapsulates metadata needed for allocating frame BLOBs. This metadata is used by **`RxFifo`** and **`TxFifo`** objects whenever a new frame needs to be allocated. The struct provides parameters for setting alignment and additional space requirements for the BLOB.

```
struct BlobMetadata
{
   int Alignment{32};          // Address alignment (e.g. 32 = 32-byte alignment).
   int ExtraSize{0};           // Extra bytes allocated at the end of the BLOB.
};
```

Members

`Alignment`

Sets the alignment of the start address of the frame BLOB. The alignment is defined as the number of bytes to which the start address of the BLOB must align. For instance, an alignment of 32 means that the BLOB's start address will be a multiple of 32 bytes. The alignment value must be a power of 2.

`ExtraSize`

Defines the additional number of bytes that should be allocated at the end of the BLOB. This extra space is typically utilized to facilitate algorithms that process data in groups of pixels, e.g. with SSE/AVX instructions. By allocating extra bytes, these algorithms can safely operate without the risk of accessing or overflowing into memory space beyond the allocated BLOB. *`ExtraSize`* defaults to zero, indicating no additional bytes are allocated by default.

Remarks

- *`BlobMetadata`* enables customized memory management for frame BLOBs. It's crucial to correctly set *`Alignment`* and *`ExtraSize`* to ensure efficient and safe memory operations.

struct ExactRatio

Represents an exact rational number as a fraction with a numerator and a denominator. It's useful for precise representation and manipulation of fractional frame rates, such as 29.97 (represented exactly as 30000/1001), and helps avoiding rounding errors from floating-point approximations.

```
struct ExactRatio
{
   int Numerator{-1};         // Represents the numerator of the rational number.
   int Denominator{-1};       // Represents the denominator of the rational number.
};

// Alias 'FrameRate' to 'Ratio' for representing video frame rates as exact
// rational numbers.
using FrameRate = ExactRatio;
```

Members

Numerator
Stores the integer value of the numerator part of the rational number.

Denominator
Stores the integer value of the denominator part of the rational number.

Remarks

- Using **ExactRatio** helps preventing synchronization issues over time by providing exact fractional frame rate representations, avoiding floating-point approximations.

struct IpPars

Defines IP parameters for transmission or reception of SMPTE 2110 and SMPTE 2022 streams, including IP address, port, and various optional settings to customize and optimize communication.

```
struct IpPars
{
   array<uint8_t, 16> IpAddr{};            // IPv4 or IPv6 address.
   IpProtocolVersion IpVersion{IPv4};      // IP protocol version.
   int Port{-1};                           // IP port number.
   vector<IpSrcFlt> SrcFlt{};              // Optional SSM filter.
   int DiffServ{34 << 2};                  // Differentiated service field.
   array<uint8_t, 16> Gateway{};           // Optional gateway address.
   int RtpPayloadType{0};                  // RTP payload packet type.
   int TimeToLive{64};                     // Time to live (TTL) value.
   IpTransportProtocol TransportProtocol{Udp};
   struct {
     int Id{0};                            // ID for network segmentation.
     int Priority{0};                      // Priority for traffic prioritization.
   } Vlan;                                 // VLAN parameters.
};
```

Members

IpAddr
Holds the IP address, either IPv4 or IPv6.

IpVersion
Specifies the IP protocol version, either IPv4 or IPv6.

Port
Defines the IP port number.

SrcFlt
Optional source-specific multicast (SSM) filter to manage incoming multicast traffic. The filter consists of a vector of IP addresses/port numbers to choose which sources are allowed.

DiffServ
Sets the Differentiated Services Field (DS Field) value for Quality of Service (QoS) purposes.

Gateway
Optional gateway address for routing purposes.

RtpPayloadType
Defines the payload type that will be set in the RTP packet.

TimeToLive
Specifies the Time to Live (TTL) value for IP packets during transmission.

TransportProtocol
Indicates the transport protocol to be used, either UDP or RTP.

Vlan
Contains VLAN parameters, including the VLAN ID and priority for network segmentation and traffic prioritization.

Remarks

struct IpSrcFilt

Defines the address filter for source-specific multicast, allowing control over incoming multicast traffic based on the source IP address and port.

```
struct IpSrcFilt
{
   array<uint8_t, 16> IpAddr{};            // 4 or 16 address bytes representing
                                           // the IPv4 or IPv6 source address.
   int Port{-1};                           // Source port number for filtering.
};
```

Members

IpAddr
: Holds the source IP address for the multicast traffic filter. The array can contain either 4 bytes for an IPv4 address or 16 bytes for an IPv6 address.

Port
: Specifies the source port number for filtering the multicast traffic. By setting a specific port, the filter will only allow multicast traffic from the source IP address and the specified port.

Remarks

Utilizing the IpSrcFilt struct, you can define a source-specific multicast (SSM) filter to enhance control over incoming multicast traffic. SSM allows you to accept data exclusively from specific sources, based on the source IP address and port number.

struct RxStatistics

Stores reception statistics as a structure containing several counter values.

```
struct RxStatistics
{
   int FramesOk{0};           // Number of correctly received frames.
   int FramesIncomplete{0};   // Number of frames with missing IP packets.
   int FramesSizeError{0};    // Number of frames with unexpected size.
   int Gaps{0};               // Number of discontinuities in complete frames.
   int IpPacketErrors{0};     // Number of IP packets with corrupted headers.
   int DroppedFrames{0};      // Number of dropped frames due to FIFO full.
   int SyncErrors{0};         // Number of out-of-sync errors.
};
```

Members

FramesOk
Counts the number of correctly received frames.

FramesIncomplete
Counts the number of frames with missing IP packets.

FramesSizeError
Counts the number of frames with unexpected size.

Gaps
Counts the number of frames with discontinuities in the RTP sequence numbers at the start of a new frame (after a frame end is detected).

IpPacketErrors
Counts the number of IP packets with an unrecognized header or with a syntax error in the header.

DroppedFrames
Counts the number of frames dropped because the receive FIFO was full.

SyncErrors
Counts the number of out-of-sync errors.

Remarks

All statistics will be reset when the **`RxFifo::Start`** function is called. This ensures that the statistics only reflect the data received since the last time the **`RxFifo`** was started, providing an accurate characterization of the current reception process.

struct TxStatistics

Stores reception statistics as a structure containing several counter values.

```
struct TxStatistics
{
   int FramesOk{0};           // Number of frames transmitted.
};
```

Members

FramesOk
Counts the number of transmitted frames.

Remarks

All statistics will be reset when the **TxFifo::Start** function is called. This ensures that the statistics only reflect the data received since the last time the **TxFifo** was started, providing an accurate characterization of the current reception process.

struct VideoSize

Represents the dimension of a video signal in number of pixels.

```
struct VideoSize
{
   int Width{-1};              // The width of the video, in pixels.
   int Height{-1};             // The height of the video, in pixels.
};
```

Members

Width
The width of the video, in pixels. Initialized to -1 to indicate that it hasn't been set yet.

Height
The height of the video, in pixels. Initialized to -1 to indicate that it hasn't been set yet.

Remarks

struct YuvPlanes

Contains pointers and sizes for 8-bit 4:2:2 YUV formatted video data in a **Frame**, assuming that the video format is **RxFrameFormat::Yuv422p_8b**. This struct is returned by **Frame::Yuv422P_8b_GetPlanes**.

```
struct YuvPlanes
{
   uint8_t* Y{nullptr};       // Pointer to the Y plane (luminance).
   uint8_t* U{nullptr};       // Pointer to the U plane.
   uint8_t* V{nullptr};       // Pointer to the V plane.
   int SizeY{0};              // Size of the Y plane in bytes.
   int SizeU{0};              // Size of the U plane in bytes.
   int SizeV{0};              // Size of the V plane in bytes.
};
```

Members

Y
A pointer to the start of the Y plane, which contains luminance samples.

U
A pointer to the start of the U plane, which contains chrominance samples representing the blue color difference.

V
A pointer to the start of the V plane, which contains chrominance samples representing the red color difference.

SizeY
The size of the Y plane in bytes, indicating the amount of memory allocated for the luminance data.

SizeU
The size of the U plane in bytes, indicating the amount of memory allocated for the blue color difference (U) data.

SizeV
The size of the V plane in bytes, indicating the amount of memory allocated for the red color difference (V) data.

Remarks

struct St2022::RxConfig

Configures the reception of an ST2022-5/6 stream.

```
struct RxConfig
{
   bool EnableFecDecoding{false};          // Enables FEC decoding.
   struct {
      IpPars IpParsLink2;                  // IP parameters for redundant link if
                                           // ST2022-7 mode is 'Dual'.
      LinkMode Mode{Single};               // Single or dual link.
      DtIpProfile Profile;                 // Maximum bitrate and maximum skew.
   } St2022_7;
};
```

Members

EnableFecDecoding
Enables or disables FEC decoding. If false, FEC decoding is disabled even if FEC packets are present.

IpParsLink2
IP parameters for the redundant link if ST2022-7 *Mode* is **LinkMode::Dual**. Note that parameters of the main link can be specified with **RxFifo::SetIpPars**.

Mode
Single link (no network redundancy), or dual link (network redundancy).

Profile
Configuration: Maximum bitrate and maximum skew between path 1 and path 2.

Remarks

struct St2022::TxConfig

Configures the transmission of an SMPTE 2022-5/6 stream.

```
struct TxConfig
{
   int VideoStandard{-1};                  // DTAPI_VIDSTD_...
   int TrOffset{-1};                       // PTP transmit time offset.
   int PayloadSize{-1};
   St2022::FecMode FecMode{Disable};       // Add FEC bytes?

   struct {
      IpPars IpParsPath2;                  // IP parameters for redundant link.
      LinkMode Mode{Single};               // Single or dual link mode.
   } St2022_7;                             // SMPTE 2022-7 parameters.
};
```

Members

VideoStandard
Specifies the video standard to be encoded, see the **DTAPI_VIDSTD_...** definitions.

TrOffset
The PTP (Precision Time Protocol) transmit time offset with respect to the most recent integer multiple of the time period between consecutive frames of video at the prevailing frame rate, starting from the EPOCH time.
If the value is -1, a default value is used.

PayloadSize
Specifies the size of the payload for the SMPTE 2022 stream. If the value is -1, the default size is used.

FecMode
Configures the Forward Error Correction (FEC) mode for the transmission. Possible values are defined in the **St2022::FecMode** enumeration.

St2022_7
Contains the SMPTE 2022-7 parameters, including:

IpParsPath2
Specifies the IP parameters for the redundant link when the SMPTE 2022-7 mode is set to 'Dual'.

Mode
Determines the link mode for the SMPTE 2022-7 transmission. Possible values are defined in the **St2022::LinkMode** enumeration: **Single** or **Dual**.

Remarks

struct St2110::RxConfigAudio

Configures the reception parameters of an SMPTE 2110 audio stream.

```
struct RxConfigAudio
{
   AudioFormat Format{Raw};   // Specifies the audio format.
   int SampleRate{48000};     // Defines the audio sample rate in Hz.
};
```

Members

Format
: Represents the audio format used for the SMPTE 2110 audio stream. It can be one of the following: 16-bit PCM, 24-bit PCM, or raw. The default value is set to **AudioFormat::Raw**.

SampleRate
: Defines the audio sample rate for the SMPTE 2110 audio stream, expressed in Hertz (Hz). The default sample rate is set to 48,000 Hz.

Remarks

The **St2110::RxConfigAudio** struct is used to set up and configure the reception parameters for an SMPTE 2110 audio stream.

struct St2110::RxConfigRaw

Configures the parameters for the reception of raw SMPTE 2110 payload data using Frame structs. It facilitates a flexible interface for raw data extraction, so that users have access to all data fields. Optionally, the raw frame data can include the RTP header.

```
struct RxConfigRaw
{
   bool IncludeRtpHeader{false};          // RTP header included in payload?
   int MaxRate{-1};                       // Max expected rate (bytes per second).
};
```

Members

IncludeRtpHeader
Determines whether the RTP header will be included in the frame data.

MaxRate
Specifies the maximum expected data rate, in bytes per second. This rate is used to set the default size of the shared buffer. Users are required to replace the default value of -1 with a valid rate; failure to do so will result in an error.

Remarks

- The **RxConfigRaw** struct is constructed for usage scenarios that require direct access to raw SMPTE 2110 data.
- Raw mode can be used to receive SMPTE 2110-40 ancillary data. Refer to §3.3 for an explanation and example of the usage of raw mode.
- Proper configuration of the **MaxRate** field is crucial to correctly dimension the shared buffer.

struct St2110::RxConfigVideo

Configures the parameters for receiving an SMPTE 2110 video stream.

```
struct RxConfigVideo
{
   RxFrameFormat Format{Raw};              // Specifies the video format.
};
```

Members

Format
Represents the pixel format of the incoming SMPTE 2110 video stream. It can be one of the following: 8-bit UYVY, 10-bit packed UYVY, 10-bit packed UYVY converted to 8-bit UYVY, or the raw underlying format without any conversion. The default value is set to **RxFrameFormat::Raw**.

Remarks

The **St2110::RxConfigVideo** struct is used to set up and configure the parameters for receiving an SMPTE 2110 video stream through a RxFifo (**RxFifo**).

struct St2110::TxConfigAudio

Configures the parameters for transmitting an SMPTE 2110 audio stream.

```
struct TxConfigAudio
{
   AudioFormat Format{L16BE};              // Audio format.
   int NumChannels{2};                     // Number of channels.
   int NumSamplesPerIpPacket{-1};          // Number of samples per IP packet.
   int SampleRate{48000};                  // Defines the audio sample rate in Hz.
};
```

Members

Format
Defines the audio format used for the SMPTE 2110 audio stream. It can be one of the following: 16-bit PCM, 24-bit PCM, or raw. The default value is set to **AudioFormat::Raw**.

NumChannels
Defines the number of audio channels. The default value is 2 for stereo audio.

NumSamplesPerIpPacket
Defines the number of samples packaged per IP packet.

SampleRate
Defines the audio sample rate for the SMPTE 2110 audio stream, expressed in Hertz (Hz). The default sample rate is set to 48,000 Hz.

Remarks

The **St2110::TxConfigAudio** struct is used to set up and configure the transmission parameters for an SMPTE 2110 audio stream.

struct St2110::TxConfigRaw

Configures the parameters for transmitting raw SMPTE 2110 payload data. This mode gives users the flexibility to fully customize the contents of the payload. Users can choose to supply their own RTP header or allow the AvFifo API to generate it automatically.

```
struct TxConfigRaw
{
   bool CustomRtpHeader{false};            // RTP header in payload?
   int MaxRate{-1};                        // Maximum rate in bytes per second.
};
```

Members

CustomRtpHeader
: Indicates whether a custom RTP header will be included in the payload data. When set to true, the user is obligated to provide an RTP header as part of the frame data.

MaxRate
: Specifies the maximum rate that will be transmitted in bytes per second. This value is used for configuring the default size of the shared buffer. The default value of -1 must be replaced by a valid rate, otherwise an error will be generated.

Remarks

- The **TxConfigRaw** struct is designed for advanced usage scenarios where complete control over payload content, possibly including a custom RTP header, is required.
- Raw mode can be used to transmit SMPTE 2110-40 ancillary data. Refer to §3.3 for an explanation and example of the usage of raw mode.
- When **CustomRtpHeader** is set to true, the user assumes responsibility for supplying a suitable RTP header as part of the frame data to be transmitted.

struct St2110::TxConfigRawVideo

Configures the parameters for transmitting raw SMPTE 2110 video. The **TxConfigRawVideo** structure lets you control all aspects of video transmission, including active video ratio, chroma subsampling, frame dimensions, packing parameters, pixel groups, row size, timing parameters, and PTP transmit time offset.

```
struct TxConfigRawVideo
{
   Ratio ActiveVideo{};       // Active video time relative to total line time.
   bool Is420{false};         // 4:2:0 chroma subsampling?
   int NumRows{-1};           // Total number of rows in a frame.
   VideoPacking Packing{};    // Packet packing parameters.
   struct {
      int NumBytes{-1};       // Number of bytes in a ST2110-20 pgroup.
      int NumPixels{-1};      // Number of pixels in a ST2110-20 pgroup.
   } PGroup;
   int RowSize{-1};           // Total number of bytes in a row.
   VideoTiming Timing{};      // Video timing parameters.
   int TrOffset{-1};          // PTP transmit time offset.
};
```

Members

ActiveVideo
Specifies the active video time relative to the total line time. This parameter determines the proportion of active video data within each transmitted line.

Is420
Specifies whether 4:2:0 chroma subsampling is to be used for video transmission.

NumRows
Specifies the total number of rows in a video frame. This parameter defines the vertical resolution of the transmitted video.

Packing
Specifies the packet packing parameters, with constraints on packet contents, packing mode, and payload size.

PGroup
A structure with two integer members, *NumBytes* and *NumPixels*, which represent the number of bytes and pixels in a ST2110-20 pixel group (pgroup), respectively.

RowSize
Specifies the total number of bytes in a row of video data.

Timing
Specifies the video timing parameters, such as frame rate, packet scheduling method, and scanning mode.

TrOffset
Specifies the PTP (Precision Time Protocol) transmit time offset with respect to the most recent integer multiple of the time period between consecutive frames of video at the prevailing frame rate, starting from the EPOCH time.
If the value is -1, a default value is used.

Remarks

struct St2110::TxConfigVideo

Configures the parameters for transmitting SMPTE 2110 video.

```
struct TxConfigVideo
{
   TxFrameFormat Format{Uyvy422_10b};
   VideoPacking Packing;      // Video packing parameters.
   VideoSize Resolution;      // Resolution of the active video.
   VideoTiming Timing;        // Video timing parameters.
};
```

Members

Format
: Specifies the video frame format. The default value is **TxFrameFormat**::**Uyvy422_10b**, which represents a 10-bit UYVY 4:2:2 format. Other supported formats can be found in the **St2110**::**TxFrameFormat** enumeration.

Packing
: Specifies the packing parameters for the video frames. These parameters include the storage layout and organization of pixel data within the video frame.

Resolution
: Sets the resolution of the active video. The **VideoSize** structure contains width and height values, specifying the dimensions of the video frame in pixels.

Timing
: Configures the video timing parameters. These parameters include frame rate, interlacing, and synchronization details, ensuring the proper display and synchronization of the transmitted video.

Remarks

struct St2110::VideoPacking

Specifies the SMPTE 2110 video packing parameters.

```
struct VideoPacking
{
   bool OneLinePerPacket{false};          // Only allow data from a single line
                                          // in one packet.
   St2110::PackingMode PackingMode{General};
   int PayloadSize{-1};                   // Specifies the payload size used.
};
```

Members

OneLinePerPacket
When set to true, restricts data to one line per packet. This ensures that each packet only contains data from a single line, simplifying processing.

PackingMode
Determines the video packing mode to be used. Available options are **General** (flexible packet size, adhering to packet group size limitations) and **Block** (packets must contain a multiple of 180 bytes).

PayloadSize
Specifies the payload size used in the video packets. A value of -1 indicates that the payload size will be automatically set to a reasonable size.

Remarks

struct St2110::VideoTiming

Configures the timing aspects of SMPTE 2110 video transmission.

```
struct VideoTiming
{
   FrameRate Rate;            // Frame rate; field rate for Interlaced/PsF.
   St2110::Scheduling Scheduling{Gapped};
   St2110::VideoScanning VideoScanning{Progressive};
};
```

Members

Rate
: Specifies the frame rate for progressive scanning modes, or the field rate for interlaced or PsF scanning mode. This parameter is crucial for determining the timing and synchronization of video transmission.

Scheduling
: Defines the packet scheduling method to be used. Available options are **Linear** (evenly spaced packet transmission) and **Gapped** (variable gaps between packet transmissions).

VideoScanning
: Sets the scanning mode for the video transmission. Options include **Progressive**, **Interlaced**, and **PsF** (progressive segmented frame).

Remarks

Frame *(namespace Dtapi::AvFifo)*

Frame public members

Represents a timestamped binary large object (BLOB) in a media stream, which serves as the fundamental unit of data for both receiving and transmitting video, audio, or ancillary data through the A/V FIFO interface.

Important Note – For progressive video, a **Frame** object represents a complete video frame. However, for interlaced or PsF (Progressive Segmented Frame) video, a **Frame** object represents a single video **FIELD** instead.

```
struct Frame
{
   uint32_t RtpTime{0};       // RTP timestamp assigned to RTP packets of the frame.
   DtTimeOfDay ToD{};         // Time-of-day of the first sample of the frame.

   uint8_t* Data{nullptr} const; // BLOB containing the actual frame/field data.
   int Field{0};              // Video only: field indication for Interlaced/PsF.
   int NumValidBytes{0};      // Number of valid bytes in the frame-data BLOB.
   size_t Size() const;       // Gets BLOB size in bytes (excl. oversizing).

   // Helper functions meaningful only when SMPTE 2110 video frames are received.
   bool Is420() const;        // Is 4:2:0 chroma subsampling used?
   int NumRows() const;       // Gets total number of rows in a video frame.
};
```

Members

RtpTime
Indicates the RTP timestamp assigned to RTP packets of the frame.

ToD
Indicates the time-of-day of the first sample of the frame.

Data
Points to the actual frame/field data as a BLOB.

Field
Indicates, for video frames with scanning mode Interlaced or PsF, whether this data is the first field (0) or the second field (1).

NumValidBytes
Indicates or specifies the number of valid bytes in the frame-data BLOB. The size of the frame BLOB may be larger, to accommodate frames of the same size but with varying odd and even field sizes.

Size()
Returns the frame BLOB's size in number of bytes, excluding any oversizing that was specified in the BlobMetadata allocator metadata.

Is420()
Returns whether 4:2:0 chroma subsampling is used. Relevant only when receiving SMPTE 2110 video frames.

NumRows()
Returns the total number of rows in a video frame. Relevant only when receiving SMPTE 2110 video frames.

Remarks

Memory management for frame data must be performed manually. When reading a Frame using **RxFifo**, the ownership of the frame data is transferred from the library to the user application. Once the application has processed the Frame, it should return ownership by calling **RxFifo::ReturnToMemPool**.

To transmit a Frame, the user application should first request empty frame memory by invoking **`TxFifo::GetFrameFromMemPool`** and then write the frame data into the obtained Frame. When the frame is written to **`TxFifo`**, its ownership will be transferred to the **`TxFifo`** instance.

Frame::Yuv422P_8b_GetPlanes

Obtains pointers and sizes for 8-bit 4:2:2 YUV formatted video data in a **Frame**.

This function assumes that the video format is **RxFrameFormat::Yuv422p_8b**. It calculates pointers to the Y (luminance), U- and V-plane data within the frame and determines the sizes in bytes for each respective plane.

```
YuvPlanes Frame::Yuv422P_8b_GetPlanes();
```

Parameters

None.

Return Value

The plane pointers and plane sizes are returned in a **YuvPlanes** struct.

Remarks

By calling **Frame::Yuv422P_8b_GetPlanes()**, users can easily access the YUV planes and their sizes for further processing or analysis of 8-bit 4:2:2 YUV formatted video frames.

RxFifo::Attach

Attaches an RxFifo object to a specified DekTec device and port.

```
void Attach
(
   const DtDevice& Device,    // DekTec device to attach to.
   int Port,                  // Port number (1-based) to attach to.
   HwOrSwPipe Pref=Auto       // Prefer hardware or software pipe?
);
```

Parameters

Device
Specifies the DekTec device to attach the RxFifo to. The device must be attached to the hardware, otherwise attaching the RxFifo will fail.

Port
Specifies the device's port number (1-based) to attach the RxFifo to.

Pref
Indicates the user's preference for a hardware or software pipe.

Value	Meaning
HwOrSwPipe::Auto	DTAPI chooses suitable pipe type.
HwOrSwPipe::ForceHwPipe	Force hardware pipe, throw an exception if none is available.
HwOrSwPipe::PreferHwPipe	Prefer hardware pipe, but accept a software pipe as second choice.
HwOrSwPipe::UseSwPipe	Use software pipe.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	Attaching the RxFifo to the device object will fail if the device object is not attached to the hardware.
DriverError	A fatal error occurred while accessing the DtPcie PCIe device driver or the DtaNw network driver.

Remarks

- If a preference for a hardware pipe is specified (**ForceHwPipe**), and no hardware is available, no exception will be raised. Instead, the exception will be thrown when executing **RxFifo::Start()**.

RxFifo::Clear

Empties the RxFifo and the Frame Memory Pool, releasing all memory resources associated with the RxFifo.

```
void Clear();
```

Parameters

None.

Return Value

None.

Exceptions

Exception	Generated when
`UsageError`	`RxFifo` not started with `RxFifo::Start()`.

Remarks

- The `RxFifo::Clear()` function cannot be called if the RxFifo is already started.

RxFifo::Configure

A set of overloaded functions used to configure the **RxFifo**. The appropriate function overload is called depending on the configuration parameter type provided.

```
void Configure
(
   const St2022::RxConfig& Config          // SMPTE 2022 configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
void Configure
(
   const St2110::RxConfigRaw& Config       // SMPTE 2110 raw data configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
void Configure
(
   const St2110::RxConfigAudio& Config     // SMPTE 2110 audio configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
void Configure
(
   const St2110::RxConfigVideo& Config     // SMPTE 2110 video configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
```

Parameters

Config
Represents the configuration parameters. The specific parameter type determines which function overload is called.

Metadata
Metadata that governs the alignment and number of extra bytes allocated when a new frame BLOB must be allocated.

Return Value

None.

Exceptions

Exception	Generated when
UsageError	This exception is thrown under the following conditions: • If the **RxFifo** is not currently attached to the hardware. • If the **RxFifo** is already started.

Remarks

- The **RxFifo** must be configured before starting it.
- Configuring the **RxFifo** also clears it.

RxFifo::Detach

Detaches an **RxFifo** object from the hardware. Releases all resources that were associated with the **RxFifo**.

```
void Detach();
```

Parameters

Return Value

None.

Remarks

All resources related to the **RxFifo** are released.

RxFifo::GetFifoLoad

Retrieves the current number of frames in the **RxFifo**.

If the **RxFifo** has been started, this method also checks if the receive thread has been unable to write a received Frame to the RxFifo because it had encountered its maximum capacity. If this condition was detected, an **OverflowError** is thrown.

```
int GetFifoLoad() const;
```

Parameters

Return Value

The current FIFO load, expressed as the number of frames in the **RxFifo**.

Exceptions

Exception	Meaning
OverflowError	The receive thread has received a new Frame, but the **RxFifo** has reached its maximum capacity. The reception process enters the stop state as if **RxFifo::Stop()** has been called. To recover from this error, it is recommended to clear the **RxFifo** and restart reception from scratch.

Remarks

- The receive thread is unable to throw exceptions directly to the user application. As a result, **RxFifo::GetFifoLoad()** serves as a means to communicate such exceptions to the user application.

RxFifo::GetMaxSize

Retrieves the maximum number of frames that the **RxFifo** can store.

```
int GetMaxSize() const;
```

Parameters

None.

Return Value

The maximum size of the **RxFifo**, expressed as the number of frames it can hold.

Remarks

- Frames in the **RxFifo** are dynamically allocated up to a maximum limit set with **SetMaxSize**(). The default **RxFifo** size is 4 frames.
- If a frame is received but cannot be written to the FIFO due to its maximum size being reached, the frame will be dropped and the **NumFramesFifoFull** statistic will be incremented by 1.

RxFifo::GetSharedBufferSize

Gets the current size of the shared buffer (in bytes) used to pass data from the NIC to the **RxFifo**.

```
int GetSharedBufferSize() const;
```

Parameters

None.

Return Value

The current size of the shared buffer expressed in bytes.

Remarks

- The shared buffer is an intermediate buffer used to transfer data from the NIC to the **RxFifo**. For hardware pipes, this buffer is a DMA buffer. For software pipes, it's a buffer shared between the AvFifo implementation and the driver.
- The default size of the shared buffer depends on the configured media format.
- Increasing the size of the shared buffer allows for greater task-scheduling jitter tolerance for the thread responsible for reading from the shared buffer.

RxFifo::GetStatistics

Retrieves the receive statistics.

```
RxStatistics GetStatistics() const;
```

Parameters

None.

Return Value

Returns an **RxStatistics** struct.

Remarks

- All statistics will be reset when the **Start** function is called.

RxFifo::Read

Reads a frame from the **RxFifo**. Users must verify that the **RxFifo** is not empty prior to invoking this method.

```
Frame* Read();
```

Parameters

Return Value

A pointer to a **Frame** containing the received frame.

Exceptions

Exception	Generated when
UsageError	• **RxFifo** empty.

Remarks

- Prior to reading from **RxFifo**, users should call the **RxFifo::GetFifoLoad()** method to ensure that at least one frame is available.

RxFifo::ReturnToMemPool

Returns a Frame to the Frame Memory Pool once the user has completed processing a frame that was read from the **RxFifo**. The memory pool is a pre-allocated space designed to store frames, minimizing the overhead associated with frequent memory allocation and deallocation of individual frames.

```
void ReturnToMemPool
(
   Frame* Frame                // Frame to be returned to the memory pool.
);
```

Parameters

Timeout
Pointer to the **Frame** to be returned to the memory pool.

Return Value

None.

Remarks

- Call this function after processing a frame to avoid memory leaks.
- Assumes the frame is no longer needed and can be reclaimed.

RxFifo::SetIpPars

Pre-configures the IP parameters for receiving a stream. Please note that these parameters are only checked and applied when the **RxFifo::Start()** function is invoked.

```
void SetIpPars
(
   const IpPars& Pars         // IP configuration parameters.
);
```

Parameters

Pars
IP parameters to be used for receiving the stream.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	This exception is thrown under the following conditions: • If the **RxFifo** is not currently attached to the hardware. • If the **RxFifo** is already started.

Remarks

- This function requires that the **RxFifo** is attached to a device.
- IP parameters cannot be changed if the **RxFifo** is already started.

RxFifo::SetMaxSize

Sets the maximum number of frames that the RxFifo can store.

```
void SetMaxSize
(
   int Size                    // New maximum FIFO size in number of frames.
);
```

Parameters

Size
The new maximum size of the RxFifo, expressed as the number of frames it can hold.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	The **RxFifo** size cannot be changed once it is started.

Remarks

- Frames in the RxFifo are dynamically allocated up to a maximum limit set with this function. The default RxFifo size is 4 frames.
- If a frame is received but cannot be written to the **RxFifo** due to its maximum size being reached, the frame will be dropped and the **NumFramesFifoFull** statistic will be incremented by 1.

RxFifo::SetSharedBufferSize

Sets the size of the shared buffer (in bytes) used to pass data from the NIC to the **RxFifo**. The size of the shared buffer can only be changed if the **RxFifo** is not started.

```
int SetSharedBufferSize
(
   int Size                    // New shared buffer size in bytes.
);
```

Parameters

Size
The new size of the shared buffer, expressed in bytes. The size must be a multiple of the page size (4096).

Return Value

None.

Exceptions

Exception	Meaning
UsageError	The shared buffer size cannot be changed if the **RxFifo** is already started.

Remarks

- The shared buffer is an intermediate buffer used to transfer data from the NIC to the **RxFifo**. For hardware pipes, this buffer is a DMA buffer. For software pipes, it's a buffer shared between the AvFifo implementation and the driver.
- The default size of the shared buffer depends on the media format.
- Increasing the size of the shared buffer allows for greater task-scheduling jitter tolerance for the thread responsible for reading from the shared buffer.

RxFifo::Start

Starts the receiving of frames in the **RxFifo** and resets the receive statistics. Before starting, this function performs a series of checks: verifies whether a hardware pipe is available (if one was requested), ensures the IP parameters are valid, and ascertains that the Rx FIFO is properly configured. In case of an error, it throws an exception for fatal errors trigger and returns **false** for non-fatal ones.

```
bool Start();
```

Parameters

None.

Return Value

Returns a boolean indicating the success or failure of starting the **RxFifo**.
If **false** is returned, use **GetStatus()** to identify the reason for the failure to start.

Exceptions

Exception	Meaning
DriverError	A fatal error occurred while accessing the DtPcie PCIe device driver or the DtaNw network driver. The cause may vary greatly. Refer to the **what()** string for an explanation of the failure.
HardwarePipeUnavailable	Thrown when **ForceHwPipe** was specified in **RxFifo::Attach()**, but no hardware pipe is currently available.
NetworkError	A network related error occurred, such as the network link being down or a failure to resolve the destination MAC address.
UsageError	This exception is thrown under any of the following conditions: • If the **RxFifo** is not currently attached to the hardware. • If the **RxFifo** is already started. • If the **RxFifo** is not configured with one of the **Configure** functions. • If the IP parameters have not been set.

Remarks

Ensure that **Configure()** (any of the overloads) and **SetIpPars()** have been executed before calling this function.

RxFifo::Stop

Stops receiving frames in the RxFifo.

```
void Start();
```

Parameters

None.

Return Value

None.

Remarks

RxFifo::UsesHwPipe

Indicates whether a hardware pipe (true) or software pipe (false) is used for receiving SMPTE 2110 or SMPTE 2022 frames. Hardware pipes are assigned when the **RxFifo** started, taking into account the **HwOrSwPipe** preference specified in the **RxFifo::Attach()** call and the count of already allocated hardware transmit pipes. Hence, this method can only be employed after the **RxFifo** has been started.

```
bool UsesHwPipe() const;
```

Parameters

None.

Return Value

A boolean value indicating whether a hardware pipe (true) or software pipe (false) is used.

Exceptions

Exception	Meaning
UsageError	The pipe type cannot be determined because the **RxFifo** is not started yet.

Remarks

This function is useful for identifying which type of pipe is used by the **RxFifo** and can help with optimizing performance. Hardware pipes get their own DMA controller and therefore have significantly higher performance than software pipes. However, hardware pipes are in limited supply.

TxFifo::Attach

Attaches a Transmit FIFO to a specified DekTec device and port.

```
void Attach
(
   const DtDevice& Device,    // DekTec device to attach to.
   int Port,                  // Port number (1-based) to attach to.
   HwOrSwPipe Pref=Auto       // Prefer hardware or software pipe?
);
```

Parameters

Device

Specifies the DekTec device to attach the Transmit FIFO to. The device must be attached to the hardware, otherwise attaching the Transmit FIFO will fail.

Port

Specifies the device's port number (1-based) to attach the Transmit FIFO to.

Pref

Indicates the user's preference for a hardware or software pipe.

Value	Meaning
HwOrSwPipe::Auto	DTAPI chooses a suitable pipe type.
HwOrSwPipe::HwOrSwPipe	Force hardware pipe, throw an exception if none is available.
HwOrSwPipe::PreferHwPipe	Prefer hardware pipe, but accept software pipe as a second choice.
HwOrSwPipe::UseSwPipe	Use software pipe.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	Attaching the TxFifo to the device object will fail if the device object is not attached to the hardware.
DriverError	A fatal error occurred while accessing the DtPcie PCIe device driver or the DtaNw network driver.

Remarks

TxFifo::Clear

Empties the Transmit FIFO and the Frame Memory Pool, releasing all memory resources associated with the Transmit FIFO.

```
void Clear();
```

Parameters

None.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	The FIFO cannot be cleared once the Transmit FIFO is started with **TxFifo::Start()**.

Remarks

The **TxFifo::Clear()** function cannot be called to clear the Transmit FIFO if it has already been started using **TxFifo::Start()**.

TxFifo::Configure

A set of overloaded functions used to configure the Transmit FIFO. The appropriate function overload is called depending on the configuration parameter type provided.

```
// SMPTE 2022-5/6
void Configure
(
   const St2022::TxConfigVideo& Config,    // SMPTE 2022 configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);

// SMPTE 2110
void Configure
(
   const St2110::TxConfigAudio& Config     // SMPTE 2110 audio configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
void Configure
(
   const St2110::TxConfigRaw& Config       // Any SMPTE 2110 substandard.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
void Configure
(
   const St2110::TxConfigRawVideo& Config  // SMPTE 2110 raw video configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
void Configure
(
   const St2110::TxConfigVideo& Config     // SMPTE 2110 video configuration.
   BlobMetadata& Metadata                  // Frame BLOB alignment metadata.
);
```

Parameters

Config
Represents the configuration parameters. The specific parameter type determines which function overload is called.

Metadata
Metadata that governs the alignment and number of extra bytes allocated when a new frame BLOB must be allocated.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	Configuring the Transmit FIFO requires that the TxFifo object is attached to the hardware and not started yet.

Remarks

- The Transmit FIFO must be configured before starting it with the **TxFifo::Start()** method.
- Configuring the Transmit FIFO also clears it.

TxFifo::Detach

Detaches a Transmit FIFO from the hardware. Releases all related resources, including the Frame Memory Pool associated with the TxFifo object.

```
void Detach();
```

Parameters

Return Value

None.

Remarks

All resources related to the Transmit FIFO are released.

TxFifo::GetFifoLoad

Retrieves the current number of frames in the **TxFifo**.

If the **TxFifo** has been started, this method also checks if the packet scheduler in the transmit thread has detected any invalid transmit timestamps. If an invalid timestamp is detected, a **SchedulingError** is thrown.

```
int GetFifoLoad() const;
```

Parameters

Return Value

The current FIFO load, expressed as the number of frames in the **TxFifo**.

Exceptions

Exception	Meaning
SchedulingError	The scheduler in the transmit thread has encountered an invalid timestamp that is too far in the future or the past, causing the transmission to enter the stop state as if **TxFifo::Stop()** has been called. To recover from this error, it is recommended to clear the **TxFifo** and restart the transmission loop from scratch.

Remarks

- The transmit thread is unable to throw exceptions directly to the user application. As a result, **TxFifo::GetFifoLoad()** serves as a means to communicate such exceptions to the user application.

TxFifo::GetFrameFromMemPool

Retrieves a Frame of the requested size from the memory pool, with the intention of filling the Frame with data for transmission and writing the Frame to the Transmit FIFO.

```
Frame* GetFrameFromMemPool
(
   size_t Size                // Requested size of the embedded frame BLOB.
);
```

Parameters

Return Value

A pointer to a **Frame** in the Frame Memory Pool.

Exceptions

Exception	Meaning
std::bad_alloc	Insufficient memory to allocate a frame.

Remarks

- To write a frame, the application must first obtain a Frame (struct Frame with embedded memory for storing frame data) from the memory pool.
- The memory pool manages available Frames. If a Frame is available and can be recycled, its size is checked to ensure it can accommodate the requested frame size. If the available frame BLOB in the frame is too small, it is resized; if too large, it remains unchanged.
- If no frames are available in the memory pool, a new frame is dynamically allocated.
- After the application writes the frame to the TxFifo, DTAPI returns the Frame to the memory pool once the data is transmitted.

TxFifo::GetMaxSize

Retrieves the maximum number of frames that the Transmit FIFO can store.

```
int GetMaxSize() const;
```

Parameters

None.

Return Value

The maximum size of the Transmit FIFO, expressed as the number of frames it can hold.

Remarks

TxFifo::GetSharedBufferSize

Gets the current size of the shared buffer (in bytes) used to pass data from the **TxFifo** to the NIC.

```
int GetSharedBufferSize() const;
```

Parameters

None.

Return Value

The current size of the shared buffer, expressed in bytes.

Remarks

- The shared buffer is an intermediate buffer used to transfer data from the **TxFifo** to the NIC. For hardware pipes, this buffer is a DMA buffer. For software pipes, it's a buffer shared between the AvFifo implementation and the driver.
- The default size of the shared buffer depends on the configured media format.
- Increasing the size of the shared buffer allows for greater task-scheduling jitter tolerance for the thread responsible for writing to the shared buffer.

TxFifo::GetStatistics

Retrieves the receive statistics.

```
TxStatistics GetStatistics() const;
```

Parameters

None.

Return Value

Transmit statistics as a structure containing several counter values:

Remarks

- All statistics will be reset when the **Start** function is called.

TxFifo::GetStatus

Retrieves the status of the **TxFifo's** IP connection.

```
FifoStatus GetStatus() const;
```

Parameters

None.

Return Value

Returns a **FifoStatus** enumeration.

Remarks

TxFifo::SetIpPars

Configures the IP parameters to be used for transmitting a stream.

```
void SetIpPars
(
   const IpPars& Pars         // IP configuration parameters.
);
```

Parameters

Pars
IP parameters to be used for transmitting the stream.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	The IP parameters cannot be set if: - the Transmit FIFO is not attached to the hardware; - the Transmit FIFO is already started.
std::invalid_argument	One of the IpPars fields is outside its valid range. The what() string will indicate which field value is invalid.

Remarks

- This function requires that the Transmit FIFO is attached to a device.
- IP parameters cannot be changed if the Transmit FIFO is already started.

TxFifo::SetMaxSize

Sets the maximum number of frames that the Transmit FIFO can store.

```
void SetMaxSize
(
   int Size                    // New maximum FIFO size in number of frames.
);
```

Parameters

Size
The new maximum size of the Transmit FIFO, expressed as the number of frames it can hold.

Return Value

None.

Exceptions

Exception	Meaning
UsageError	The Transmit FIFO size cannot be changed once it is started.

Remarks

TxFifo::SetSharedBufferSize

Sets the size of the shared buffer (in bytes) used to pass data from the **TxFifo** to the NIC. The size of the shared buffer can only be changed if the **TxFifo** is not started.

```
void SetSharedBufferSize
(
   int Size                    // New shared buffer size in bytes.
);
```

Parameters

Size
The new size of the shared buffer, expressed in bytes. The size must be a multiple of the page size (4096).

Return Value

None.

Exceptions

Exception	Meaning
UsageError	The shared buffer size cannot be changed if the **TxFifo** is already started.

Remarks

- The shared buffer is an intermediate buffer used to transfer data from the **TxFifo** to the NIC. For hardware pipes, this buffer is a DMA buffer. For software pipes, it's a buffer shared between the AvFifo implementation and the driver.
- The default size of the shared buffer depends on the media format.
- Increasing the size of the shared buffer allows for greater task-scheduling jitter tolerance for the thread responsible for writing to the shared buffer.

TxFifo::Start

Starts transmitting frames in the Transmit FIFO.

```
void Start();
```

Parameters

None.

Return Value

None.

Exceptions

Exception	Meaning
DriverError	A fatal error occurred while accessing the DtPcie PCIe device driver or the DtaNw network driver.
NetworkError	A network related error occurred, e.g. the network link is down or resolving the destination MAC address failed.
UsageError	Frame transmission cannot be started if: - the Transmit FIFO is not attached to the hardware; - the Transmit FIFO is not configured; - IP parameters have not been configured.

Remarks

TxFifo::Stop

Stops transmitting frames.

```
void Stop();
```

Parameters

None.

Return Value

None.

Remarks

TxFifo::UsesHwPipe

Indicates whether a hardware pipe (true) or software pipe (false) is used for transmitting SMPTE 2110 or SMPTE 2022 frames. Hardware pipes are assigned when the **TxFifo** started, taking into account the **HwOrSwPipe** preference specified in the **TxFifo::Attach()** call and the count of already allocated hardware transmit pipes. Hence, this method can only be employed after the TxFifo has been started.

```
bool UsesHwPipe() const;
```

Parameters

None.

Return Value

A boolean value indicating whether a hardware pipe (true) or software pipe (false) is used.

Exceptions

Exception	Meaning
UsageError	The pipe type cannot be determined because the **TxFifo** is not started yet.

Remarks

This function is useful for identifying which type of pipe is used by the **TxFifo** and can help with optimizing performance. Hardware pipes get their own DMA controller and therefore have significantly higher performance than software pipes. However, hardware pipes are in limited supply.

TxFifo::Write

Writes a frame to the Transmit FIFO.

```
void Write
(
   Frame* Frame                // Unit of audio/video/ancillary data.
);
```

Parameters

Frame
Frame to be stored in the Transmit FIFO for transmission.

Return Value

None.

Remarks

The user must obtain a Frame by calling **TxFifo::GetFrameFromMemPool()**.

FifoStatusToMessage

Converts a **FifoStatus** enumeration value to a readable string message.

```
string FifoStatusToMessage
(
   FifoStatus Status          // Status code to be converted to a message.
)
```

Parameters

Status
The **FifoStatus** enumeration value to be converted into a message.

Return Value

string
A brief message describing the corresponding link status.

Remarks

- The **FifoStatusToMessage** function provides a mechanism to translate the **FifoStatus** enumeration values into human-readable messages, aiding in the debugging and understanding of the link status. Always check the returned message to understand the status of the IP link in your network operation.

St2022::Tod2Rtp

Converts a time-of-day value to the corresponding RTP timestamp for a SMPTE 2022-5/6 (SDI over IP) stream.

```
uint32_t Tod2Rtp
(
   DtTimeOfDay ToD            // Time-of-day value to be converted.
)
```

Parameters

ToD
Time-of-day value to be converted to an RTP timestamp.

Return Value

uint32_t
The RTP timestamp corresponding to the time-of-day value.

Remark

St2110::Rtp2Tod_Audio

Converts an RTP timestamp to a time-of-day value for audio. The calculated time-of-day will align with the audio media grid for the specified sample rate, provided that the given RTP timestamp is also grid-aligned.

```
DtTimeOfDay Rtp2Tod_Audio
(
   uint32_t RtpTime,          // Timestamp in RTP header.
   DtTimeOfDay ApproxToD,     // Approximate time of day.
   int SampleRate             // Audio sample rate in Hz.
)
```

Parameters

RtpTime
Timestamp retrieved from the RTP header, in 90kHz units.

ApproxToD
Specifies the approximate time of day, which helps the function determine the time window containing the RTP timestamp. The approximate time of day does not need to be accurate, even a deviation of plus or minus one day is sufficient.

SampleRate
Specifies the audio sample rate expressed in Hertz (Hz).

Return Value

DtTimeOfDay
The time-of-day value corresponding to the RTP timestamp.

Remarks

Typically, the current time of day can be used for *ApproxTod*.

St2110::Rtp2Tod_Video

Converts an RTP timestamp to a time-of-day value for video. The calculated time-of-day will align with the video media grid, provided that the given RTP timestamp is also grid-aligned.

```
DtTimeOfDay Rtp2Tod_Video
(
   uint32_t RtpTime,          // Timestamp in RTP header.
   DtTimeOfDay ApproxToD      // Approximate time of day.
)
```

Parameters

RtpTime
Timestamp retrieved from the RTP header, in 90kHz units.

ApproxToD
Specifies the approximate time of day, which helps the function determine the time window containing the RTP timestamp. The approximate time of day does not need to be accurate, even a deviation of plus or minus one day is sufficient.

Return Value

DtTimeOfDay
The time-of-day value corresponding to the RTP timestamp.

Remarks

Typically, the current time of day can be used for *ApproxTod*.

St2110::Tod2Rtp_Audio

Converts a time-of-day value to the corresponding RTP timestamp for audio. If the specified time-of-day value is aligned to the audio media grid for the given sample rate, then the calculated RTP timestamp is guaranteed to be aligned as well.

```
uint32_t Tod2Rtp_Audio
(
   DtTimeOfDay ToD            // Time-of-day value to be converted.
   int SampleRate             // Audio sample rate in Hz.
)
```

Parameters

ToD
Time-of-day value to be converted to an RTP timestamp.

SampleRate
Specifies the audio sample rate expressed in Hertz (Hz).

Return Value

uint32_t
The RTP timestamp corresponding to the time-of-day value.

Remarks

If the input time-of-day value is not grid-aligned, the function will make a reasonable rounding attempt to determine the closest RTP timestamp, taking into account complexities arising from fractional video frame rates.

St2110::Tod2Rtp_Video

Converts a time-of-day value to the corresponding RTP timestamp for video. If the specified time-of-day value is aligned to the video media grid, then the calculated RTP timestamp is guaranteed to be aligned as well.

```
uint32_t Tod2Rtp_Video
(
   DtTimeOfDay ToD            // Time-of-day value to be converted.
)
```

Parameters

ToD
Time-of-day value to be converted to an RTP timestamp.

Return Value

uint32_t
The RTP timestamp corresponding to the time-of-day value.

Remarks

If the input time-of-day value is not grid-aligned, the function will make a reasonable rounding attempt to determine the closest RTP timestamp, taking into account complexities arising from fractional video frame rates.

Tod2Grid_Audio

Aligns a specified time-of-day value with the media grid for audio.

```
DtTimeOfDay Tod2Grid_Audio
(
   DtTimeOfDay ToD,           // Time of day to be aligned.
   int SampleRate             // Audio sample rate in Hz.
)
```

Parameters

ToD
Time of day to be aligned to the audio media clock grid.

SampleRate
Specifies the audio sample rate expressed in Hertz (Hz).

Return Value

DtTimeOfDay
The nearest media-grid aligned time-of-day value corresponding to the input parameters.

Remarks

Tod2Grid_Video

Aligns a specified time-of-day value with the media grid for video.

```
DtTimeOfDay Tod2Grid_Video
(
   DtTimeOfDay ToD,           // Time of day to be aligned.
   FrameRate Rate;            // Frame rate; field rate for Interlaced/PsF.
)
```

Parameters

ToD
Time of day to be aligned to the audio media clock grid.

Rate
Specifies the frame rate for progressive scanning modes, or the field rate for interlaced or PsF scanning mode.

Return Value

DtTimeOfDay
The nearest media-clock grid aligned time-of-day value corresponding to the input parameters.

Remarks

In SMPTE 2110, timestamps are calculated relative to the starting point of the PTP timescale, denoted as T0. This function computes the number of frames that have occurred since T0 for a given time-of-day value (also referred to as PTP time), considering the specified frame rate. The function then returns the time-of-day corresponding to the nearest start-of-frame time point.

Providing an exact frame rate is crucial for accurate computation, which is why the frame rate is specified as an exact rational number.