DTAPI

Overview and Data Formats

REFERENCE

Nov 2017

Table of Contents

1. General Description

1.1. What is DTAPI?

DTAPI is an acronym for DekTec Application Programming Interface, an API for controlling DekTec PC add-on hardware (PCIe cards and USB devices), and reading and writing data to it. DTAPI is part of the DekTec SDK, which also contains device drivers, documentation, example code, etc.

DTAPI enables application programs to access the functions of DekTec devices at a higher level of abstraction than would be possible using direct device-driver calls. Nonetheless, it allows efficient access to nearly all hardware features.

From a technical point of view, DTAPI is a C++ library with an object oriented interface that links to a user application. The DTAPI library uses three device drivers (`Dta`, `Dtu`, `DtaNw`,) for accessing the hardware: `Dta` handles PCI and PCI express cards, `Dtu` handles USB-2 and USB-3 devices and `DtaNw` is the network driver for IP-enabled devices. An auxiliary service (on Windows) or daemon (on Linux) is running to provide services that should run continuously or that span multiple applications. Collectively, DTAPI, the device drivers, the DTAPI service and the documentation are called the DekTec SDK. It's available as "Windows SDK" for Windows XP onwards and as "Linux SDK" for Linux 2.6 onwards.

From a programmer's point of view, DTAPI is composed of a header file (`DTAPI.h`), to be included in the application's source code, and a library file, to be linked to the application's executable. DTAPI is also available as .NET assembly.

1.2. Documentation Overview

The table below shows the documents describing DTAPI.

Document	Description
DTAPI Reference – Overview and Data Formats	This document. Overview of DTAPI and definition of data formats.
DTAPI Reference – Core Classes	Reference for the core classes and methods in DTAPI, mainly the device and channel classes.
DTAPI Reference – Advanced Demodulator API	Reference for the advanced demodulator classes and structures in DTAPI.
DTAPI Reference – DekTec Matrix API	Reference for real-time processing of uncompressed audio and video with the DekTec Matrix API (part of DTAPI).
DTAPI Reference – Encoder Control	Reference of the DTAPI classes for controlling audio- and video encoding hardware.
DTAPI Reference – Multi-PLP Extensions	Reference for the multi-PLP ATSC 3.0, DVB-C2, DVB-T2 and ISDB-Tmm modulator classes in DTAPI.
DekTec SDK – Revision History	List of changes for each release of the Windows/Linux SDK since the May2012 SDK release.

1.3. DTAPI Object Model

DTAPI consists of a collection of C++ classes. Some classes represent hardware functions, others represent control parameters. The hardware is controlled and managed by invoking methods on DTAPI objects. The core classes of DTAPI are `DtDevice`, `DtInpChannel` and `DtOutpChannel`.

A DekTec device is represented by a **`DtDevice`** object. An application that wants to interact with a device first 'attaches' a **`DtDevice`** object to the hardware. To build an inventory of DekTec devices in the system, the **`DtDevice`** class is supplemented by a global function **`DtapiDeviceScan`**.

Figure 1 illustrates DTAPI in action. The application interacts with DTAPI objects, which in turn communicate with the hardware through a device driver.

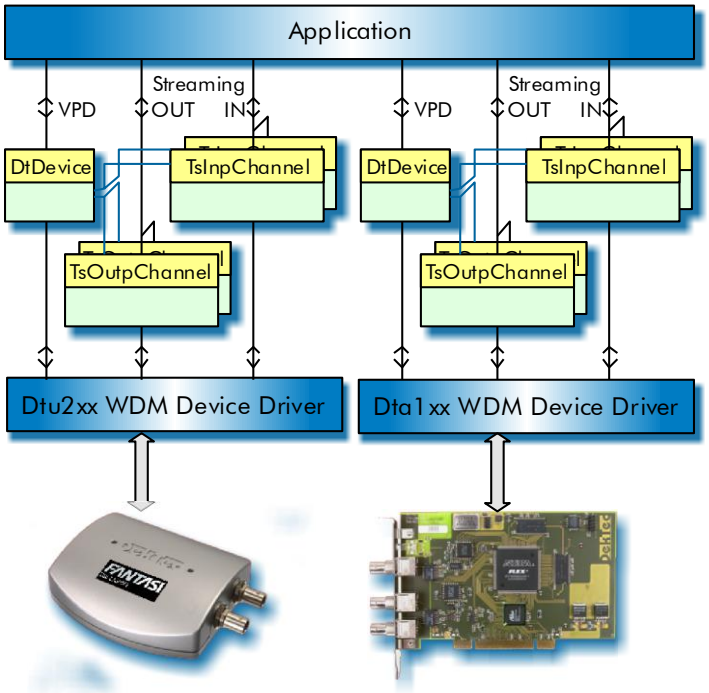

Figure 1. Example of DTAPI objects representing two devices.

The I/O ports on a device are represented by *channel* objects. Two channel classes are defined: **`DtInpChannel`** for representing an input port and **`DtOutpChannel`** for an output port. A network (IP) port is a special case: a channel object is instantiated for each logical stream. An application attaches a channel object to an I/O port by specifying a **`DtDevice`** object and a port number. The core methods of the channel classes are **`DtInpChannel::Read`** for reading data from an input port and **`DtOutpChannel::Write`** for streaming data to an output port.

1.4. List of Abbreviations and Glossary of Terms

bit string – Sequence of bits. Bit strings are written as a string of 1s and 0s within single quote marks, e.g. '1000 0001'. Blanks within a bit string are for ease of reading and have no significance.

bslbf – Bit string, left bit first. Used in bit stream definitions. "Left" refers to the order in which bit strings are written in this document. "First" refers to the first bit transmitted or received. For example, in '1000' the first bit transmitted or received is a '1'.

bsrlb – Bit string right-to-left in bytes.

channel object – Instance of a C++ class that represents a physical input or output stream. A user application streams data in or out of an I/O port by invoking methods on the channel object.

device object – Instance of a C++ class that represents a DekTec device.

DTA-xxx card – Any DekTec PCI or PCI Express card in the DTA series.

Dta – Name of the device driver for DekTec PCI or PCI Express cards. This device driver is generic: a single device driver is used for all PCI devices (instead of using one device driver for each device type).

DTAPI – DekTec Application Programming Interface.

Dtu – Name of the device driver for DekTec USB devices. This device driver is generic: a single device driver is used for all USB devices (instead of using one device driver for each device type).

uimsbf – Unsigned integer, most significant bit first.

VPD – Vital Product Data. Information stored in a PCI device to uniquely identify the hardware and, potentially, software elements of the device. DekTec devices store VPD in on-board serial EEPROMs. DTAPI supports methods to read and write VPD items.

1.5. References

- *ISO/IEC 13818-1, Information technology – Generic coding of moving pictures and associated audio information: Systems*, also known as "MPEG-2 Systems" – Specification of the structure of a MPEG-2 Transport Stream.
- *Recommendation ITU-R BT.656-4. Interfaces for digital component video signals in 525-line and 625-line television systems operating at the 4:2:2 level of recommendation ITU-R BT.601 (Part A).*
- *ETSI EN 302 769, Digital Video Broadcasting (DVB); Frame structure channel coding and modulation for a second generation transmission system for cable systems (DVB-C2).*
- *ETSI EN 302 755, Digital Video Broadcasting (DVB); Frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2).*
- *ETSI EN 102 773, Digital Video Broadcasting (DVB); Modulator Interface (T2-MI) for a second generation digital terrestrial television broadcasting system (DVB-T2).*

2. Using DTAPI in your Project

This section describes how to use DTAPI on Windows (§2.1) and on Linux (§2.4).

2.1. DTAPI on the Windows Platform

DTAPI for Windows is available as a static link library and as .NET 4.0 assembly. All DTAPI declarations and definitions are contained in a single C++ header file: **DTAPI.h**. Each module that uses DTAPI functionality has to include this file.

2.2. Using the Static Link Library

The static link libraries are available for VC8 (Visual Studio 2005), VC9 (Visual Studio 2008), VC10 (Visual Studio 2010), VC11 (Visual Studio 2012), VC12 (Visual Studio 2013) and VC14 (Visual Studio 2015). For each compiler platform, eight versions of the library are available.

Library File	#bits	Run-Time Library	Configuration
DTAPIMD.lib	32	multi-threaded DLL (/MD)	release
DTAPIMDd.lib	32	multi-threaded DLL (/MD)	debug
DTAPIMT.lib	32	multi-threaded (/MT)	release
DTAPIMTd.lib	32	multi-threaded (/MT)	debug
DTAPI64MD.lib	64	multi-threaded DLL (/MD)	release
DTAPI64MDd.lib	64	multi-threaded DLL (/MD)	debug
DTAPI64MT.lib	64	multi-threaded (/MT)	release
DTAPI64MTd.lib	64	multi-threaded (/MT)	debug

The correct version of the DTAPI library is automatically linked to the application. This is accomplished with pragma directives in **DTAPI.h**, e.g. "**#pragma comment(lib, "DTAPI64MDd.lib")**", embedded in **#ifdef** statements.

Automatic linking can be disabled by defining **_DTAPI_DISABLE_AUTO_LINK** in your source code with a **#define** before including **DTAPI.h**. Alternatively, you can define this constant in the Configuration Properties in the C++, Preprocessor Definitions section.

So, to use the static link library of the DTAPI follow these steps:

1. Copy **DTAPI.h** and the right version(s) of **DTAPIxxx.lib** to your project or to a standard location visible to VC++.
2. Add "**#include "DTAPI.h**" to each file that uses DTAPI constants and/or functions.
3. Compile your application using compiler settings that match those of the lib file.

Instead of the manual copy it also possible to use a search path to look for the **DTAPI.h** and **DTAPIXX.lib** files in the WinSDK installation directory, which is typically:

C:\Program Files\DekTec\SDKs\WinSDK\DTAPI

For Visual Studio 2010 (VC10) and later version the WinSDK installer adds two convenience macros[1]:

- $(DtapiIncludePath), pointing to **<installdir>/DTAPI/Include**
- $(DtapiLibraryPath), pointing to **<installdir>/DTAPI/Lib**

[1] For a multi user PC development environment each user should initially do an installation of the WinSDK to make sure that the convenience macros are installed for each user.

You can use these convenience macros to update the search path in your project settings:

- Add $(DtapiIncludePath) to the "Additional Include Directories" in the "C/C++ General" settings section.
- Add $(DtapiLibraryPath)\VC10 to the "Additional Library Directories" in the "Linker General" settings section.

NOTE: add \VC10 to the end of $(DtapiLibraryPath) for VS.2010 projects, \VC11 for VS.2012 projects, etc., to link with the correct version of the DTAPI library.

For earlier versions of Visual Studio, e.g. VS.2008, there are no such convenience macros and you should manually add the DTAPI include and lib paths to the global Visual Studio include and library search paths.

2.3. Using the .NET Assembly

`DTAPINET.dll` and **`DTAPINET64.dll`** are .NET 4.0 compatible assemblies of DTAPI. To use it you should perform the following steps:

1. Make sure the .NET 4.0 SDK has been installed on your system.
2. Copy **`DTAPINET.dll`** to your project or to a standard location visible to VC# (or other .NET IDE).
3. Add a reference to the **`DTAPINET.dll`** assembly to your project.
4. Add a "**`#using DTAPINET`**" statement to the beginning of each source file that uses the classes, methods, and or constants exported by the **`DTAPINET`** assembly.

2.4. DTAPI on the Linux Platform

Using DTAPI in a Linux application is straightforward:

1. Make sure that **`DTAPI.h`** and **`DTAPI.o`** are located in a path reachable from your project.
2. Add "**`#include DTAPI.h`**" to each file using DTAPI.
3. Link the **`DTAPI.o`** library file to your application.
4. DTAPI requires the **`pthread`** library, so link this library to your application too.

The DTAPI library file is available for different GCC versions. Please refer to the …**`/LinuxSDK/DTAPI/Bin/`** directory.

3. DTAPI Basics

3.1. Attaching to a Device

Programs that use DTAPI first have to instantiate a **DtDevice** object and "attach" it to a hardware device. This can be accomplished in several ways.

DtDevice::AttachToType is convenient when the DekTec device type number is known and the system contains a single adapter of the given type.

```
DtDevice Dvc;
if (Dvc.AttachToType(2145) != DTAPI_OK)
  // No DTA-2145 in the system ...
```

Figure 2. Attaching a **DtDevice** object to the hardware based on type number.

DtDevice::AttachToSerial can be used if the serial number of the device is known.

```
DtDevice Dvc;
if (Dvc.AttachToSerial(2145000123) != DTAPI_OK)
  // No card with serial# 2145000123
```

Figure 3. Attaching a **DtDevice** object to the hardware based on serial number.

DtDevice::AttachToSlot can be used if the physical location of a PCI or PCI Express card in the system is known.

```
DtDevice Dvc;
if (Dvc.AttachToSlot(1, 3) != DTAPI_OK)
  // No card in slot 3 on PCI bus 1 ...
```

Figure 4. Attaching a **DtDevice** object to the hardware based on PCI bus and slot number.

For DTEs (e.g. DTE-3100) in DTAPI mode, **DtDevice::AttachToIpAddr** can be used:

```
DtDevice Dvc;
unsigned char IpAddr[4] = { 192, 168, 23, 114 };
if (Dvc.AttachToIpAddr(IpAddr) != DTAPI_OK)
  // No DTE found at 192.168.23.114
```

Figure 5. Attaching a **DtDevice** object to the hardware based on IP address.

A more sophisticated application creates an inventory of DekTec devices, with global function **DtapiHwFuncScan** or **DtapiDeviceScan**, and lets the user configure which device is to be used.

```
DtHwFuncDesc HwFuncs[10];
int f, NumberOfHwFuncs;
::DtapiHwFuncScan(10, NumberOfHwFuncs, HwFuncs);

for (f=0; f<NumberOfHwFuncs; f++)
  if (HwFuncs[f].m_ChanType & DTAPI_CHAN_OUTPUT)
    break;

if (f == NumberOfHwFuncs) { // No output card }

DtDevice Dvc;
Dvc.AttachToSerial(HwFuncs[f].m_DvcDesc.m_Serial);
```

Figure 6. Attaching to the first device with an output port.

After all operations have been completed, the **DtDevice** object may be detached from the hardware with method **Detach**.

3.2. Attaching to a Channel

Before you can stream data into or out of a DekTec device, two objects must have been instantiated and attached to the hardware:

- A **DtDevice** object (§3.1);
- A channel object: **DtInpChannel** for streaming data from an input port into your application, or **DtOutpChannel** for streaming data to an output port.

The channel object is attached to the hardware with the channel's **AttachToPort** member function. The first parameter of this function is a pointer to the **DtDevice** object that hosts the channel. The second parameter identifies the port number.

```
DtDevice Dvc;
// Code to attach to the device hardware goes here

DtOutpChannel Outp;
if (Outp.AttachToPort(&Dvc, 1) != DTAPI_OK)
  // Error-handling code

DtInpChannel Inp;
if (Inp.AttachToPort(&Dvc, 2) != DTAPI_OK)
  // Error-handling code
```

Figure 7. Attaching a **DtOutpChannel** and a **DtInpChannel** object to the hardware.

Just like device objects, **DtOutpChannel** and **DtInpChannel** objects should be detached from the hardware after all operations on the channel have been completed.

3.3. Initialising a Channel

After attaching to the hardware, and before streaming can commence, the channel must be initialized.

Port type	Channel object	Initialization
DVB-ASI input	`DtInpChannel`	**`SetRxMode`** sets the packet size of packets stored in the receive FIFO.
DVB-ASI output	`DtOutpChannel`	**`SetTsRateBps`** sets the output bit rate. **`SetTxMode`** sets the packet size and burst- or continuous mode.
IP input	`DtInpChannel`	**`SetIpPars`** sets the IP reception parameters, primarily the IP source address. **`SetRxMode`** sets the packet size of packets stored in the receive FIFO.
IP output	`DtOutpChannel`	**`SetIpPars`** sets the IP transmission parameters, primarily the IP destination address. **`SetTsRateBps`** sets the output bit rate. **`SetTxMode`** sets the packet size and burst- or continuous mode.

3.4. Receiving Data

This section considers the actual reception of data (usually a Transport Stream) from an external source to your application. The core of an elementary reception program is shown in Figure 8. This code assumes the following:

- Device object **`Dvc`** and channel object **`Inp`** have been attached to the hardware;
- The receive FIFO is empty and receive mode has been initialised;
- **`ProcessData(DataBuffer, NumBytes)`** is the function that processes the data;
- **`StopCondition()`** is a user-supplied function to break out of the reception loop.

```
// PRE-CONDITION: Dvc and Inp have been attached to the hardware
char DataBuffer[BUFSIZE];

// Signal the hardware to start receiving data into the receive FIFO
Inp.SetRxControl(DTAPI_RXCTRL_RCV);

// Main loop
while (!StopCondition())
{
  Inp.Read(DataBuffer, BUFSIZE);
  ProcessData(DataBuffer, BUFSIZE);
}
```

Figure 8. Minimal program for receiving data from an external data source.

The code is straightforward. First receive mode is set to **`Rcv`**, which instructs the hardware to start storing data in the receive FIFO. In the main loop, **`Inp.Read`** sleeps until **`BUFSIZE`** bytes are received. The main loop alternates between reading data and processing the data, until the stop condition becomes true.

The following factors should be considered to achieve optimal results:

- The buffer size (constant **BUFSIZE**) should not be chosen too small. Every data transfer from the receive FIFO to the buffer in host memory incurs non-negligible overhead for setting up a DMA transfer.
  A reasonable minimum buffer/transfer size is 4096 bytes. No maximum size exists; the buffer size may very well be a few Mbytes.
- The number of bytes returned by method **Read** always is a multiple of 4. It is not guaranteed that the data aligns to Transport-Packet boundaries, even if the buffer size is a multiple of the packet size. The processing software should always start with a synchronisation stage.
- If using SDI, the ReadFrame function can be used instead of the Read function to read the complete SDI frame at ones. The **BUFSIZE** must be the size of a complete SDI frame.

3.5. Transmitting Data

Transmitting data to an output is somewhat more involved than receiving data. The core of a minimal program that transmits data is shown in Figure 9. The code assumes the following:

- Device object **Dvc** and channel object **Outp** have been attached to the hardware;
- The transmission parameters have been initialised;
- **GetData(DataBuffer, NumBytes)** is the function that generates data bytes to be transmitted.

The first part of the code builds an initial load in the transmit FIFO before actual transmission begins. Hereto transmission control is set to **HOLD**, which enables DMA to the transmit FIFO on the device but keeps transmission disabled.

```
// PRE-CONDITION: Dvc and Outp have been attached to the hardware
//                Transmission parameters have been initialized

// Build initial load in transmit FIFO
Outp.SetTxControl(DTAPI_TXCTRL_HOLD);         // Start in HOLD mode
char DataBuffer[BUFSIZE];
for (int Load=0; Load<INITIAL_LOAD; Load+=BUFSIZE)
{
  GetData(DataBuffer, BUFSIZE);
  Outp.Write(DataBuffer, BUFSIZE);
}

// Go to SEND mode: this starts the transmission of data
Outp.SetTxControl(DTAPI_TXCTRL_SEND);

// Main loop
while (!StopCondition())
{
  Outp.Write(DataBuffer, BUFSIZE);
  GetData(DataBuffer, BUFSIZE);
}
```

Figure 9. Minimal program to transmit data.

When the transmit FIFO contains its initial load, actual transmission is started by setting transmission control to **SEND**. The main loop then supplies additional data to the transmit FIFO.

The following factors should be considered to achieve optimal results:

- The buffer size (constant **BUFSIZE**) should not be chosen too small. Every data transfer to the transmit FIFO incurs overhead for setting up a DMA transfer.

- The initial load written to the transmit FIFO (**`INITIAL_LOAD`**) should not be too small either, to prevent an early underflow of the transmit FIFO in the main loop. A value close to the maximum FIFO size is recommended.
  The initial load cannot be larger than the size of the transmit FIFO: this would cause an application "stall", because **`Outp.Write`** will sleep forever.

3.6. Example Code for a Simple Stream Player

Figure 10 shows the code of a simple but fully functional command-line stream player that is capable of transmitting a TS file to DVB-ASI output port #1 of a DTA-2145. The filename and bit rate at which to play out the file can be specified as command-line arguments.

The example exploits good-old "stdio" functions for reading file data. By using a relatively large buffer, performance is more than adequate.

Obviously, this example is just a first step towards a production-quality streamer application. With respect to DTAPI, one obvious improvement would be to check the return code for every DTAPI-call, and add corresponding error-handling code.

```
// Command-line program TsOut to transmit a TS file out of a DTA-2145

#define BUFSIZE 0x10000            // 64kB buffer size
#define INITIAL_LOAD (7*1024*1024) // 7MB initial load

#include "DTAPI.h"
#include <stdio.h>

int main(int argc, char* argv[])
{
  if (argc != 3) {
    printf("Usage: TsOut bitrate tsfile\nQuitting...\n");
    return -1;
  }
  FILE* fp = fopen(argv[2], "rb");
  if (fp == NULL) {
    printf("Can't open '%s' for read\nQuitting...\n", argv[2]);
    return -2;
  }
  // Attach device and output channel objects to hardware
  DtDevice Dvc;
  if (Dvc.AttachToType(2145) != DTAPI_OK) {
    printf("No DTA-2145 in system. Quitting...\n");
    return -3;
  }
  DtOutpChannel TsOut;
  if (TsOut.AttachToPort(&Dvc, 1) != DTAPI_OK) {
    printf("Can't attach output channel.\nQuitting...\n");
    return -4;
  }
  // Initialise bit rate and packet mode
  TsOut.SetTsRateBps(atoi(argv[1]));
  TsOut.SetTxMode(DTAPI_TXMODE_188, DTAPI_TXSTUFF_MODE_ON);

  // Build initial load in Transmit FIFO
  TsOut.SetTxControl(DTAPI_TXCTRL_HOLD);
  char Buf[BUFSIZE];
  int Load = 0;
  int NumBytes = fread(Buf, 1, BUFSIZE, fp);
  while (Load<INITIAL_LOAD && NumBytes!=0) {
    TsOut.Write(Buf, NumBytes);
    Load += NumBytes;
    NumBytes = fread(Buf, 1, BUFSIZE, fp);
  }

  // Start transmission
  TsOut.SetTxControl(DTAPI_TXCTRL_SEND);

  // Main loop
  while (NumBytes != 0) {
    TsOut.Write(Buf, NumBytes);
    NumBytes = fread(Buf, 1, BUFSIZE, fp);
  }
  return 0;
}
```

Figure 10. Complete command-line application to stream a file with the DTA-2145.

4. Capabilities and I/O Configuration

DTAPI supports mechanisms to discover the capabilities of DekTec I/O adapters programmatically, and configure the hardware dynamically.

4.1. Introduction

A DTAPI "**capability**" is a constant that identifies a characteristic or feature of a physical port. For example, **DTAP_CAP_ASI** indicates that a ports supports ASI (it doesn't say whether ASI reception and/or ASI transmission are supported, other capabilities are used for that).

The global DTAPI function **::DtapiHwFuncScan** scans the hardware and creates a hardware function descriptor (**DtHwFuncDesc**) for each port. Capabilities are encoded in member *m_Flags* of data type: **DtCaps**. Capabilities can be OR-ed together.

To test for a certain capability:

```
if ((HwFuncDesc.m_Flags & DTAPI_CAP_ASI) != 0)
   // Port supports ASI
```

I/O configuration is the process to dynamically configure I/O ports from software. You can use the **SetIoConfig** method to set the I/O configuration of a port, and **GetIoConfig** to read it back.

4.2. Capabilities

Capabilities are organized in *groups*, *capabilities* and *sub-capabilities*.

Capability Group	A set of capabilities applying to comparable characteristics. For example, capabilities in group **IOSTD** all apply to the I/O standards supported by a physical port.
Capability	A constant that identifies a characteristic of a port. For example **DTAPI_CAP_HDSDI**, a member of group **IOSTD**, indicates that the port supports HD-SDI.
Sub-Capability	A constant that identifies a sub-characteristic of a port. For example, sub-capabilities of **DTAPI_CAP_HDSDI** include **DTAPI_CAP_1080I50**, **DTAPI_CAP_1080I59_94**, etc.

The DekTec SDK contains the following documentation on capabilities.

CapList.xlsx	A spreadsheet that lists the capabilities, sub-capabilities and attributes supported by each DekTec I/O adapter.
DTAPI.h	This header file contains a complete list of all available capabilities in the form of **DTAPI_CAP_XXX** defines.

There are two main categories of capabilities: I/O capabilities and standard capabilities.

I/O Capability	Capability that is linked to I/O configuration: If an I/O capability is supported, `SetIoConfig` can be used to enable the port feature.
Standard Capability	These capabilities indicate whether a certain function is supported by the port, and are unrelated to I/O configuration.

4.2.1. I/O Capability Groups

I/O capabilities describe features of physical I/O ports. The main I/O capability groups are listed in the table below. Capabilities in group **BOOLIO** are present, or not. Capabilities in the other I/O capability groups are mutually exclusive: only one of them can be active at a time.

Group	Description
BOOLIO	Boolean I/O capabilities that, if present, indicate that a feature is supported. Capabilities in this group include **FAILSAFE**, **FRACMODE**, **GENLOCKED**, **GENREF** and **SWS2APSK**.
IODIR	The direction of the signal flow: **INPUT**, **OUTPUT** or **DISABLED**. The sub capabilities in this group indicate how a physical port is connected to the input or output channel. This encodes features like double buffering.
IOSTD	The I/O standard used on this port. Capabilities in this group include: **3GSDI**, **ASI**, **DEMOD**, **GPSTIME**, **HDSDI**, **IP**, **MOD**, **PHASENOISE**, **SDI** and **SPI**.
PWRMODE	High-quality modulation (**MODHQ**) or low-power mode (**LOWPWR**)
RFCLKSEL	Modulator RF clock - Selection of reference source: internal (**RFXCLKINT**) or external (**RFXCLKEXT**).
TSRATESEL	Capabilities in this group selects between ways to generate the transport-stream rate: **EXTTSRATE**, **EXTRATIO**, **INTTSRATE** or **LOCK2INP**.

For a complete list of I/O capabilities, please refer to **CapList.xlsx**.

4.2.2. Standard Capability Groups

The main standard capability groups are listed in the table below.

Group	Description
DEMODPROPS	General demodulator properties: **ANTPWR** (antenna power), **LNB** and **RX_ADV** (advanced demodulation).
FREQBAND	Frequency band supported: **LBAND**, **VHF**, **UHF**.
IOPROPS	Miscellaneous capabilities that don't fit elsewhere, e.g. **TRPMODE** (transparent mode)
MODSTD	Modulation standards, all starting with **TX_**: **TX_ATSC**, **TX_DVBT2**, etc.
MODPROPS	Other capabilities (besides **MODSTD**) related to modulation, e.g. **CM** (channel simulation).
RXSTD	Receiver standards, all starting with **RX_**: **RX_ATSC**, **RX_DVBT2**, etc.

For a complete list of standard capabilities, please refer to **CapList.xlsx**.

4.3. I/O Configuration

4.3.1. SetIoConfig and GetIoConfig

Use the **SetIoConfig** to set the I/O configuration of a port, and **GetIoConfig** to read it back.

On Windows, I/O configuration settings are persisted in the registry. After a power down and a reboot, the I/O configurations will be automatically restored to the last-applied settings.

On Linux, no such mechanism exists and the application itself is responsible for configuring the ports.

Example

On many DekTec adapters, ports can be configured in ASI or in SDI mode. The code below configures a port in ASI mode:

```
if ((HwFuncDesc.m_Flags & DTAPI_CAP_ASI) != 0)
   Dvc.SetIoConfig(Port, DTAPI_IOCONFIG_IOSTD, DTAPI_IOCONFIG_ASI);
```

4.3.2. Relation to Capabilities

SetIoConfig and **GetIoConfig** have four parameters that are closely linked to capabilities:

Parameter	Description
Port	Physical port number
Group	Same as capability group. Only I/O capabilities have a corresponding I/O configuration group.
Value	Capability
SubValue	Sub-capability

Example

An output port that can be configured in "double-buffered" mode (output signal available on two ports) has capability **DTAPI_CAP_OUTPUT** and sub-capability **DTAPI_CAP_DBLBUF**, both located in the **IODIR** group.

To configure such a port for double-buffering, use:

```
Dvc.SetIoConfig(Port, DTAPI_IOCONFIG_IODIR,      // Group
                      DTAPI_IOCONFIG_OUTPUT,     // Value
                      DTAPI_IOCONFIG_DBLBUF);    // Subvalue
```

For a complete list of I/O configuration groups, values and subvalues, see the **DTAPI_IOCONIG_XXX** constants in **DTAPI.h**

4.3.3. SetIoConfig Variants

Two **SetIoConfig** functions are defined, one at device level and one at channel level. The I/O configuration of a port at device level can only be changed when the port is not used (no channel object attached). Some, but not all, I/O configuration changes can also be performed at channel level. This can only be done when the channel object is attached to the hardware.

In some cases there are dependencies between I/O ports on the same DekTec device. The driver validates whether the I/O configuration of multiple ports is consistent with each other. For example,

on the DTA-2137 only one port can be set to **`DTAPI_IOCONFIG_SWS2APSK`**, otherwise an error is returned.

To simplify configuration changes that must be done in a specific order, and to prevent temporary invalid configurations, a "transaction" variant of **`SetIoConfig`** is available. With this variant the I/O configuration settings only needs to be valid before and after the complete transaction, not after each individual configuration action.

5. DTAPI Concepts

5.1. Getting Statistics

DTAPI uses a new class to represent measurements (typically for receivers) and statistics: **DtStatistic**. A summary of its declaration is shown below. Refer to **DTAPI.h** for the full definition.

```
struct DtStatistic
{
  DtStatistic();
  DtStatistic(int StatisticId);    // Constructor with DTAPI_STAT_xxx initialization

  enum StatValueType
  {
     STAT_VT_UNDEFINED, STAT_VT_BOOL, STAT_VT_DOUBLE, STAT_VT_INT
  };
  DTAPI_RESULT  m_Result;          // Result of retrieving the statistic
  int  m_StatisticId;              // Identifies the statistic: DTAPI_STAT_XXX
  StatValueType  m_ValueType;      // Value type of statistic: STAT_VT_XXX
  union {
     bool  m_ValueBool;            // Value if value type is STAT_VT_BOOL
     double  m_ValueDouble;        // Value if value type is STAT_VT_DOUBLE
     int  m_ValueInt;              // Value if value type is STAT_VT_INT
  };
  DTAPI_RESULT  GetName(..), GetValue(..), SetId(..);
};
```

Statistics are identified by their ID (**m_StatisticId**). See **DTAPI.h** for a list of **DTAPI_STAT_XXX** identifiers. The function **GetName()** returns both a full name and a short name of the statistic. The value of the statistic can be retrieved with **GetValue()**.

Note: If the type of a statistic is **STAT_VT_INT**, its value can be retrieved both as **int** and as **double**.

The following statistics functions are available:

Function	Description
GetStatistic(int, int&) **GetStatistic(int, double&)** **GetStatistic(int, bool&)**	Return a single statistic
GetStatistics(int, DtStatistic*)	Return an array of statistics
GetSupportedStatistics(int&, DtStatistic*)	Returns all supported statistics on a port

5.2. Transmit on Timestamp

'Transmit on Timestamp' is a special transmission mode for ASI outputs to transmit transport packets at user-defined times. This enables generation of a jittered stream, a feature that can be used to build test generators that simulate actual network conditions.

The transmit-on-timestamp mode is enabled by including the **DTAPI_TXMODE_TXONTIME** flag in the first argument of the **SetTxMode** method. Timestamps reference the 54-MHz system reference clock.

The data format of a transmit-on-timestamp stream is described in §9.7. Timestamps are stored in little-endian format in 4 bytes that are located before each packet. The format is identical to that generated by an ASI receive channel in time-stamped mode. Timestamps are used only for timing of the output stream; the timestamps itself are not transmitted.

It is important that the data in txontime_stream() is formatted correctly from the start. When the stream is not aligned correctly, data in the stream may be interpreted as timestamp, potentially causing long delays between transmission of packets. The device cannot automatically recover from this situation and a channel reset is required to resume synchronised operation.

The following limitations apply:

- Transmit on timestamp is not supported in raw mode (**DTAPI_TXMODE_RAW**);
- Null packet stuffing (**DTAPI_TXSTUFF_MODE_ON**) is not supported.
- In transmit-on-timestamp mode, the transmit channel will automatically operate in burst mode, even if the **DTAPI_TXMODE_BURST** flag is not specified.

When starting transmission by setting transmit control to **DTAPI_TXCTRL_SEND**, the timestamp of the first packet is stored as reference and the packet is sent out immediately. For all other packets the number of 54-MHz cycles relative to the timestamp of the first packet is computed, and the packets are sent out the computed number of cycles after the first packet.

5.3. SDI Genlock Support

SDI I/O adapters with an on-board VCXO (DTA-145, DTA-2144 and DTA-2145) are capable of 'SDI genlock'. The SDI output(s) can be locked to an incoming SDI signal, such that the Start-of-Active-Video (SAV) symbol is sent on the SDI output at virtually the same time as the SAV symbol is entering the SDI input.

To genlock an SDI output, an application shall do the following:

1. Set the I/O configuration for port #1 to **DTAPI_IOCONFIG_GENREF** and specify the video standard the port should lock to. Once port #1 is configured as GENREF input, the driver will extract the SDI timing from the SDI signal presented to the port.
2. Set the I/O configuration for the output port to **DTAPI_IOCONFIG_GENLOCKED**.
3. Attach an output channel object to the output port and set TxMode to match the configured reference video standard.

```
// Pre-condition: Dvc is attached to

// Configure port #1 as genlock reference input for SDI625
Dvc.SetIoConfig(1, DTAPI_IOCONFIG_GENREF, DTA1XX_GENLOCK_SDI625);

// Configure port #2 as genlocked output port
Dvc.SetIoConfig(2, DTAPI_IOCONFIG_GENLOCKED);

// Attach to port #2
DtOutp.AttachToPort(&DtDvc, 2);

// Initialise channel to initial 'safe' state
DtOutp.SetTxControl(DTAPI_TXCTRL_IDLE);

// Set the TxMode to SDI
DtOutp.SetTxMode(DTAPI_TXMODE_SDI_FULL, DTAPI_TXSTUFF_MODE_ON);
etc.
```

Figure 11. Configuring genlock.

It is not necessary to attach to the genlock reference port. The application, or another application, may still open the port as an input, with the limitation that port #1 must be operated in an SDI mode that matches the configured reference video standard.

5.4. Vital Product Data (VPD)

Vital Product Data (VPD) is product identification information stored in an EEPROM on board of DekTec devices. The read-only part of VPD is loaded in the manufacturing process. The read/write part is used for licensing purposes and for storing customer-specific product information.

VPD is initialized as a collection of items, each identified by a keyword. Most keywords are 2-character strings (e.g. "PD" for Production Date), with the exception of the VPD ID String, which is identified by "VPDID".

Three member functions of DtDevice are defined to manipulate VPD:

- **`VpdRead`** – Read VPD-item, given a keyword.
- **`VpdWrite`** – Write VPD-item, given a keyword and item string. If the item existed, the item string is overwritten, unless the VPD item is read-only, in which case an error code is returned.
- **`VpdDelete`** – Delete VPD item. Read-only VPD item cannot be deleted.

6. Multi-PLP Extensions

Multi-PLP modulation is a specific DTAPI function that enables application programs to create single-PLP and multi-PLP modulators for ATSC 3.0, DVB-C2, DVB-T2 and ISDB-Tmm. The classes and structures that are related to multi-PLP modulation are specified in the document *DTAPI Reference – Multi-PLP Extensions*. The main DTAPI header file and library include the class definitions required for multi-PLP modulation. This section describes the usage of these classes and structures.

6.1. Licensing

The multi-PLP classes require an **`ATSC 3.0`** (DTC-386), a **`DVB-C2`** (DTC-379), a **`DVB-T2`** (DTC-378)), an **`ISDB-Tmm`** (DTC-382) or a **`GOLD`** license on the modulator card. If access to I/Q samples is required, an additional **`IQ`** license (DTC-371) must be present.

6.2. Multi-PLP Object Model

The multi-PLP modulator is represented by a "Multi-PLP Modulator" object that is encapsulated by the **`DtMplpOutpChannel`** class. When multi-PLP modulation parameters are set through a **`SetModControl`** method, multi-PLP modulation is enabled and input FIFOs are created for each PLP source. Methods are provided to write into the individual MPLP FIFOs and to control them. The **`DtMplpOutpChannel`** object transfers the modulation results through the device driver to the device.

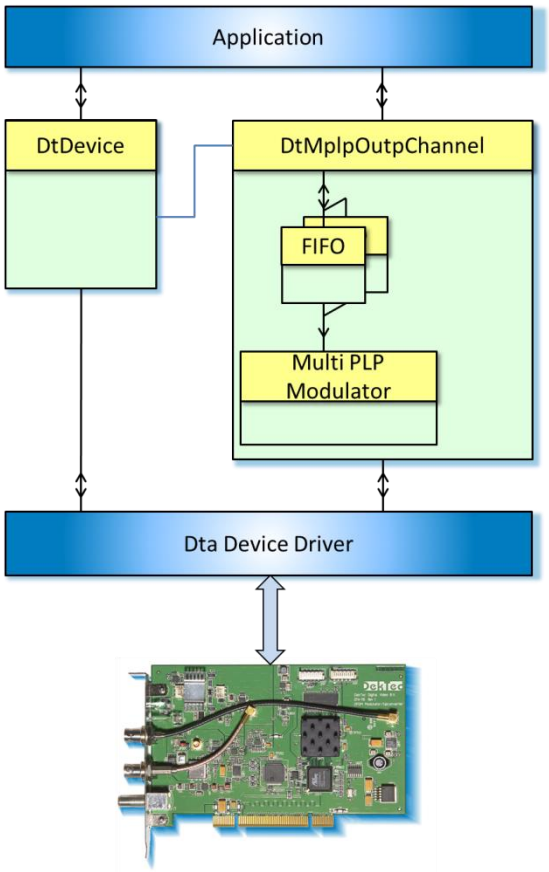

Figure 12. Example of a **`DtDevice`** object and a **`DtMplpOutpChannel`** object encapsulating a multi-PLP modulator.

In case single-PLP modulation parameters are set, only one FIFO is created and the multi-PLP modulator acts as a single-PLP modulator.

6.3. Attaching to a Multi-PLP Modulator

Using the DTAPI multi-PLP extensions is no different from using a standard modulator channel, except that **DtMplpOutpChannel** is used instead of **OutpChannel**. First, a **DtDevice** object has to be instantiated and attached to the hardware, and then a **DtMplpOutpChannel** object has to be attached to the device.

```
// Error-handling code has been omitted
DtDevice Dvc;
Dvc.AttachToSerial(4115123456);
DtMplpOutpChannel Outp;
Outp.AttachToPort(&Dvc, 1);
```

Figure 13. Attaching a multi-PLP modulator channel to the hardware.

6.4. Virtual Channels

A standard output channel writes modulated I/Q samples directly to the hardware. The DTAPI multi-PLP extensions support a new type of channel, a *virtual* channel, enabling custom processing of the multi-PLP modulator output. For example, the modulated I/Q samples can be written to a file.

A virtual channel can be created using the channel's **AttachVirtual** member function. The first parameter of this function, a pointer to a **DtDevice** object, identifies the hardware device carrying the licenses to enable the MPLP extensions. The second parameter specifies the callback function and the third parameter an opaque pointer. When DTAPI has generated new output, the callback function is invoked with the opaque pointer and I/Q samples as arguments

Example code to create a virtual channel is shown in Figure 14.

```
{
  DtDevice Dvc;
  // Code to attach to device goes here

  DtMplpOutpChannel Outp;
  if (Outp.AttachVirtual(&Dvc, ::WriteMySsmps, NULL) != DTAPI_OK)
  {
    // Error-handling code
  }
  etc.
}

bool WriteMySamps(void* pOpaque, void* pVirtOut)
{
  // Code processing the generated data,
  // e.g. writing to file
}
```

Figure 14. Attaching a **DtMplpOutpChannel** object to a virtual output.

To avoid memory leaks, a virtual **DtMplpOutpChannel** object shall be detached from the hardware after all operations on the channel have been completed.

6.5. Streaming MPLP Data

The core of a multi-PLP modulator program is shown in Figure 15. The code assumes:

- **DtDevice** object **Dvc** and **DtMplpOutpChannel** object **Outp** have been attached to the hardware;
- Multi-PLP modulation parameters have been set;

- **GetTsData(i, Buf, Max)** is the user-supplied function that writes maximally **Max** new Transport-Stream data bytes in **Buf** for MPLP-FIFO/PLP index **i**, and returns the number of bytes written.

The transmission control is set to **Hold**, which enables multi-PLP modulation and DMA but keeps actual transmission disabled.

```
// PRE: Dvc and Outp attached
//      MPLP modulation parameters set
char Buf[BUFSIZE];
int NumBytes = 1;

int TxControl = DTAPI_TXCTRL_HOLD;
Outp.SetTxControl(TxControl);

// Main loop
while (NumBytes != 0)
{
  // Transmission in hold?
  if (TxControl == DTAPI_TXCTRL_HOLD)
  {
    // Check whether initial load reached
    int Load;
    Outp.GetFifoLoad(Load);
    if (Load >= INILOAD)
    {
      TxControl = DTAPI_TXCTRL_SEND;
      Outp.SetTxControl(TxControl);
    }
  }

  // Try to fill all input FIFOs
  bool AllFifosFilled = true;
  for (int i=0; i<NumInputs && NumBytes!=0; i++)
  {
    // MPLP FIFO (still) filled?
    int NumFree;
    Outp.GetMplpFifoFree(i, NumFree);
    if (NumFree < BUFSIZE)
      continue; // Yes; Next FIFO

    AllFifosFilled = false;
    NumBytes = GetTsData(i, Buf, BUFSIZE);
    Outp.WriteMplp(i, Buf, NumBytes);
  }

  // All FIFOs filled?
  if (AllFifosFilled)
     Sleep(10);    // Sleep for a while
}
```

Figure 15. Streaming data to an output.

When the Transmit FIFO contains its initial load, actual transmission can be started by setting transmission control to **Send**. The main loop then supplies additional data to the MPLP FIFOs until the data sources are exhausted.

The following factors should be considered to achieve optimal results:

- Modulation of a frame is only possible when sufficient data is available for all PLPs. A lengthy transfer to one MPLP FIFO may cause underflow of another MPLP FIFO, stalling the modulation process. To prevent this, the transfer size should not be too large. For efficiency reasons, the transfer

size should not be too small either. Therefore it is recommended to use a transfer size between 4K bytes and 32K bytes.

- The initial transmit-FIFO load (**INILOAD**) should not be too small, to prevent an early transmit-FIFO underflow in the main loop. A value close to the maximum hardware FIFO size is recommended. *Warning*: The initial load cannot be larger than the transmit-FIFO size: when the transmit FIFO is full, DMA will stall and the application "hangs."
- As far as DTAPI is concerned, the **GetTsData** function may return Transport-Stream data aligned at arbitrary 4-byte boundaries. However, for many data-generating algorithms, alignment on packet boundaries will be a natural choice. In such applications it is convenient and efficient to set the buffer size to a multiple of the packet size.

6.6. Complete Example

Figure 16 shows the code of a simple DVB-T2 stream generator containing 2 data PLPs and a common PLP.

Obviously, this example is just a first step towards a production-quality stream generator application.

```
// Command-line program T2Sample
// Outputs DVB-T2 signal according to V&V402 through DTA-115
#include "DTAPI.h"
#include <stdio.h>

int main(int argc, char* argv[])
{
    char TempRdBuf[8192];
    DTAPI_RESULT dr;
    DtDevice Dvc;
    DtMplpOutpChannel Outp;

    // Attach to the DTA-115
    dr = Dvc.AttachToType(115);
    if (dr != DTAPI_OK)
        exit(dr);

    // Use the modulator port
    dr = Outp.AttachToPort(&Dvc, 2);
    if (dr != DTAPI_OK)
        exit(dr);

    // Set RF frequency to 666MHz
    dr = Outp.SetRfControl(666000000);
    if (dr != DTAPI_OK)
        exit(dr);

    // Set RF level -20.0 dBm
    dr = Outp.SetOutputLevel(-200);
    if (dr != DTAPI_OK)
        exit(dr);

    // Set default DVB-T2 values
    DtDvbT2Pars DvbT2Pars;

    // Below you'll find the parameter setting according to VV402
    // General parameters
    DvbT2Pars.m_T2Version       = DTAPI_DVBT2_VERSION_1_2_1;
    DvbT2Pars.m_Bandwidth       = DTAPI_DVBT2_8MHZ;
    DvbT2Pars.m_FftMode         = DTAPI_DVBT2_FFT_32K;
    DvbT2Pars.m_GuardInterval   = DTAPI_DVBT2_GI_1_128;
    DvbT2Pars.m_Miso            = DTAPI_DVBT2_MISO_OFF;
    DvbT2Pars.m_Papr            = DTAPI_DVBT2_PAPR_NONE;
    DvbT2Pars.m_BwtExt          = true;
    DvbT2Pars.m_PilotPattern    = 7;
    DvbT2Pars.m_L1Modulation    = DTAPI_DVBT2_BPSK;
    DvbT2Pars.m_CellId          = 0;
    DvbT2Pars.m_NetworkId       = 12421;
    DvbT2Pars.m_T2SystemId      = 32769;
    DvbT2Pars.m_L1Repetition    = false;

    // T2-Frame related parameters
    DvbT2Pars.m_NumT2Frames     = 2;
    DvbT2Pars.m_NumDataSyms     = 27;
    DvbT2Pars.m_NumSubslices    = 108;

    // No FEF
    DvbT2Pars.m_FefEnable       = false;
```

```
// 1 RF channel
DvbT2Pars.m_NumRfChans       = 1;
DvbT2Pars.m_StartRfIdx       = 0; // n.a. for non-TFS
DvbT2Pars.m_RfChanFreqs[0]   = 666000000;

// 3 PLPs
DvbT2Pars.m_NumPlps          = 3;

// PLP[0] First data PLP
DvbT2Pars.m_Plps[0].m_Id                = 0;
DvbT2Pars.m_Plps[0].m_GroupId           = 0;
DvbT2Pars.m_Plps[0].m_Type              = DTAPI_DVBT2_PLP_TYPE_2;
DvbT2Pars.m_Plps[0].m_Modulation        = DTAPI_DVBT2_QPSK;
DvbT2Pars.m_Plps[0].m_CodeRate          = DTAPI_DVBT2_COD_1_2;
DvbT2Pars.m_Plps[0].m_FecType           = DTAPI_DVBT2_LDPC_64K;
DvbT2Pars.m_Plps[0].m_Hem               = true;
DvbT2Pars.m_Plps[0].m_Npd               = true;
DvbT2Pars.m_Plps[0].m_Issy              = DTAPI_DVBT2_ISSY_LONG;
DvbT2Pars.m_Plps[0].m_IssyBufs          = 1048576;
DvbT2Pars.m_Plps[0].m_IssyTDesign       = 949777;
DvbT2Pars.m_Plps[0].m_CompensatingDelay = -1;   // Auto
DvbT2Pars.m_Plps[0].m_TimeIlType        = DTAPI_DVBT2_IL_ONETOONE;
DvbT2Pars.m_Plps[0].m_TimeIlLength      = 1;
DvbT2Pars.m_Plps[0].m_FrameInterval     = 1;
DvbT2Pars.m_Plps[0].m_FirstFrameIdx     = 0;
DvbT2Pars.m_Plps[0].m_Rotation          = true;
DvbT2Pars.m_Plps[0].m_InBandAFlag       = true;
DvbT2Pars.m_Plps[0].m_NumOtherPlpInBand = 0;
DvbT2Pars.m_Plps[0].m_InBandBFlag       = false;
DvbT2Pars.m_Plps[0].m_FfFlag            = false;
DvbT2Pars.m_Plps[0].m_FirstRfIdx        = 0;
DvbT2Pars.m_Plps[0].m_NumBlocks         = 14;
DvbT2Pars.m_Plps[0].m_TsRate            = 6000000;

// PLP[1] Second data PLP
DvbT2Pars.m_Plps[1].m_Id                = 1;
DvbT2Pars.m_Plps[1].m_GroupId           = 0;
DvbT2Pars.m_Plps[1].m_Type              = DTAPI_DVBT2_PLP_TYPE_2;
DvbT2Pars.m_Plps[1].m_Modulation        = DTAPI_DVBT2_QPSK;
DvbT2Pars.m_Plps[1].m_CodeRate          = DTAPI_DVBT2_COD_1_2;
DvbT2Pars.m_Plps[1].m_FecType           = DTAPI_DVBT2_LDPC_64K;
DvbT2Pars.m_Plps[1].m_Hem               = true;
DvbT2Pars.m_Plps[1].m_Npd               = true;
DvbT2Pars.m_Plps[1].m_Issy              = DTAPI_DVBT2_ISSY_LONG;
DvbT2Pars.m_Plps[1].m_IssyBufs          = 1048576;
DvbT2Pars.m_Plps[1].m_IssyTDesign       = 949777;
DvbT2Pars.m_Plps[1].m_CompensatingDelay = -1;   // Auto
DvbT2Pars.m_Plps[1].m_TimeIlType        = DTAPI_DVBT2_IL_ONETOONE;
DvbT2Pars.m_Plps[1].m_TimeIlLength      = 1;
DvbT2Pars.m_Plps[1].m_FrameInterval     = 1;
DvbT2Pars.m_Plps[1].m_FirstFrameIdx     = 0;
DvbT2Pars.m_Plps[1].m_Rotation          = true;
DvbT2Pars.m_Plps[1].m_InBandAFlag       = true;
DvbT2Pars.m_Plps[1].m_NumOtherPlpInBand = 0;
DvbT2Pars.m_Plps[1].m_InBandBFlag       = false;
DvbT2Pars.m_Plps[1].m_FfFlag            = false;
DvbT2Pars.m_Plps[1].m_FirstRfIdx        = 0;
DvbT2Pars.m_Plps[1].m_NumBlocks         = 14;
DvbT2Pars.m_Plps[1].m_TsRate            = 6000000;

// PLP[2] Common PLP
```

```
DvbT2Pars.m_Plps[2].m_Id                = 2;
DvbT2Pars.m_Plps[2].m_GroupId           = 0;
DvbT2Pars.m_Plps[2].m_Type              = DTAPI_DVBT2_PLP_TYPE_COMM;
DvbT2Pars.m_Plps[2].m_Modulation        = DTAPI_DVBT2_QPSK;
DvbT2Pars.m_Plps[2].m_CodeRate          = DTAPI_DVBT2_COD_1_2;
DvbT2Pars.m_Plps[2].m_FecType           = DTAPI_DVBT2_LDPC_16K;
DvbT2Pars.m_Plps[2].m_Hem               = true;
DvbT2Pars.m_Plps[2].m_Npd               = true;
DvbT2Pars.m_Plps[2].m_Issy              = DTAPI_DVBT2_ISSY_LONG;
DvbT2Pars.m_Plps[2].m_IssyBufs          = 1048576;
DvbT2Pars.m_Plps[2].m_IssyTDesign       = 949777;
DvbT2Pars.m_Plps[2].m_CompensatingDelay = -1;   // Auto
DvbT2Pars.m_Plps[2].m_TimeIlType        = DTAPI_DVBT2_IL_ONETOONE;
DvbT2Pars.m_Plps[2].m_TimeIlLength      = 1;
DvbT2Pars.m_Plps[2].m_FrameInterval     = 1;
DvbT2Pars.m_Plps[2].m_FirstFrameIdx     = 0;
DvbT2Pars.m_Plps[2].m_Rotation          = true;
DvbT2Pars.m_Plps[2].m_InBandAFlag       = true;
DvbT2Pars.m_Plps[2].m_NumOtherPlpInBand = 0;
DvbT2Pars.m_Plps[2].m_InBandBFlag       = false;
DvbT2Pars.m_Plps[2].m_FfFlag            = false;
DvbT2Pars.m_Plps[2].m_FirstRfIdx        = 0;
DvbT2Pars.m_Plps[2].m_NumBlocks         = 9;
DvbT2Pars.m_Plps[2].m_TsRate            = 6000000;


// PLP Inputs
// PLP[0] input uses MPLP FIFO index 0
DvbT2Pars.m_PlpInputs[0].m_DataType     = DtPlpInpPars::TS188;
DvbT2Pars.m_PlpInputs[0].m_FifoIdx      = 0;
DvbT2Pars.m_PlpInputs[0].m_BigTsSplit.m_Enabled = false;

// PLP[1] input uses MPLP FIFO index 1
DvbT2Pars.m_PlpInputs[1].m_DataType     = DtPlpInpPars::TS188;
DvbT2Pars.m_PlpInputs[1].m_FifoIdx      = 1;
DvbT2Pars.m_PlpInputs[1].m_BigTsSplit.m_Enabled = false;

// PLP[2] input uses MPLP FIFO index 2
DvbT2Pars.m_PlpInputs[2].m_DataType     = DtPlpInpPars::TS188;
DvbT2Pars.m_PlpInputs[2].m_FifoIdx      = 2;
DvbT2Pars.m_PlpInputs[2].m_BigTsSplit.m_Enabled = false;

// No virtual output is used through callback functions
DvbT2Pars.m_VirtOutput.m_Enabled = false;

// No test point data output
DvbT2Pars.m_TpOutput.m_Enabled = false;

// No PAPR ACE
DvbT2Pars.m_PaprPars.m_AceEnabled   = false;

// Only P2 P2 PAPR TR
DvbT2Pars.m_PaprPars.m_TrEnabled    = true;
DvbT2Pars.m_PaprPars.m_TrP2Only     = true;
DvbT2Pars.m_PaprPars.m_TrMaxIter    = 1;
DvbT2Pars.m_PaprPars.m_TrVclip      = 4.32;

// Enable L1 PAPR
DvbT2Pars.m_PaprPars.m_L1AceEnabled = true;
DvbT2Pars.m_PaprPars.m_L1AceCMax    = 0.0;
```

```
// PAPR Bias ballancing and bias ballancing cells
DvbT2Pars.m_PaprPars.m_BiasBalancing   = 1;
DvbT2Pars.m_PaprPars.m_NumBiasBalCells = 0;

// No TX signalling
DvbT2Pars.m_TxSignature.m_TxSigAuxEnabled   = false;
DvbT2Pars.m_TxSignature.m_TxSigFefEnabled   = false;

// We have RF output so no T2MI output
DvbT2Pars.m_T2Mi.m_Enabled  = false;

// No RBM validation
DvbT2Pars.m_RbmValidation.m_Enabled = false;

// Check whether parameters are valid
dr = DvbT2Pars.CheckValidity();
if (dr != DTAPI_OK)
    exit(dr);

// Get the TSRates of the PLPs
for (int i=0; i<3; i++)
{
    int TsRate;
    DtapiModPars2TsRate(TsRate, DvbT2Pars, i);
    printf("TS-rate PLP[%d]: %d bps\n", i, TsRate);
}

// Set transmitter to IDLE
dr = Outp.SetTxControl(DTAPI_TXCTRL_IDLE);
if (dr != DTAPI_OK)
    exit(dr);

// Initialize the modulator.
dr = Outp.SetModControl(DvbT2Pars);
if (dr != DTAPI_OK)
    exit(dr);

// Set transmitter to HOLD
dr = Outp.SetTxControl(DTAPI_TXCTRL_HOLD);
if (dr != DTAPI_OK)
    exit(dr);
bool InSendMode = false;    // Not in SEND mode (yet)

// Determine the FIFO load threshold
int RfFifoSize;
dr = Outp.GetFifoSize(RfFifoSize);
if (dr != DTAPI_OK)
    exit(dr);

// Threshold is set to 75% of the FIFO size
int IniLoad = 3*RfFifoSize / 4;

// Open the input files and check the opened files
const int NumInputs = 3;
FILE* Files[NumInputs];
Files[0] = fopen("C:\\Data\\VV402_Plp0.ts", "rb");
Files[1] = fopen("C:\\Data\\VV402_Plp1.ts", "rb");
Files[2] = fopen("C:\\Data\\VV402_Plp2.ts", "rb");
if (Files[0]==NULL || Files[1]==NULL || Files[2]==NULL)
    dr = DTAPI_E;
```

```
    printf("Press any key to stop...");

    // Do while no keyboard key is hit and all is OK
    while (!_kbhit() && dr == DTAPI_OK)
    {
        // If not in SEND mode yet, check whether we can go to SEND mode
        if (!InSendMode)
        {
            // Get the FIFO load of the RF output
            int RfFifoLoad;
            dr = Outp.GetFifoLoad(RfFifoLoad);
            if (dr != DTAPI_OK)
                break;

            if (RfFifoLoad >= IniLoad)
            {
                // Goto SEND mode
                dr = Outp.SetTxControl(DTAPI_TXCTRL_SEND);
                if (dr != DTAPI_OK)
                    break;

                // Now we can enter SEND mode
                InSendMode = true;
            }
        }

        // Lets assume all MPLP FIFOs are filled until found otherwise
        bool AllFifosFilled = true;
        for (int FifoIdx=0; FifoIdx<NumInputs && dr == DTAPI_OK; FifoIdx++)
        {
            // Check the amount free in the MPLP FIFO
            int NumFree;
            dr = Outp.GetMplpFifoFree(FifoIdx, NumFree);
            if (dr != DTAPI_OK)
                break;
            // Skip this MPLP FIFO if too less room is available
            if (NumFree < sizeof(TempRdBuf))
                continue;   // next FIFO

            // This MPLP FIFO is not filled enough
            AllFifosFilled = false;

            // Read a chunck of data
            int NumRead = (int)::fread(TempRdBuf, 1, sizeof(TempRdBuf),
                                                          Files[FifoIdx]);
            // EOF? then goto begin of file
            if (feof(Files[FifoIdx]))
                ::fseek(Files[FifoIdx], 0, SEEK_SET);

            // Write the data to the MPLP FIFO
            dr = Outp.WriteMplp(FifoIdx, TempRdBuf,  NumRead);
            if (dr != DTAPI_OK)
                break;
        }

        // All FIFOs filled? then sleep for a while, to prevent an endless loop.
        if (AllFifosFilled)
            Sleep(10);
    }

    // Get and print the status of the DVB-T2 modulation
```

```
    DtDvbT2ModStatus ModStatus;
    Outp.GetMplpModStatus(&ModStatus );
    printf("\nDVB-T2 Modulator Status:"
            "\n\t#BitrateOVF: %I64d"
            "\n\t#BlockOVF  : %I64d"
            "\n\t#TTO-Error : %I64d\n",
            ModStatus.m_BitrateOverflows,
            ModStatus.m_PlpNumBlocksOverflows,
            ModStatus.m_TtoErrorCount);

    // Set transmitter to IDLE again
    Outp.SetTxControl(DTAPI_TXCTRL_IDLE);

    // Detach hardware
    Outp.Detach(DTAPI_INSTANT_DETACH);
    Dvc.Detach();

    // Close the input files
    for (int i=0; i<NumInputs; i++)
        if (Files[i] != NULL)
            fclose(Files[i]);

    return dr;
}
```

Figure 16. DVB-T2 stream generator with the DTA-115.

7. Advanced Demodulator API

7.1. Introduction

The advanced demodulator API is a subsystem of DTAPI that supports the reading of one or multiple real-time streams and getting advanced measurements using SDR (Software Defined Radio) techniques. Each stream can be a data stream or a stream of measurement values.

Capability `RX_ADV`, which requires a license, enables the advanced demodulator API. It is only available on receiver devices that can receive and demodulate I/Q samples, at the moment only the DTA-2131.

7.2. Streaming Model

The streams are generated with call-back functions that are to be provided by the DTAPI user. Multiple parallel streams can be generated in parallel:

- For DVB-C2 and DVB-T2: multiple PLPs can be generated in parallel. Data PLPs can be combined with common PLPs;
- For DVB-T2 a T2MI stream with all PLPs embedded can be generated[2];
- For ISDB-T, layer A, B and C can be demodulated in parallel.

The classes and structures that are related to Advanced Demodulator are specified in the document *DTAPI Reference – Advanced Demodulator API*. The main DTAPI header file and library include the class definitions required for the Advanced Demodulator. This section describes the usage of these classes and structures.

7.3. Licensing

Standard demodulation in DTAPI without advanced measurements doesn't require a license. It doesn't matter whether the underlying device is a hardware demodulator, such as the DTA-2138 DVB-T2 / DVB-C2 receiver, or an I/Q demodulator card such as the DTA-2131.

The following receiver functions are available without license:

- Single-PLP DVB-T2 or DVB-C2;
- Single stream reception, e.g. ISDB-T.

The following licenses are available to enable advanced receiver functionality:

- The `RXA` license (DTC-360) enables usage of the Advanced Demodulator API classes;
- The `IQ` license (DTC-361) enables direct access to I/Q samples;
- The `T2MI` license (DTC-362) allows access to a full DVB-T2 T2MI stream.

7.4. Advanced Demodulator Object Model

The advanced demodulator is represented by the `DtAdvDemod` class. The usage of the advanced classes differs significantly from the "normal" DTAPI input channels.

- The standard input channel class `DtInpChannel` uses a FIFO to store the demodulated data packets and a read function is used to process the packets.
- The advanced demodulator class `DtAdvDemod` doesn't store the data but uses call-back functions to convey Transport Packets or measurements values. By registering multiple call-backs, the user can receive multiple PLP and stream with measurement values simultaneously.

[2] Either a T2MI stream can be generated, or one or more PLPs, but not both at the same time.

7.5. Attaching to an Advanced Demodulator

Attaching an advanced demodulator object to a device is no different from using a standard demodulator channel, except that **DtAdvDemod** is used instead of **DtInpChannel**. First, a **DtDevice** object has to be instantiated and attached to the hardware, and then a **DtAdvDemod** object can be attached to the device.

```
// Error-handling code has been omitted
DtDevice Dvc;
Dvc.AttachToSerial(2131123456);

// Attach an advanced demodulator object to the device
DtAdvDemod AdvDemod;
AdvDemod.AttachToPort(&Dvc, 1);
```

Figure 17. Attaching a **DtAdvDemod** object to the hardware.

7.6. Virtual Input Channel – User-Supplied I/Q Samples

The advanced demodulator API supports a new type of input channel, a *virtual* input, enabling processing of user-supplied I/Q samples. It enables the user to feed the advanced demodulator with I/Q samples, e.g. from file, instead of DTAPI reading the data from a physical receiver such as the DTA-2131.

A virtual channel can be created with the **AttachVirtual** member function. The first parameter of this function is a pointer to a **DtDevice** object. This device is only required to hold the license for the advanced demodulator API (license **RX_ADV**). The second parameter specifies the call-back function for obtaining the I/Q samples, while the third parameter is an opaque pointer. When the advanced demodulator requires new samples, the call-back function is invoked with the opaque pointer and an I/Q sample sample buffer as arguments.

Example code to create a virtual channel is shown in Figure 18.

```
{
  // Device is only used for holding the RX_ADV license
  DtDevice LicDvc;
  // Code to attach to device goes here

  DtAdvDemod AdvDemod;
  if (AdvDemod.AttachVirtual(&LicDvc, ::ReadIqSamps, NULL) != DTAPI_OK)
  {
    // Error-handling code
  }
  etc.
}

void ReadIqSamps(void* pOpaque, unsigned char* pIqBuf,
                                                int IqBufSize, int& IqLength)
{
  // Code to get I/Q samples (eg from file) and write to pIqBuf
}
```

Figure 18. Attaching a **DtAdvDemod** object to a virtual input.

To avoid memory leaks, a virtual **DtAdvDemod** object shall be detached from the hardware after all operations on the channel have been completed.

7.7. Receiving PLP Data and Constellation points

The core of an application using the advanced demodulator is shown in Figure 19. The code assumes that an `DtAdvDemod` object `T2In` has been attached to the hardware.

This example demodulates one PLP and for the same PLP the constellation points are streamed to the application using call-back functions. These "streaming" callback functions should not block and should be kept short in processing time to avoid that the advanced demodulator stalls. This example could easily be extended to demodulate all PLPs and retrieve multiple streams with measurements

simultaneously. The configuration parameters for stream selection are explained in detail in the document *DTAPI Reference – Advanced Demodulator API*.

```
// Select DVB-T2 demodulation, 8MHz bandwidth
DtDemodPars* pModPars = new DtDemodPars();
pModPars->SetModType(DTAPI_MOD_DVBT2);
DtDemodParsDvbT2* T2Pars = pModPars->DvbT2();
T2Pars->m_Bandwidth = DTAPI_DVBT2_8MHZ;
T2Pars->m_T2Profile = 0;
T2In.SetDemodControl(pModPars);
T2In.SetTunerFrequency(666000000);

// Create DVB-T2 selection parameters with PLP number
DtDvbT2StreamSelPars T2StreamSelPars;
T2StreamSelPars.m_PlpId = 0;
T2StreamSelPars.m_CommonPlpId = -1;      // Don't use a common PLP

// Open a PLP stream using the selection parameters
DtStreamSelPars StreamSelPars;
StreamSelPars.m_Id = 1;                  // Unique ID
StreamSelPars.m_Type = DtStreamSelType::STREAM_DVBT2;
StreamSelPars.u.m_DvbT2 = T2StreamSelPars;
T2In.OpenStream(StreamSelPars);

// Select a stream of constellation points
DtConstelPlotSelPars ConstellationPars;
ConstellationPars.m_Index = 0;           // PLP index
ConstellationPars.m_MaxNumPoints = 500;
ConstellationPars.m_ConstellationType = 0;
ConstellationPars.m_Period = 100;  // 100ms

// Open this stream of constellation points
StreamSelPars.m_Id = 2;                  // Unique ID
StreamSelPars.m_Type = DtStreamSelType::STREAM_CONSTEL;
StreamSelPars.u.m_Constel = ConstellationPars;
T2In.OpenStream(StreamSelPars);

// Streaming data callback functions
T2In.RegisterCallback(WriteStreamFunc, NULL);
T2In.RegisterCallback(WriteMeasFunc, NULL);

// Start advanced demodulation
T2In.SetRxControl(DTAPI_RXCTRL_RCV);

// Callback functions
static void WriteStreamFunc(void* pOpaque, DtStreamSelPars& StreamSel,
                                        const unsigned char* pData, int Length)
{
  if (StreamSel.m_Id == 1)
     // Process transport packets
}

static void WriteMeasFunc(void* pOpaque, DtStreamSelPars& StreamSel,
                                                    DtMeasurement* pMeasurement)
{
  if (StreamSel.m_Id == 2 &&
           pMeasurement ->m_MeasurementType == DtStreamSelType::STREAM_CONSTEL)
     // Process constellation points
}
```

Figure 19. Receiving a PLP and constellation points from a DVB-T2 signal.

7.8. Retrieving Statistics

The **GetStatistics** method retrieves dynamic statistical information about the input signal. The example in Figure 20 shows how to retrieve LDPC related statistics and the DVB-T2 L1 data structure. Demodulation statistics can be retrieved using the **DtInpChannel** or the **DtAdvDemod** class and are explained in detail in the document *DTAPI Reference – Advanced Demodulator API*.

```
DtDemodLdpcStats LdpcStats;
DtDemodLdpcStats* pLdpcStats = &LdpcStats;
DtDvbT2DemodL1Data L1data;
DtDvbT2DemodL1Data* pL1Data = &L1data;

DtStatistic Stats[2];
Stats[0].SetId(DTAPI_STAT_LDPC_STATS);
Stats[0].m_IdXtra[0] = 0; // plp id
Stats[1].SetId(DTAPI_STAT_DVBT2_L1DATA);

DTAPI_RESULT dr = TunPort.GetStatistics(2, Stats);

// Get LDPC statistics
Stats[0].GetValue(pLdpcStats);
if (Stats[0].m_Result == DTAPI_OK)
  printf("FEC max it: %d\n", pLdpcStats->m_FecBlocksItMax);
else
  printf("Result: %d\n",Stats[0].m_Result);

// Get DVB-T2 L1 data
Stats[1].GetValue(pL1Data);
if (Stats[1].m_Result == DTAPI_OK && pL1Data->m_L1Post.m_Valid)
  printf("MOD: %d\n", pL1Data->m_L1Post.m_Plps.at(0).m_Modulation);
else
  printf("Result: %d\n",Stats[1].m_Result);
```

Figure 20. Retrieve LDPC and DVB-T2 L1 statistical data.

7.9. Set generic demodulation parameters

The software demodulation core has some generic parameters that are not specific to a demodulation standard. These parameters can be set using the **SetPars** method in **DtInpChannel** or **DtAdvDemod**. The example in Figure 21 sets two parameters that influence the CPU usage for the advanced demodulation software.

```
// Configure average LDPC iterations, to set a limit on CPU usage
DtPar Pars[2];
Pars[0].m_ParId = DTAPI_PAR_DEMOD_LDPC_AVG;
Pars[0].m_ValueType = DtPar::ParValueType::PAR_VT_INT;
Pars[0].SetValue(3);

// Configure if MER measurement should be done (will influence CPU load)
Pars[1].m_ParId = DTAPI_PAR_DEMOD_MER_ENA;
Pars[1].m_ValueType = DtPar::ParValueType::PAR_VT_BOOL;
Pars[1].SetValue(true);

DTAPI_RESULT dr = T2In.SetPars(2,Pars);
if (dr != DTAPI_OK)
  printf("SetPars() failed.\n");
```

Figure 21. Configure CPU usage for demodulation using **SetPars**

8. SDI over IP

8.1. Overview

SDI-over-IP is encapsulating an SDI stream in IP packets, transmitting it over an IP network and de-encapsulating the stream back to SDI.

The DekTec network cards (DTA-160, DTA-2160, DTA-2162) support SD-SDI over RTP conforming to the SMPTE-2022-5, SMPTE-2022-6 and SMPTE-2022-7 specifications. HD-SDI and 3G-SDI are not (yet) supported.

DTAPI supports 10-bit full-frame SDI in both 525-line mode and 625-line mode. Please refer to section 9.3 for details about the representation of 10-bit SDI in DTAPI.

8.2. Using SDI-over-IP with DTAPI

The usage DTAPI to transmit or receive SDI-over-IP is relatively straightforward. The sections below provide code examples for initialising and configuring SDI-over-IP transmission and reception.

Before reception or transmission can begin, your application has to configure the SDI standard with the **`SetIpPars`** function. The **`m_VideoStandard`** member in class **`DtIpPars`** indicates the SDI standard to use. Each IP channel may use a different SDI standard.

NOTE: Native SDI channels (not over IP) use **`SetIoConfig`** to set the SDI standard. This I/O configuration mechanism is not supported for SDI-over-IP channels.

8.3. SDI Transmit

Figure 22 shows a code snippet for initializing and configuring an output channel for transmitting 10-bit 525-line SDI over IP to multicast IP address 239.1.1.1 with IP port 9999.

For correct operation, the size of the data written to the output channel must be a multiple of the SDI frame size.

```
DtDevice Dvc;
DtOutpChannel Outp;
DtIpPars IpPars;
DTAPI_RESULT dr;

// Attach to a DTA-2162
dr = Dvc.AttachToType(2162);
if (dr != DTAPI_OK)
  exit(dr);

// Attach to GigE port 1
dr = Outp.AttachToPort(&Dvc, 1);
if (dr != DTAPI_OK)
  exit(dr);

// Set the transmit mode to 10-bit SDI full-frame mode
dr = Outp.SetTxMode(DTAPI_TXMODE_SDI_10B | DTAPI_TXMODE_SDI_FULL, 0);
if (dr != DTAPI_OK)
  exit(dr);

// Initialise the IP parameters
DtapiInitDtIpParsFromIpString(IpPars, "239.1.1.1", NULL);
IpPars.m_DstPort = 9999;
IpPars.m_Protocol = DTAPI_PROTO_RTP;
IpPars.m_IpProfile.m_VideoStandard = DTAPI_VIDSTD_525I59_94;
dr = Outp.SetIpPars(&IpPars);
if (dr != DTAPI_OK)
  exit(dr);

// At this point the IP output channel is initialised.
// We can now set the output channel TxControl to HOLD and SEND.
// Writing SDI frames to the output buffer can be achieved with the
// code snippet described in section 3.5 of this document.
```

Figure 22. Code snippet to initialize transmission of 10-bit SD-SDI over IP

8.4. SDI Receive

Figure 23 shows a code snippet for initializing and configuring an input channel for receiving 10-bit 525-line SDI format over IP from multicast IP address 239.1.1.1 and IP port 9999.

The input channel supports the ReadFrame function that returns a complete SDI frame from the input buffer. See the "DTAPI Reference – Core Classes" for the details of this function.

```
DtDevice Dvc;
DtInpChannel Inp;
DtIpPars IpPars;
DTAPI_RESULT dr;

// Attach to a DTA-2162
dr = Dvc.AttachToType(2162);
if (dr != DTAPI_OK)
  exit(dr);

// Attach to GigE port 2
dr = Outp.AttachToPort(&Dvc, 2);
if (dr != DTAPI_OK)
  exit(dr);

// Set the receive mode to 10-bit SDI full-frame mode
dr = Outp.SetRxMode(DTAPI_RXMODE_SDI_10B | DTAPI_RXMODE_SDI_FULL);
if (dr != DTAPI_OK)
  exit(dr);

// Initialise the IP parameters
DtapiInitDtIpParsFromIpString(IpPars, "239.1.1.1", NULL);
IpPars.m_DstPort = 9999;
IpPars.m_Protocol = DTAPI_PROTO_RTP;
IpPars.m_IpProfile.m_VideoStandard = DTAPI_VIDSTD_525I59_94;
dr = Inp.SetIpPars(&IpPars);
if (dr != DTAPI_OK)
  exit(dr);

// At this point the IP input channel is initialised.
// We can now set RxControl to RCV and wait for the SDI data.
// See section 3.4 for details. Instead of the Read function, we can use
// the ReadFrame function to read the complete SDI frame at once.
```

Figure 23. Code snippet to initialize reception of 10-bit SD-SDI over IP

9. Definition of data formats

This section provides details about the different data formats used by DTAPI for transmitting and receiving data.

9.1. Generic Stream Encapsulation (GSE) Packet

GSE provides means to carry packet oriented protocols such as IP on physical layers such as DVB-T2 and DVB-C2, see ETSI TS 102 606. The multi-PLP modulator API and the advanced demodulator API support GSE-packets. The GSE-packets have a fixed size header followed by a variable size extension header and data part.

Syntax	#bits	Mnemonic
GsePacket() {		
protocol_type	**16**	**uimsbf**
if (GseLabelType == **NONE**) {		
reserved	**48**	**bslbf**
} else if (GseLabelType == **3BYTE**) {		
label_3byte	**24**	**bslbf**
reserved	**24**	**bslbf**
} else if (GseLabelType == **6BYTE**) {		
label_6byte	**48**	**bslbf**
}		
for (i=0; i<N1; i++)		
extension_header_byte	**8**	**bslbf**
for (i=0; i<N2; i++)		
data_byte	**8**	**bslbf**
}		

protocol_type
Protocol type carried in the packet, 16-bit field (2 bytes network order). Refer to IETF RFC 4326: "Unidirectional Lightweight Encapsulation (ULE) for Transmission of IP Datagrams over an MPEG-2 Transport Stream (TS)" for details.

label_3byte, label_6byte
Label used for addressing, 0, 3 or 6 bytes. For the modulator the address length is specified by *m_GseLabelType*. The total length of the label and the reserved bits is 6 byte.

extension_header_byte
Optional extension header bytes, the format depends on protocol type and is defined by the ULE specification IETF RFC 4326.

data_byte
Packet data byte.

9.2. L.3 Baseband Frame

L.3 Baseband frames are generated by an input channel if the **DTAPI_RXMODE_STL3** receive-mode is used (see DTAPI Reference – Core Classes function **DtInpChannel::SetRxMode**). The L.3 Baseband

frames are also used for DVB-S.2(X) L.3 baseband frame modulation (when the `DtOutpChannel::SetModControl` is set to `DTAPI_MOD_DVBS2_L3` or `DTAPI_MOD_DVBS2X_L3`).

Refer to the SatLabs L.3 document and for more details on the L3 fields.

Syntax	**#bits**	**Mnemonic**
L3_frame() {		
if (timestamp_flag) {		
TimeStamp[7..0]	**8**	**uimsbf**
TimeStamp[15..8]	**8**	**uimsbf**
TimeStamp[23..16]	**8**	**uimsbf**
TimeStamp[31..24]	**8**	**uimsbf**
}		
L3Sync /* 0xB8 */	**8**	**uimsbf**
AcmCommand	**8**	**uimsbf**
if (AcmCommand points to DVB-S2X VL-SNR) {		
AcmCommand2	**8**	**uimsbf**
}		
CNI	**8**	**uimsbf**
PlFrameId	**8**	**uimsbf**
BBHEADER() {		
MaType1	**8**	**uimsbf**
MaType2	**8**	**uimsbf**
Upl	**16**	**uimsbf**
Dfl	**16**	**uimsbf**
Sync	**8**	**uimsbf**
SyncD	**16**	**uimsbf**
Crc8	**8**	**uimsbf**
}		
for (i=0; i<n; i++)		
PayloadByte	**8**	**uimsbf**
}		

Notes:

- The *Timestamp* field is not transmitted on the ASI output port when looping-through L.3 baseband frames to the ASI output port on the DTA-2137.
- The *Timestamp* field is not used for DVB-S.2(X) L.3 baseband frame modulation

TimeStamp

The timestamp is a 32-bit field that indicates the value of the system reference clock at the moment the first byte of the baseband frame was received.

L3Sync

Synchronisation word: fixed value 0xB8.

AcmCommand

MODCOD and frame type. The meaning of bits 7..1 depend on bit 0.

AcmCommand bit 0 == 0	
Bit 7..3	DVB-S2 MODCOD, see the MODCOD field in the DVB-S2 specification.
Bit 2	FECFRAME size (0 = normal: 64 800 bits; 1 = short: 16 200 bits)
Bit 1	Pilots configuration (0 = no pilots, 1 = pilots)
AcmCommand bit 0 == 1	
Bit 7..2	DVB-S2X MODCOD, see the MODCOD field in the DVB-S2X specification. Note that the PLS-code = DVB-S2X MODCOD * 2 + 128.
Bit 1	Pilots configuration (0 = no pilots, 1 = pilots)

Please note that the receiver firmware (DTA-2137) deletes dummy frames (MODCOD=0).

AcmCommand2

This field is only present for DVB-S2X MODCOD >=0 and < 2 (PLS-code >=128 and PLS-code <132).
Bit 3..0 Index pointing to the VL-SNR header sequence.
Bit 7..4 Reserved; set to 0

CNI

8-bit Carrier-to-Noise plus interference ratio. The *CNI* value is updated every 50ms. The resolution is 0.125dB per unit and the range is -1.0...30.75dB. The encoding is shown in the following table:

Value	Meaning
0x00	Receiver is not in lock, CNI is not available
0x01	-1.0 dB
0x02	-0.875 dB
. . .	...
0xFE	30.625 dB
0xFF	≥ 30.75 dB

PlFrameId

Modulo-256 frame counter generated by the demodulator. The counter is incremented for each baseband frame received by the demodulator. Dummy frames are deleted by the firmware.

BBHEADER()

The DVB-S2 BBHEADER. Please refer to the DVB-S2 specification for details.

MaType1, MaType2

Describes the input stream format, Mode Adaptation and transmission roll off.

Upl

User-packet length in bits, in the range 0...65535.

Dfl

Data-field length in bits, in the range 0...58112.

Sync
Copy of the user-packet sync byte (e.g. 0x47 for MPEG2 Transport Stream packets)

SyncD
Distance in bits from the beginning of the DATA FIELD and the first UP from this frame (first bit of the CRC-8).

Crc8
Error detection code applied to the first 9 bytes of the BBHEADER.

PayloadByte
The payload of the baseband frame.

9.3. SDI – 10 bit

In 10-bit SDI format, all 10 bits of the SDI samples are stored. The first sample is the EAV code of the first line of a frame. The first line of a frame is considered to be the first line in which the Field bit in the EAV code is '0', indicating the first field: line 1 in 625-line mode or line 4 in 525-line mode. The first sample of a frame is always stored on a 32-bit boundary. Data stuffing of three bytes is needed in 525-line video mode, since the number of bytes in such a 10-bit frame is not a multiple of four.

Syntax	#bits	Mnemonic
sdi_10bit_stream() {		
if (timestamp_flag) {		
timestamp[7..0]	**8**	**uimsbf**
timestamp[15..8]	**8**	**uimsbf**
timestamp[23..16]	**8**	**uimsbf**
timestamp[31..24]	**8**	**uimsbf**
}		
do {		
for (line=1; line <= num_lines; line++) {		
sync_code /* '1111 1111 11' */	**10**	**bsrtlb**
sync_code /* '0000 0000 00' */	**10**	**bsrtlb**
sync_code /* '0000 0000 00' */	**10**	**bsrtlb**
eav_code(line)	**10**	**bsrtlb**
for (samp=1; samp<=hsyncs_per_line; samp++)		
sample_data	**10**	**bsrtlb**
sync_code /* '1111 1111 11' */	**10**	**bsrtlb**
sync_code /* '0000 0000 00' */	**10**	**bsrtlb**
sync_code /* '0000 0000 00' */	**10**	**bsrtlb**
sav_code(line)	**10**	**bsrtlb**
for (samp=1; samp<=samps_per_line; samp++)		
sample_data	**10**	**bsrtlb**
}		
if (sdi_std==Mode525) {		
for (i=0; i<3; i++)		
stuffing_byte /* '0000 0000' */	**8**	**bslbf**
}		
}		
}		

timestamp
The value of the reference clock at the moment the first SDI sample of the payload enters the input channel.

sync_code
Synchronisation byte as defined in the BT-656 specification.

eav_code
End of Active Video (EAV) code as defined in the BT-656 specification. The line number is encoded in EAV.

sav_code
Start of Active Video code as defined in the BT-656 specification. The line number is encoded in SAV.

sample_data
The 10-bit SDI samples.

stuffing_byte
Byte that is produced at the end of a 525-line mode frame only, with the purpose of aligning the first sample of the next frame to a 32-bit boundary

9.4. SDI – 8 bit

In 8-bit SDI format, only the most significant eight bits of each SDI sample are stored. The first sample is the EAV code of the first line of a frame. The first line of a frame is considered to be the first line in which the Field bit in the EAV code is '0', indicating the first field: line 1 in 625-line mode or line 4 in

525-line mode. The first sample of a frame is always stored on a 32-bit boundary. No data stuffing is required since in all modes the number of bytes in an 8-bit frame is divisible by four.

Syntax	#bits	Mnemonic
sdi_8bit_stream() {		
if (timestamp_flag) {		
timestamp[7..0]	**8**	**uimsbf**
timestamp[15..8]	**8**	**uimsbf**
timestamp[23..16]	**8**	**uimsbf**
timestamp[31..24]	**8**	**uimsbf**
}		
do {		
for (line=1; line <= num_lines; line++) {		
sync_code /* '1111 1111' */	**8**	**bslbf**
sync_code /* '0000 0000' */	**8**	**bslbf**
sync_code /* '0000 0000' */	**8**	**bslbf**
eav_code(line)	**8**	**bslbf**
for (samp=1; samp<=hsyncs_per_line; samp++)		
sample_byte	**8**	**bslbf**
sync_code /* '1111 1111' */	**8**	**bslbf**
sync_code /* '0000 0000' */	**8**	**bslbf**
sync_code /* '0000 0000' */	**8**	**bslbf**
sav_code(line)	**8**	**bslbf**
for (samp=1; samp<=samps_per_line; samp++)		
sample_byte	**8**	**bslbf**
}		
}		
}		

timestamp
The value of the reference clock at the moment the first SDI sample of the payload enters the input channel.

sync_code
Synchronisation byte as defined in the BT-656 specification.

eav_code
End of Active Video (EAV) code as defined in the BT-656 specification. The line number is encoded in EAV.

sav_code
End of Active Video (EAV) code as defined in the BT-656 specification. The line number is encoded in EAV.

sample_byte
The SDI video data with the two least significant bits removed.

9.5. SDI – Huffman-Compressed

Some of DekTec's SDI devices support a custom Huffman encoding scheme for compressing of SDI frames. You can detect the support for this feature by using the DTAPI_CAP_HUFFMAN capability flag. Using the compressed format can be useful to reduce the size of recorded SDI files. The using compression can also be used to reduce PCI or USB bandwidth requirements.

The table below provides the syntax of a compressed SDI frame.

Syntax	#bits	Mnemonic
sdi_compressed_stream_with_blanking () {		
if (timestamp_flag) {		
timestamp[7..0]	**8**	**uimsbf**
timestamp[15..8]	**8**	**uimsbf**
timestamp[23..16]	**8**	**uimsbf**
timestamp[31..24]	**8**	**uimsbf**
}		
do {		
sync_word /* '11 1111 1111 1111 1111' */	**18**	**bsrtlb**
for (line=1; line <= num_lines; line++) {		
skip_samples(4); /* skip EAV */		
prev_data = blanking_level		
for (samp=1; samp<=hsyncs_per_line; samp++) {		
huffman(**sample_data** - **prev_data**)	**2-16**	**bsrtlb**
prev_data = **sample_data**		
}		
skip_samples(4); /* skip SAV */		
prev_data = blanking_level		
for (samp=1; samp<=samps_per_line; samp++) {		
huffman(**sample_data** - **prev_data**)	**2-16**	**bsrtlb**
prev_data = **sample_data**		
}		
}		
if (alignment()!=32)		
stuffing_data /* '0' */	**2-30**	**bsrtlb**
}		
}		

timestamp
The value of the reference clock at the moment the first SDI sample of the payload enters the input channel.

sync_word
Synchronisation code consisting of 18 consecutive '1's.

sample_data
The SDI video data.

prev_data
The previous sample of the SDI video data of the same type (Cb, Y, or Cr) as the current sample.

stuffing_data
Data that is produced at the end of a frame only, with the purpose of aligning the **sync_word** of the next frame to a 32-bit boundary

The table below provides the syntax of a compressed frame with only the active video part.

Syntax	#bits	Mnemonic
sdi_compressed_stream_with_blanking () {		
if (timestamp_flag) {		
timestamp[7..0]	**8**	**uimsbf**
timestamp[15..8]	**8**	**uimsbf**
timestamp[23..16]	**8**	**uimsbf**
timestamp[31..24]	**8**	**uimsbf**
}		
do {		
sync_word /* '11 1111 1111 1111 1111' */	**18**	**bsrtlb**
for (line=1; line <= num_lines; line++) {		
skip_samples(4); /* skip EAV */		
prev_data = blanking_level		
for (samp=1; samp<=hsyncs_per_line; samp++) {		
huffman(**sample_data** - **prev_data**)	**2-16**	**bsrtlb**
prev_data = **sample_data**		
}		
skip_samples(4); /* skip SAV */		
prev_data = blanking_level		
for (samp=1; samp<=samps_per_line; samp++) {		
huffman(**sample_data** - **prev_data**)	**2-16**	**bsrtlb**
prev_data = **sample_data**		
}		
}		
if (alignment()!=32) {		
stuffing_data /* '0' */	**2-30**	**bsrtlb**
}		
}		
}		

timestamp
The value of the reference clock at the moment the first SDI sample of the payload enters the input channel.

sync_word
Synchronisation code consisting of 18 consecutive '1's.

sample_data
The SDI video data.

prev_data
The previous sample of the SDI video data of the same type (Cb, Y, or Cr) as the current sample.

stuffing_data
Data that is produced at the end of a frame only, with the purpose of aligning the **`sync_word`** of the next frame to a 32-bit boundary

See DTAPI Reference – Core Classes function **`DtSdi::ConvertFrame`** for conversion between the compressed Huffman format and one of the uncompressed SDI formats.

9.6. Transparent Mode

Transparent mode adds an extra packetization layer to combine a TS-packet-oriented mode and raw mode. If the input data contains valid TS packets, each "transparent packet" contains exactly one TS packet, an optional time stamp and the in-sync flag is set. If the input data is out of sync, the transparent packet contains the raw input data and the in-sync flag is cleared.

Transparent mode is selected by setting the receive mode to **`DTAPI_RXMODE_STTRP`**. Transport-stream monitoring applications can use this mode to receive time-stamped packets for jitter analysis, while sync errors can still be detected.

Syntax	#bits	Mnemonic
transparent_packet() {		
if (timestamp_flag) {		
timestamp[7..0]	**8**	**uimsbf**
timestamp[15..8]	**8**	**uimsbf**
timestamp[23..16]	**8**	**uimsbf**
timestamp[31..24]	**8**	**uimsbf**
}		
for (i=0; i<204; i++)		
payload_byte	**8**	**bslbf**
sync_nibble /* '0101' */	**4**	**bslbf**
packet_sync	**1**	**bslbf**
reserved	**3**	**bslbf**
valid_count	**8**	**uimsbf**
sequence_count[7..0]	**8**	**uimsbf**
sequence_count[15..8]	**8**	**uimsbf**
}		

timestamp
The value of the reference clock at the moment the first byte of the data is received.

payload_byte
Payload of the transparent packet containing the received data, which is either a TS packet or raw data. The number of valid bytes in the payload is indicated by the *valid_count* field. When *packet_sync* is '1' the first payload byte will usually be 47h, but not necessarily! This is because an incidental error in the sync byte will not cause loss of synchronisation.

sync_nibble
The *sync_nibble* is a fixed 4-bit field whose value is '0101' (5). Applications can use this nibble to synchronize to transparent packets.

packet_sync
When set to '1' this flag indicates that synchronisation to TS packets has been achieved.

reserved
These bits are reserved for future use.

valid_count
This field indicates the number of valid bytes in the payload of the transparent packet. If the *packet_sync* flag is set this field will be either 188 or 204. If the *packet_sync* flag is not set the value can be anything between 1 and 204.
If the number of valid bytes is less than 204, then the value of the remaining payload bytes is undefined.

sequence_count

The *sequence_count* is a 16-bit field that contains the original sequence number of the packet in the Transport Stream. The value of the sequence counter is only meaningful if *packet_sync* is '1'. Without PID filtering, *sequence_count* will be incremented by 1 for each received packet. When PID filtering is used, *sequence_count* can be used to determine the number of packets that has been skipped.

9.7. Transmit on Timestamp

The transmit-on-timestamp mode is used to transmit transport packets at user-defined timestamps. Details of the transmit-on-timestamp mode are described in DTAPI Reference – Core Classes.

Syntax	#bits	Mnemonic
txontime_stream() {		
do {		
timestamp[7..0]	**8**	**uimsbf**
timestamp[15..8]	**8**	**uimsbf**
timestamp[23..16]	**8**	**uimsbf**
timestamp[31..24]	**8**	**uimsbf**
if (TxMode == **188** || TxMode == **Add16**)		
for (i=0; i<188; i++)		
tp_byte	**8**	**bslbf**
if (TxMode == **204** || TxMode == **Min16**)		
for (i=0; i<204; i++)		
tp_byte	**8**	**bslbf**
}		
}		

timestamp

Relative time at which to transmit the packet. The timestamp is encoded in four bytes in little-endian format.

tp_byte

Byte in a transport packet.