Nutaq® INNOVATION TODAY FOR TOMORROW®

RTDExSync

Programmer's Reference Guide

February 2016

Revision history

Revision	Date	Comments
1.0	March 2015	Firs release of the document
1.1	October 2015	Added Host Applications Using the RTDExSync EAPI Library Up to date for release 7.0
1.2	February 2016	Up to date for PicoSDR official release

To ensure the accuracy of the information contained herein, particular attention was given to usage in preparing this document. It corresponds to the product version manufactured prior to the date appearing on the title page. There may be differences between the document and the product, if the product was modified after the production of the document.

Nutaq reserves itself the right to make changes and improvements to the product described in this document at any time and without notice.

Version 1.2

Trademarks

WARNING

Do not use Nutaq products in conjunction with life-monitoring or life-critical equipment. Failure to observe this warning relieves Nutaq of any and all responsibility.

FCC WARNING

This equipment is intended for use in a controlled environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of personal computers and peripherals pursuant to subpart J of part 15 of the FCC rules. These rules are designed to provide reasonable protection against radio frequency interference. Operating this equipment in other environments may cause interference with radio communications, in which case the user must, at his/her expense, take whatever measures are required to correct this interference.

Table of Contents

List of Figures and Tables

1 Introduction

This document contains all the information necessary to understand and use the RTDExSync module with your product. It should be read carefully before using the card and stored in a handy location for future reference.

1.1 Conventions

In a procedure containing several steps, the operations that the user has to execute are numbered (1, 2, 3…). The diamond (♦) is used to indicate a procedure containing only one step, or secondary steps. Lowercase letters (a, b, c…) can also be used to indicate secondary steps in a complex procedure.

The abbreviation NC is used to indicate no connection.

Capitals are used to identify any term marked as is on an instrument, such as the names of connectors, buttons, indicator lights, etc. Capitals are also used to identify key names of the computer keyboard.

All terms used in software, such as the names of menus, commands, dialog boxes, text boxes, and options, are presented in bold font style.

The abbreviation N/A is used to indicate something that is not applicable or not available at the time of press.

Note:
The screen captures in this document are taken from the software version available at the time of press. For this reason, they may differ slightly from what appears on your screen, depending on the software version that you are using. Furthermore, the screen captures may differ from what appears on your screen if you use different appearance settings.

1.2 Glossary

This section presents a list of terms used throughout this document and their definition.

Term	Definition
Application programming interface (API)	An application programming interface is the interface that a computer system, library, or application provides to allow requests for services to be made of it by other computer programs or to allow data to be exchanged between them.
Board software development kit	Abbreviated BSDK, this kit gives users the possibility to quickly become fully functional developing C/C++ for the host computer and HDL code for the FPGA through an understanding of all Nutaq boards major interfaces.
Chassis	Refers to the rigid framework onto which the CPU board, Nutaq development platforms, and other equipment are mounted. It also supports the shell-like case—the housing that protects all the vital internal equipment from dust, moisture, and tampering.
Default design	Design loaded by default on Nutaq boards used for FPGA design.
Digital signal processing	Digital signal processing is the study of signals in a digital representation and the processing methods of these signals. The algorithms required for DSP are sometimes performed using specialized devices that use specialized microprocessors called digital signal processors (DSP).
Example	Refers to examples used to demonstrate functions or applications supplied with the board software development kit. For this reason, examples come in two flavors: application examples and functional examples.
FIFO	First in first out memory.
HDL	Stands for hardware description language.
Host	A host is defined as the device that configures and controls a Nutaq board. The host may be a standard computer or an embedded CPU board in the same chassis system where the Nutaq board is installed. You can develop applications on the host for Nutaq boards through the use of an application programming interface (API) that comprises protocols and functions necessary to build software applications. These API are supplied with the Nutaq board.
Model-based design	Refers to all the Nutaq board-specific tools and software used for development with the boards in MATLAB and Simulink and the Nutaq model-based design kits.
Peer	A host peer is an associated host running RTDEx on either Linux or Windows. An FPGA peer is an associated FPGA device.
PPS	Pulse per second. Event to indicate the start of a new second.
Reception	Any data received by the referent is a reception. Abbreviated RX.
Reference design	Blueprint of an FPGA system implanted on Nutaq boards. It is intended for others to copy and contains the essential elements of a working system (in other words, it is capable of data processing), but third parties may enhance or modify the design as necessary.
RTDEx	Real-time data exchange. Nutaq's library and module available to transfer raw data between a host computer and a platform.
Software development	Refers to development performed with and for the board with a software development kit. Software development for a board comes in three flavors: host software development and FPGA software development.
Transmission	Any data transmitted by the referent is a transmission. Abbreviated TX.
VHDL	Stands for VHSIC hardware description language.

Table 1 Glossary

1.3 Technical Support

Nutaq is firmly committed to providing the highest level of customer service and product support. If you experience any difficulties using our products or if it fails to operate as described, first refer to the documentation accompanying the product. If you find yourself still in need of assistance, visit the technical support page in the Support section of our Web site at www.nutaq.com.

2 Product Description

Nutaq's RTDExSync core adds synchronous functionalities to the Gigabit Ethernet and PCIe real-time data exchange (RTDEx) data flows. The present RTDEx implementation allows a host computer to transmit and receive data to/from hardware platforms. The data can contain samples going to DACs or coming from ADCs.

The synchronous implementation of the RTDEx gives the user control over the start and stop conditions of an RTDEx transfer. The conditions that can be used to control such transfers are: immediately, after a rising-edge trigger is detected or at a specific time. These start and stop conditions can be used to synchronize the uplink and downlink channels together or to synchronize multiple channels together even across multiple hardware platforms. The RTDExSync functionalities also add a synchronization recovery mechanism so different channels can remain synchronized even when samples are lost during a transfer.

The RTDExSync module is intended to be used to synchronize the transmission and reception of samples through many hardware platform boards for MIMO and Massive MIMO systems.

2.1 Outstanding Features

Start and stop transfers immediately

The RTDExSync module can start and stop a transfer as soon as the command is received by the hardware platform. Even if this mode looks like the default RTDEx functionality, the synchronization recovery mechanism is active when using RTDExSync in this mode. Furthermore, this mode can be used in combination with any other RTDExSync mode.

Start and stop transfers on trigger events

The RTDExSync module enables the possibility to start and stop transfers upon reception of a rising-edge trigger. Four global trigger inputs are shared between the eight RTDEx channels. Each channel start or stop condition can use any of those four trigger inputs.

The triggers are made available in the user logic and can be connected to any signal. They can be directly connected to the input trigger signal of an FMC, a backplane GPIO or any data processing result such as a signal power threshold.

Start and stop transfers at a specific time

The RTDExSync module enables the possibility to start and stop transfers upon reaching a certain time. The time at which an event can occur has a sample-level precision. The RTDExSync module receives the system time from another module. The time is expressed in a number of ticks of the clock used by the time generator module.

The **Timestamp** module provides the time to the **RTDExSync** module. The user clock of the **Timestamp** and the **RTDExSync** modules must be the same.

Synchronization recovery mechanism

In both directions, if some data samples are lost or are not available when required, the missing data will be replaced by zeros to provide a continuous data flow. As soon as some up-to-date data are available, the samples will be outputted the same way as if no data has been lost.

This mechanism makes it possible to reliably base the sample time on the start time and the number of samples received or transmitted. It is also intended to keep the synchronization across multiple channels even if samples losses occur on some of them.

2.2 Usage Overview

The **RTDExSync** has been designed to transfer 32-bit (4-byte) samples. The **RTDExSync** FPGA core user data ports are 32-bit wide and the **RTDExSync** host API functions express the number of samples to transfer or that have been transferred in number of 32-bit samples. For the transmission of complex I/Q samples, I samples can be expressed in 16-bit format and located in the least significant bits (LSB) while Q samples can be expressed in 16-bit format and located in the most significant bits (MSB).

The **RTDExSync** FPGA core user ports provide a FIFO-like interface for each channel. When a channel is not started, its **Ready** output port is low to inform the user that no data can be written or read to/from it.

When the channel state is **Idle**, the host computer can send a packet with a start condition. On every state change, a context packet is sent to the host to indicate the new channel state as well as some contextual information. When the channel receives the start condition, a message will be sent to the host indicating a change of state to **Start Pending**.

Once the start condition is met, a message is sent to the host to indicate a change of state to **Running**, the channel **Ready** port will become high and the data can be written or read at the same clock cycle.

To maintain the synchronization, it is important to never write or read data when the channel is not started, i.e. when the **Ready** flag is low. Once started, the **Ready** flag will always be high since, in downlink, zeros are inserted if new samples are not available and, in uplink, samples are discarded if the samples cannot be sent quickly enough. In uplink, it is the host computer that inserts zeros when data discontinuities are detected.

When the channel is in **Start Pending** or in **Running** state, the host computer can send a packet containing the stop condition. If the stop condition is received and the channel is in **Running** state, the state will change to **Stop Pending** and the corresponding message will be sent to the host computer.

Once the stop condition is met, the **Ready** port is set low to indicate that the transfer is over. The channel state is set to **Idle** and the corresponding message is sent to the host.

Once the state is back to **Idle**, a new transfer can be performed by sending the desired start condition.

3 RTDExSync FPGA Cores Description

3.1 RTDExSync FPGA Core Overview

The

Figure 1 below provides a block diagram of the interaction between the RTDExSync, the existing RTDEx cores and an FMC containing an ADC and a DAC. The **RTDExSync** module is intended to be used on top on the existing **RTDEx** module. The **RTDExSync** is directly connected to the **RTDEx Gigabit Ethernet** (GigE) or the **RTDEx PCIe** user FIFO interfaces so it has access to a raw data transfer interface with a host computer. Instead of streaming raw data, the **RTDExSync** module creates and decodes packets containing contextual information and data samples. The packet encoder and decoder are based on the VITA 49 standard.

The **Timestamp** module can be used to provide the time base to the **RTDExSync** module. The time is expressed in clock ticks.

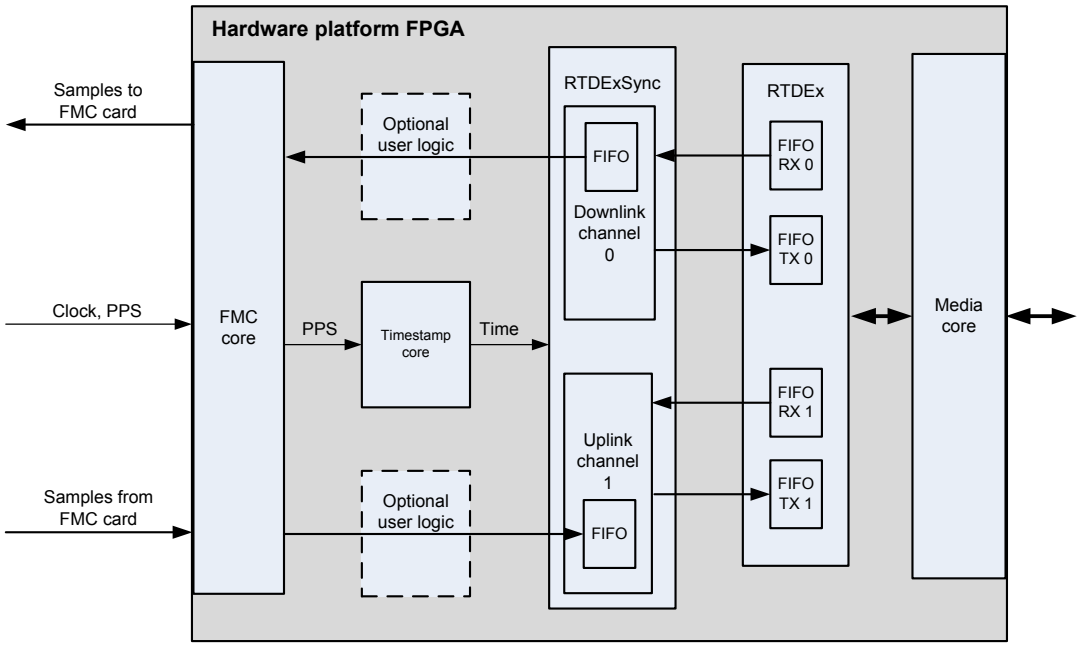

Figure 1 RTDExSync interaction with RTDEx

Figure 2 presents a block diagram of the **RTDExSync** core showing one channel configured in each direction. The **RTDExSync** core can handle up to eight channels, each configured in the downlink or the uplink direction. In this example, the downlink decoder uses the RTDEx RX channel 0 to decode data packets received from the host while the downlink encoder uses the RTDEx TX channel 0 to encode context packets to send to the host. In uplink direction, the uplink decoder uses the RTDEx Rx channel 1 to decode data packets containing the control information destined the RTDExSync core, such as start and stop commands. The uplink encoder uses the RTDEx TX channel 1 to encode data and context packets destined to the host.

The time gating and trigger detection logic is performed by the downlink/uplink finite-state machines (FSM). These logic blocks control the moment when the FIFOs on enabled on the user side.

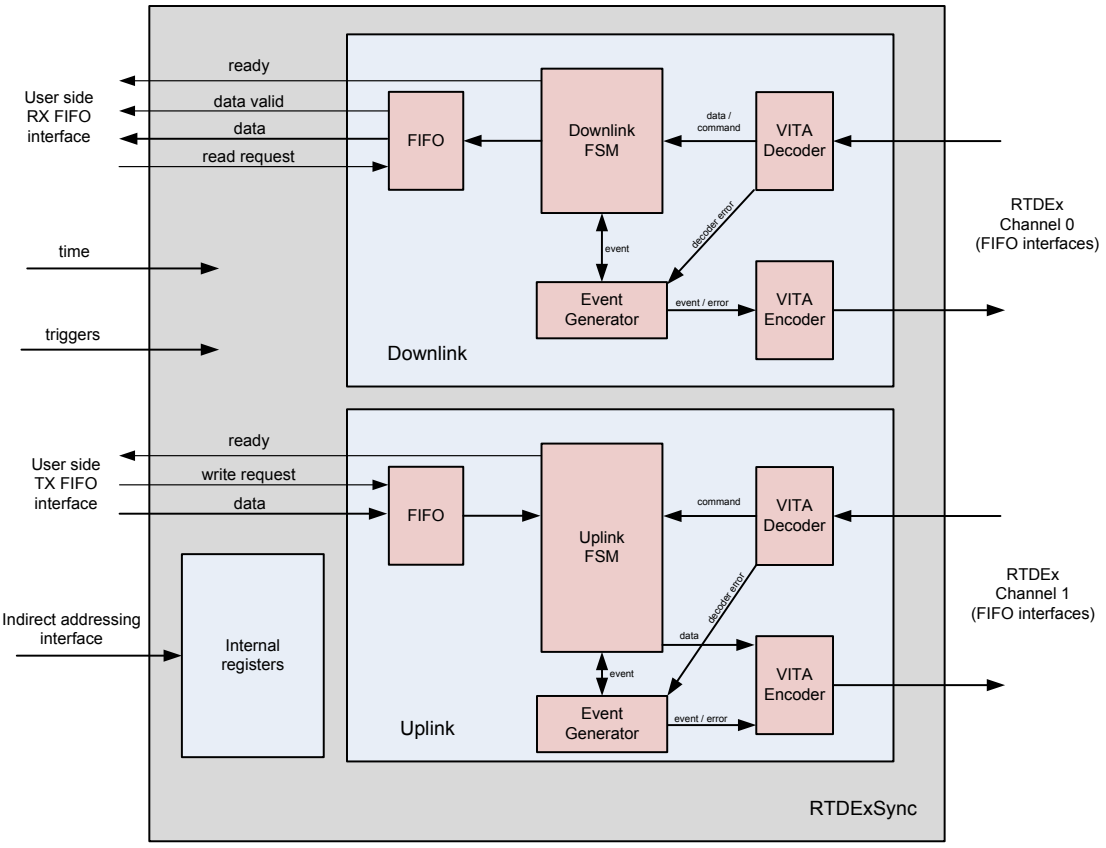

Figure 2 RTDExSync block diagram

3.2 RTDExSync VITA Packet Decoder

The decoder decodes VITA 49 VRL frames and VRT data packets. Neither context packets nor extended context packets are actually sent from the host to the FPGA. All control is passed through VRT data packets.

The FIFO interface of the decoder is compatible with the current RTDEx interface.

From the data packet, the start and stop command are extracted from the packets as well as the data samples. The decoder propagates with the data samples, the associated sample count to the FSM in order to maintain the synchronization of the data stream.

3.3 RTDExSync VITA Packet Encoder

The VITA 49 packet encoder encodes VRT data, context and extended context packets. The encoder prioritizes the extended context packet first, then the context packet and finally the data packet in case many packets must be encoded and transmitted. The current implementation supports one VRT packet per VRL frame. For context and extended context packets, the VRL frames are padded to fit to the RTDEx frame size. The size of the VRT data packets is set to fit the RTDEx frame size without padding.

The FIFO interface of the encoder is compatible with the current RTDEx interface.

The encoder has two different frame size parameters. There is one for the data packet and there is another one for the context and extended context packets. The data packet size can be set high in order to minimize the header overhead compared to the data packet. The context packet size can be set low in order to avoid sending more than one RTDEx packet when it is not necessary.

3.4 Downlink Finite State Machine (FSM)

This module has two main tasks:

- Process the commands received from the host and gate the FIFO in downlink accordingly.
- Maintain the ordering of the samples during active transmissions.

Gating FIFO and processing received commands

Figure 3 shows a diagram representing the states of the downlink FSM. These states correspond to the 2-bit channel state field contained in context packets. A context packet is sent to the host every time the FSM changes state. The downlink FIFO can be read by the user when the state changes from **Start Pending** to **Running**, and is deactivated when it changes from **Stop Pending** to **Idle**.

When the FSM state is set back to the **Idle** state, the samples present in the FIFO memory are cleared to make sure no data from a previous transmission will be transmitted in the next one.

If the time specified for a start or a stop condition is in the past at the moment a command is processed, an extended context packet is sent to the host to indicate a miss-timed command. When a miss-timed command is detected, the FSM state is set directly to **Idle** state.

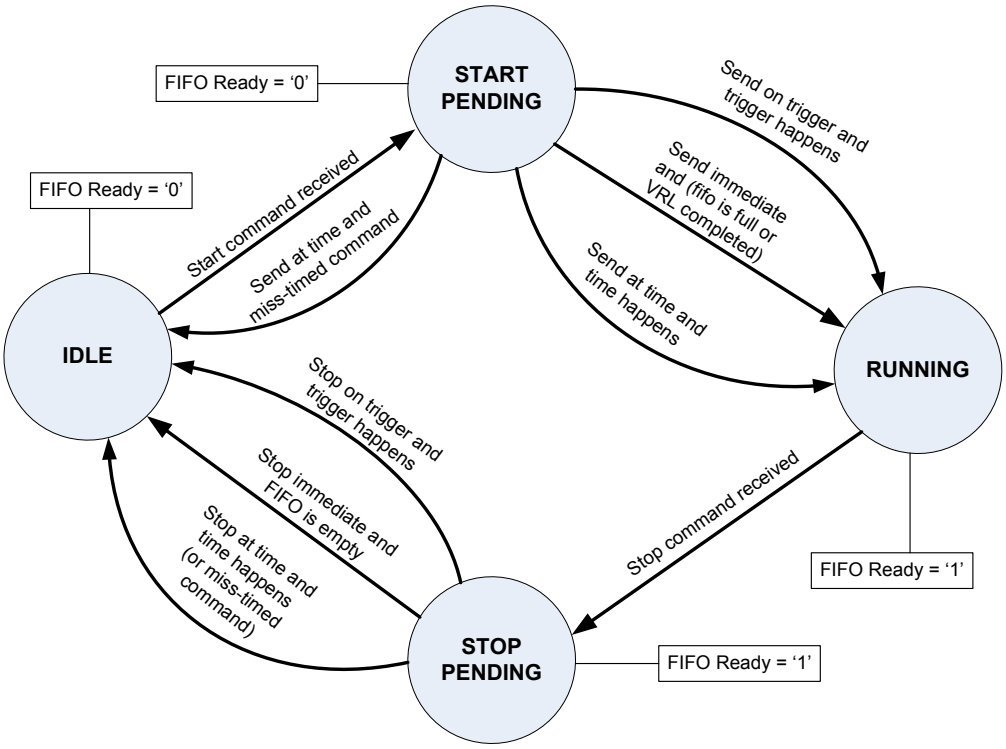

Figure 3 Downlink FSM

Samples ordering during an active transfer

During a transmission, this module has to feed the FIFO and make sure it never gets empty. It increments a local sample count every time a sample is written to the FIFO.

- The received samples from the VITA decoder are paired with a sample count. When this sample count matches the local sample count, the samples are transferred to the downlink FIFO. If not, they are discarded until the counts match.
- In the meantime, this module must make sure the FIFO never empties. If samples from the decoder with matching sample counts are available, the module transfers them, if not, when the FIFO is almost empty, the module feeds the FIFO with zeros and increments its internal counter to keep the synchronization. When this happens, an underrun event is generated and is sent to the host.

  When data samples with valid sample counts are received after an underrun occurred, a resume event is sent to the host to indicate that the synchronization has been recovered.

3.5 Uplink FSM

This module has two main tasks:

- Process the commands received from the host and gate the FIFO in uplink accordingly.
- Maintain the ordering of the samples during active transmissions.

Gating FIFO and processing received commands

Figure 4 shows a diagram representing the states of the uplink FSM. These states correspond to the 2-bit channel state field contained in context packets. A context packet is sent to the host each time the FSM changes state. The uplink FIFO can be written to by the user when the state changes from **Start Pending** to **Running**, and is deactivated when it changes from **Stop Pending** to **Idle**.

When the FSM changes from the **Idle** state to the **Start Pending** state, the samples present in the FIFO memory are cleared to make sure no data from a the previous transmission will be transmitted in the new one.

An extended context packet is sent to the host to indicate a miss-timed command when the time for the start and stop condition is in the past at the moment of the command reception. When a miss-timed command is detected, the FSM state is set directly to **Idle** state.

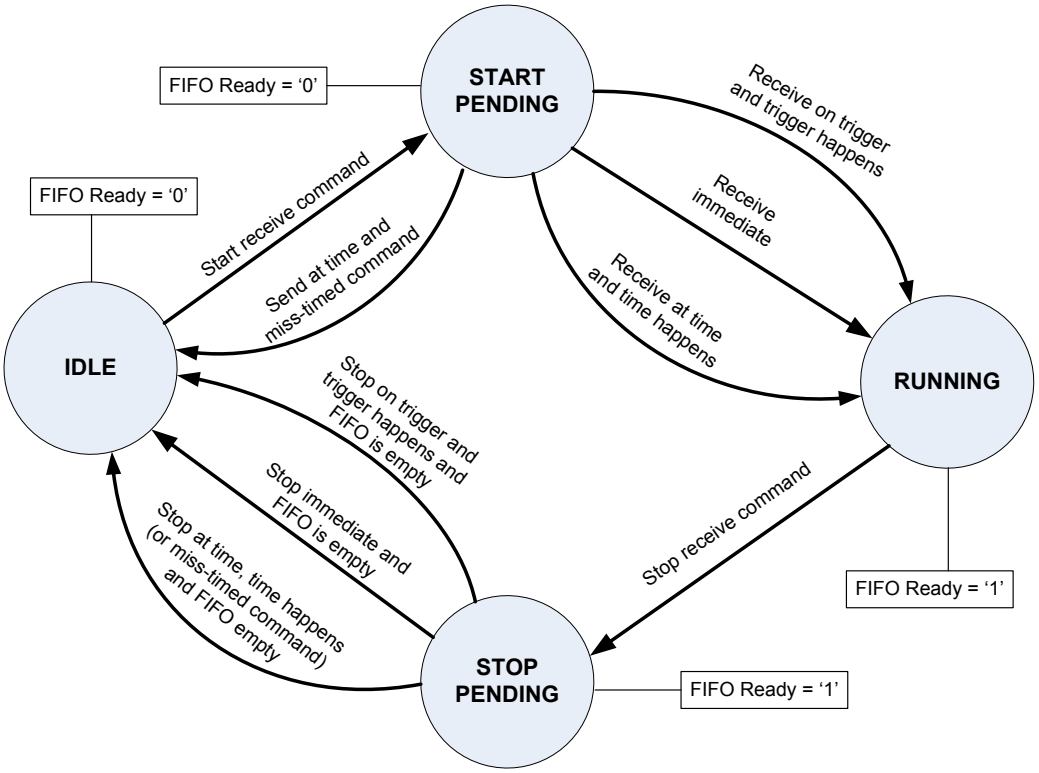

Figure 4 Uplink FSM

Sample ordering during active transfer

During a transmission, this module has to empty the FIFO and make sure it never gets full on the user side. It increments a local sample count every time a sample is read from the FIFO.

- When the VITA encoder is available, samples are read from the uplink FIFO along with their sample count and transferred to the RTDEx interface.
- In the meantime, this module must make sure the FIFO never gets full. When the FIFO is almost full, it reads samples even if the encoder is not ready. When this happens, the samples are discarded and an overrun status is generated.

  The host computer will know, from the sample count sent in a data packet, if some samples were discarded. It will then generate an overflow event for the user. When the user requests these missing samples, the receive function returns samples with a value of zero to keep the synchronization on the host user side even when samples are dropped. Once valid samples are received again, a resume event is generated and the data samples returned to the user will be meaningful again.

3.6 Event Generator

The event generator module is responsible for the generation of the events and errors contained in both context and extended context packets sent to the host computer.

3.6.1 Events

Four different events can be sent in a context packet:

- The state of the FSM has changed
- Zeros have been inserted into the downlink FIFO
- The **PLL locked** input port value has changed
- The **GPS locked** input port value has changed.

A timestamp value is included in each context packet. When the state of a FSM changes to the **Running** or the **Idle** state the timestamp value included in the context packet is precise. The timestamp value is imprecise, a few clock periods more or less, for all the other events. The reason the timestamp is imprecise for these other events is because it is not directly related to specific data samples in the user logic. When zeros are inserted into the FIFO, the event timestamp is related to the time the zeros are written inside the FIFO and not the time at which they are read and accessible in the user logic.

3.6.2 Errors

When errors happen, they are sent to the host inside an extended context packet. There are five different errors detected:

- Invalid command: The **Start** command can only be received when the FSM state is in the **Idle** state and the **Stop** command can only be received when the FSM is not in the **Idle** state.
- Invalid timestamp: The **Start at time** or **Stop at time** command refers to a timestamp value in the past compared to the moment of the reception of the command. When this happens, the FSM state is set back to **Idle**. In this miss-timed error packet, the reception time and the requested time are both included in the message. This can happen if the host time and the FPGA time are not synchronized or if the latency of the command transmission is high causing the time command to be out-dated at the moment it is received by the FPGA.

  To make sure a packet arrives at the RTDExSync core before the start condition time, the host needs to take into account the propagation delay from the host to the FPGA and the host processing time. It is important to note that the start and stop conditions are propagated along with the data packets. The start and stop conditions will not be processed until the decoder has decoded all the previous data packets.
- Invalid VRT packet: The VRT packet received is composed of invalid parameters.
- Invalid VRL frame: The VRL frame received is composed of invalid parameters.
- Synchronization lost: The first word of the VRL frame does not match the VRL header value.

3.7 Cores Ports

Port	Direction	Description
Clock and reset		
i_RTDExSyncCoreClk_p	IN STD_LOGIC	RTDExSync core clock. This clock must be higher than the user clock frequency in order to read and write new data at each clock cycle.
i_userClk_p	IN STD_LOGIC	Clock used at the FIFO user side.
i_CoreReset_p	IN STD_LOGIC	Active high core reset. Resets all the internal registers and channel states.
ov32_CoreIdVers_p	OUT STD_LOGIC_VECTOR(31 downto 0)	RTDExSync core ID and version. This port must be connected to the related input port of the platform register core. The RTDExSync core ID is 0xCC02.
External triggers		

Port	Direction	Description
iv4_ExtTrigger_p	IN STD_LOGIC_VECTOR(3 downto 0)	External triggers used for start and stop at trigger commands. The triggers are rising-edge sensitive and are shared across all channels.
Indirect access to internal register These registers must be connected to the platform register core		
i_IndWrEn_p	IN STD_LOGIC	Indirect addressing write enable
iv8_IndAddr_p	IN STD_LOGIC_VECTOR(7 downto 0)	Indirect addressing address
iv32_IndWrReg_p	IN STD_LOGIC_VECTOR(31 downto 0)	Indirect addressing write register
ov32_IndRdReg_p	OUT STD_LOGIC_VECTOR(31 downto 0)	Indirect addressing read register
System status Event message are sent to the host computer when the state of these inputs change.		
i_PllLocked_p	IN STD_LOGIC	PLL lock status.
i_GpsLocked_p	IN STD_LOGIC	PPS sync lock status.
Time		
iv64_FpgaTime_p	IN STD_LOGIC_VECTOR(63 downto 0)	Free-running timestamp. The time must be synchronous to the user clock.
RTDEx RX Interface These ports must be connected to a specific RX channel of the RTDEx core. These signals must be synchronous to the RTDExSync core clock and the RTDEx core RX user clock must be the same clock. Must be connected in uplink and in downlink direction.		
i_RTDExRxReadyCh*_p	IN STD_LOGIC	RTDEx receiver has valid data to read
o_RTDExRxReadReqCh*_p	OUT STD_LOGIC	RTDEx receiver reads FIFO request
i_RTDExRxDataValidCh*_p	IN STD_LOGIC	Data at RTDEx receiver data bus is valid
iv32_RTDExRxDataCh*_p	IN STD_LOGIC_VECTOR(31 downto 0)	RTDEx receiver output data bus
RTDEx TX Interface These ports must be connected to the same channel number of the RTDEx core than the RX interface but in the TX direction. These signals must be synchronous to the RTDExSync core clock and the RTDEx core TX user clock must be the same clock. Must be connected in uplink and in downlink direction.		
i_RTDExTxReadyCh*_p	IN STD_LOGIC	RTDEx transmitter ready
o_RTDExTxWriteReqCh*_p	OUT STD_LOGIC	RTDEx transmitter writes FIFO request
ov32_RTDExTxDataCh*_p	OUT STD_LOGIC_VECTOR(31 downto 0)	RTDEx transmitter data bus
RTDExSync RX Interface User interface to the data FIFO in downlink direction.		
o_RTDExSyncRxReadyCh*_p	OUT STD_LOGIC	RTDExSync receiver has valid data to read. This signal is high when the transmission is started and goes low when the transmission is over.
i_RTDExSyncRxReReqCh*_p	IN STD_LOGIC	RTDExSync receiver reads FIFO request. A read request must only be done when the ready signal is high.
o_RTDExRxSyncDataValidCh*_p	OUT STD_LOGIC	Data at RTDExSync receiver data bus is valid. The data is valid two clock cycles after a valid read request.
ov32_RTDExSyncRxDataCh*_p	OUT STD_LOGIC_VECTOR(31 downto 0)	RTDExSync receiver output data bus. This data is only valid when
RTDExSync TX Interface User interface to the data FIFO in uplink direction.		
o_RTDExSyncTxReadyCh*_p	OUT STD_LOGIC	RTDExSync transmitter ready. This signal is high when the transmission is started and goes low when the transmission is over.

Port	Direction	Description
i_RTDExSyncTxWriteReqCh*_p	IN STD_LOGIC	RTDExSync transmitter writes FIFO request. A write request must only be done when the ready signal is high.
iov32_RTDExSyncTxDataCh*_p	IN STD_LOGIC_VECTOR(31 downto 0)	RTDExSync transmitter data bus

Table 2 RTDExSync core ports

3.8 Core Registers (Indirect Addressing)

The core registers are accessible through indirect addressing. The indirect access register ports must be connected to the related platform register core ports.

To write a value to an internal register, set the address register and the write register with proper values, set the write enable high (1) and set it back to low (0).

To read the value of an internal register, set the address register and then read the read register.

The register index for a specific channel is located in the 3 LSB on the address port vector:

```
iv8_IndAddr_p(2 downto 0)
```

The channel index, from 0 to 7, can be selected with the following address bits:

```
iv8_IndAddr_p(6 downto 4)
```

Register name	Address	Direction	Description
RTDEXSYNC_CH_CTRL	ChIdx * 16 + 0x0	R/W	Channel Control Register
RTDEXSYNC_CH_RX_FRAME_SIZE	ChIdx * 16 + 0x1	R/W	RX channel frame size Register
RTDEXSYNC_CH_TX_DATA_FRAME_SIZE	ChIdx * 16 + 0x2	R/W	TX channel frame size register for data packet.
RESERVED	ChIdx * 16 + 0x3	R	Reserved
RTDEXSYNC_CH_TX_CNTX_FRAME_SIZE	ChIdx * 16 + 0x4	R/W	TX channel frame size register for context and extended context packet.

Table 3 RTDEx channel registers

Offset 0x0 — RTDEXSYNC_CH_CTRL

This register controls the RTDEx Sync channel.

31 to 4	3	2	1	0
Reserved	**Present**	**Direction**	**Enable**	**Reset**
-	R	R	R/W	R/W
-				

Table 4 RTDExSync channel control

RTDEXSYNC_CH_CTRL register bit description

Bit	Description	Configuration
Reset	Reset the channel state to initial.	0: None 1: Active high reset
Enable	Enable the channel operation.	0 = Disabled 1 = Enabled

Bit	Description	Configuration
Direction	Get the current channel direction.	0 = Up link 1 = Down link
Present	Get the presence status of the channel.	0: RTDEx Sync core is bypassed for this channel 1: RTDEx Sync core is present for this channel

Table 5 RTDExSync VRT payload format register bit description

Offset 0x1 — RTDEXSYNC_CH_RX_FRAME_SIZE

This register specifies the RTDEx frame size used for the receive path (host to FPGA). The frame size must be an integer multiple of the RTDEx transfer size.

31 to 24	23 to 0
Reserved	**RTDExSyncRxFrameSize**
-	R/W
-	-

Table 6 RTDExSync RX frame size

Offset 0x2 — RTDEXSYNC_CH_TX_DATA_FRAME_SIZE

This register specifies the RTDEx frame size used for the data transmit path (FPGA to host). The frame size must be an integer multiple of the RTDEx transfer size.

31 to 24	23 to 0
Reserved	**RTDExSyncTxFrameSize**
-	R/W
-	-

Table 7 RTDExSync TX data frame size

Offset 0x4 — RTDEXSYNC_CH_TX_CNTX_FRAME_SIZE

This register specifies the RTDEx frame size used for the context and extended context transmit path (FPGA to host). This value can be different that the data packet size in order to maximize the packet overhead compare to the data payload and to minimize the context packet size. The context frame size cannot be smaller than the RTDEx transfer size.

31 to 24	23 to 0
Reserved	**RTDExSyncTxFrameSize**
-	R/W
-	-

Table 8 RTDExSync TX context frame size

3.9 Interfacing the RTDExSync Core to the User Logic

The user side of an RTDExSync channel is a very simple FIFO interface. This section describes this interface.

3.9.1 Downlink Ports

The signals available to the user for RX channels are:

- **FIFO not empty** (RTDExSyncRxReady)
- **Read strobe** (RTDExSyncRxReReq)
- **Data out** (RTDExSyncRxData)
- **Data valid** (RTDExSyncRxDataValid)

The signals of the unused channels can be left unconnected. All the FIFOs of the RX channels share the same user clock.

When receiving data in the FPGA from the host (**downlink**), RTDExSyncRxReady indicates that there is an active transmission and that data is available in the FIFO. If so, RTDExSyncRxReReq can be set high to perform a read operation. If the read operation is successful, RTDExSyncRxDataValid will become high two clock cycles of user clock later and the data present on RTDExSyncRxData will then be valid.

3.9.2 Uplink Ports

The signals available to the user for TX channels are:

- **FIFO not full** (RTDExSyncTxReady)
- **Write strobe** (RTDExSyncTxWriteReq)
- **Data input** (RTDExSyncTxData)

The signals of the unused channels can be left unconnected. All the FIFOs of the TX channels share the user clock of the RX channels.

To transmit data from the FPGA to the host (**uplink**), the RTDExSyncTxReady flag must be high. This indicates that a transmission is active and the FIFO will correctly behave if a write operation is performed. If this flag is high, RTDExSyncTxWriteReq can be set high as well and the data currently on the RTDExSyncTxData bus will be written to the transmit FIFO. If RTDExSyncTxReady is low, there is no active transmission so any write operation must be avoided. The FIFO will never reach a full status since samples are discarded by the RTDExSync module to avoid this condition.

NOTE:
In the case the RTDExSync core clock is slower than the user clock, it is possible that the samples cannot be discarded fast enough causing the FIFO full condition to happen. **Always use a core clock that has a higher frequency than the user clock**.

3.9.3 Downlink Timing

In **downlink**, the value of the timestamp associated to the state-machine changing to the **Running** state is equal to the time when the first sample is available in the user logic. This is true if the FIFO **read request** is done at the same clock cycle as when the FIFO **ready** signal goes high. Since the FIFO has a read latency of two clock periods, the value of this timestamp can also be interpreted has the time the FIFO became **ready** plus two clock cycles.

The value of the timestamp associated to the state-machine changing to the **Idle** change of state is equal to the time when the first invalid sample is available in the user logic. This is true if the FIFO **read request** is done at the clock cycle just before the FIFO **ready** signal goes low. Since the FIFO has a read latency of two clock periods, the value of this timestamp can also be interpreted has the time the FIFO becomes **not ready** plus two clock cycles.

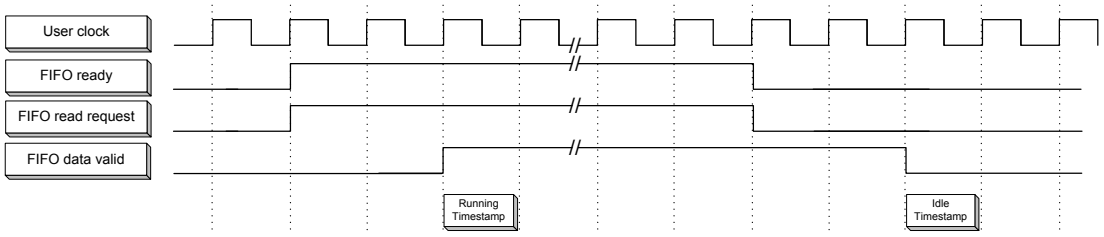

Figure 5 Downlink precise timestamp timing diagram

When a transfer is configured in the **start immediately** mode, the FIFO ready signal goes high once the first packet has been stored inside the FIFO or when the FIFO is almost full.

When a transfer is configured in the **stop immediately** mode, the FIFO ready signal goes low once the last sample has been requested from the FIFO.

When a transfer is configured in the **start at trigger** mode, the FIFO ready signal goes high 2 clock cycles after the trigger rising-edge event. The data becomes valid only 4 clock cycles after the trigger event. This delay is due to the trigger detection logic.

When a transfer is configured in the **stop at trigger** mode, the FIFO ready signal goes low 2 clock cycles after the trigger rising-edge event. The data becomes invalid only 4 clock cycles after the trigger event. This delay is due to the trigger detection logic.

When a transfer is configured in the **start at time** mode, the FIFO ready signal goes high 2 clock cycles before the specified time. The data will become valid exactly at the specified time.

When a transfer is configured in the **stop at time** condition, the FIFO ready signal goes low 2 clock cycles before the specified time. The data will become invalid exactly at the specified time.

3.9.4 Uplink Timing

In **uplink**, the value of the timestamp associated to the state-machine changing to the **Running** state is equal to the time when the first sample is written inside the FIFO by the user logic. This is true if the FIFO **write request** is done at the same clock cycle as when the FIFO **ready** signal goes high. The value of this timestamp can also be interpreted as the time the FIFO became **ready**.

The value of the timestamp associated to the state-machine changing to the **Idle** state is equal to the time where the first sample cannot be written inside FIFO by the user logic. This is true if the FIFO **write request** is done at the same clock cycle than the FIFO **ready** signal goes low. The value of this timestamp can also be interpreted as the time the FIFO became **not ready**.

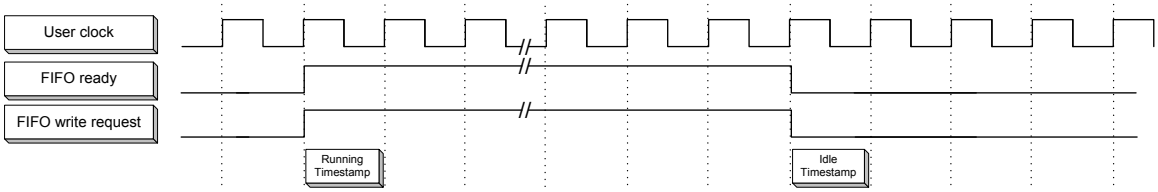

Figure 6 Uplink precise timestamp timing diagram

When a transfer is configured in the **start immediately** mode, the FIFO ready signal goes high as soon as possible.

When a transfer is configured in the **stop immediately** mode, the FIFO ready signal goes low as soon as possible.

When a transfer is configured in the **start at trigger** mode, the FIFO ready signal goes high 4 clock cycles after the trigger rising-edge event. This delay has been set to match the uplink trigger delay value.

When a transfer is configured in the **stop at trigger** mode, the FIFO ready signal goes low 4 clock cycles after the trigger rising-edge event. This delay has been set to match the uplink trigger delay value

When a transfer is configured in the **start at time** mode, the FIFO ready signal goes high at the specified time.

When a transfer is configured in the **stop at time** mode, the FIFO ready signal goes low at the specified time.

4 RTDExSync Host Programming

This chapter provides the functions of the RTDExSync library as well as the procedure to build a host application.

4.1 RTDExSync Library

The table below presents the functions and use of the RTDExSync library.

Function	Description
adp_result_t RTDExSync_Open(connection_state state, uint8_t u8ChIdx, RTDExSync_Dir_t dir, uint32_t u32RTDExEthBaseAddr, uint32_t u32SubframeSize, uint32_t u32FrameSize, uint32_t *pu32MaxIQSamples, RTDExSync_Handle_t *phRTDExSync);	Open RTDExSync instance. The sub-frame size (physical media packet size) and the frame size of an RTDExSync packet must be specified. The frame size must be an integer multiple of the sub-frame size. The open function returns the handle to the RTDExSync instance for the designated channel as well as the maximal IQ samples (32-bit samples) that can be contained in one frame size. In order to maximize the payload versus the overhead caused by the frame header, it is recommended to use integer multiple of this maximal IQ samples when performing “Send” command.
adp_result_t RTDExSync_Start(RTDExSync_Handle_t hRTDExSync, RTDExSync_Mode_t mode, RTDExSync_TrigSel_t trigSel, uint64_t u64StartTime);	Program the start mode of a new stream of transmission. The available modes are **immediately**, **at trigger** and **at time**. The **trigSel** is only used in **at trigger** mode and is used to choose between input trigger 0 to input trigger 3. The **u64StartTime** is only used in **at time** mode and specifies the start time in clock ticks based on the **FPGA time** input port of the RTDExSync FPGA core. In **downlink**, the start command will be sent along the first data packet (when RTDExSync_Send is called). It is done to avoid sending an empty packet and to make sure that there is available data inside the FIFO when the start condition is met. In **uplink**, the stop command will be sent immediately in an empty data packet inside this function.
adp_result_t RTDExSync_Stop(RTDExSync_Handle_t hRTDExSync, RTDExSync_Mode_t mode, RTDExSync_TrigSel_t trigSel, uint64_t u64StopTime);	Program the stop mode of a transmission stream. The available modes are **immediately**, **at trigger** and **at time**. The **trigSel** is only used in **at trigger** mode and is used to choose between input trigger 0 to input trigger 3. The **u64StopTime** is only used in **at time** mode and specifies the stop time in clock ticks based on the **FPGA time** input port of the RTDExSync FPGA core. In **downlink**, In **Stop at trigger** or **Stop at time** mode, the stop command will be sent along a data packet (when RTDExSync_Send will be called). The stop condition cannot be sent in the same packet as the start condition. It is recommended to send the stop condition in the second data packet sent. In **downlink**, in **Stop immediately** mode, the stop command will be sent immediately in an empty data packet inside this function. The transmission will stop once all the data inside the FIFO is read. In **uplink**, the stop command will be sent immediately in an empty data packet inside this function.

Function	Description
adp_result_t RTDExSync_Send(RTDExSync_Handle_t hRTDExSync, void *pIQSample, uint32_t u32NumSample, uint32_t u32EvtQueueSize, RTDExSync_Event_t *pEvtQueue, uint32_t *pu32EvtCount);	Perform an RTDExSync transfer from the host to the device. Can only be called in **downlink** direction. At the end of the send function, if a context packet is received, the context will be extracted and stored as the event queue. These received context packets generate events when the FPGA state changes or when errors are detected. The send function is always blocking. It will block until all requested samples has been sent. If fewer samples than requested are sent and no error are returned, it means the channel state is idle (the stop condition has been met).
adp_result_t RTDExSync_Receive(RTDExSync_Handle_t hRTDExSync, void *pIQSample, uint32_t u32NumSample, uint32_t u32EvtQueueSize, RTDExSync_Event_t *pEvtQueue, uint32_t *pu32EvtCount, int block);	Perform an RTDExSync transfer from the device to the host. Can only be called in **uplink** direction. The function iterates until the specified receive size is reached. In blocking mode, the receive function will only return when the requested amount of data is available or if the FPGA state is idle (the stop condition has been met). In non-blocking mode, the function does not wait for an additional packet to be received. If a context packet is received, the context will be extracted and stored as the event queue. If a discontinuity is found in the samples received, an "overrun" event is generated and the returned values will be zeros until all the missing samples have been requested by the user.
adp_result_t RTDExSync_IsStarted(RTDExSync_Handle_t hRTDExSync, uint32_t u32EvtQueueSize, RTDExSync_Event_t *pEvtQueue, uint32_t *pu32EvtCount, uint32_t u32TimeoutMs);	Check if the RTDExSync channel is started or not. To return "started", the change of state to **Running** event must has been received by the host computer. The function will keep returning "started" even if the channel is not in the **Running** state anymore. When the transmission is over, the channel will be "started" until the next call to the start function. In **downlink** direction, if the state is unknown, the receive function will be called to see if a context packet has been received. The timeout is only used in this mode when the received function is called. In **uplink** direction, if the state is unknown, the receive function is not called to see if a context packet has been received since it can corrupt the received data buffer. The RTDExSync_Receive function must be called in order to receive the context packet.
adp_result_t RTDExSync_IsStopped(RTDExSync_Handle_t hRTDExSync, uint32_t u32EvtQueueSize, RTDExSync_Event_t *pEvtQueue, uint32_t *pu32EvtCount, uint32_t u32TimeoutMs);	Check if the RTDExSync channel is stopped or not. To return "stopped", the change of state to **Idle** event must has been received by the host computer. The channel will be "stopped" until the next call to the start function.
void RTDExSync_Close(RTDExSync_Handle_t hRTDExSync);	Close the RTDEx channel, terminate the connection with the platform and free dynamically allocated memory.
adp_result_t RTDExSync_GetChInfo(connection_state * state, uint8_t u8Channel, int *pPresent, int *pDir);	Get the info about a RTDExSync channel. Return the channel presence status and its direction. • Present: 1 means that the channel is present. • Direction: 0 means uplink direction, 1 means downlink direction.
adp_result_t RTDExSync_ResetCore(connection_state * state);	Reset the RTDExSync FPGA core. This will affect all channels. It is only recommended to use this function to start an example from a known state. During a program execution, it is recommended to use the RTDExSync_Abort function since it only affects a specific channel.
adp_result_t RTDExSync_Abort(RTDExSync_Handle_t hRTDExSync);	Abort the current RTDExSync state in order to be able to handle a new start command in a known state.

Table 9 RTDExSync library functions

4.2 Building a Host Application Using RTDExSync

4.2.1 Send and Stop Immediately in the Downlink Direction

The following diagram illustrates the typical flow of an RTDExSync application in **downlink** direction in **Start immediately** and **Stop immediately** mode.

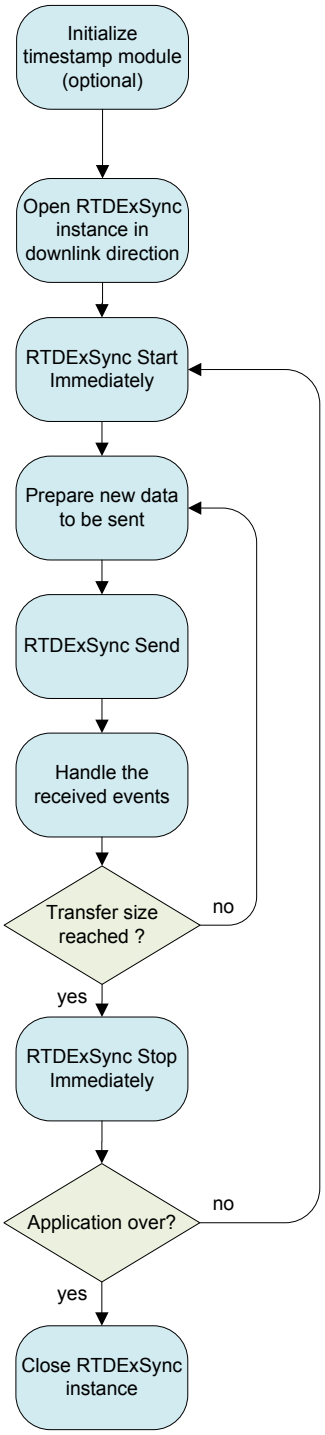

Figure 7 RTDExSync downlink, start and stop immediately flow graph

4.2.2 Send and Stop Immediately in the Uplink Direction

The following diagram illustrates the typical flow of an RTDExSync application in **uplink** direction in **Start immediately** and **Stop immediately** mode.

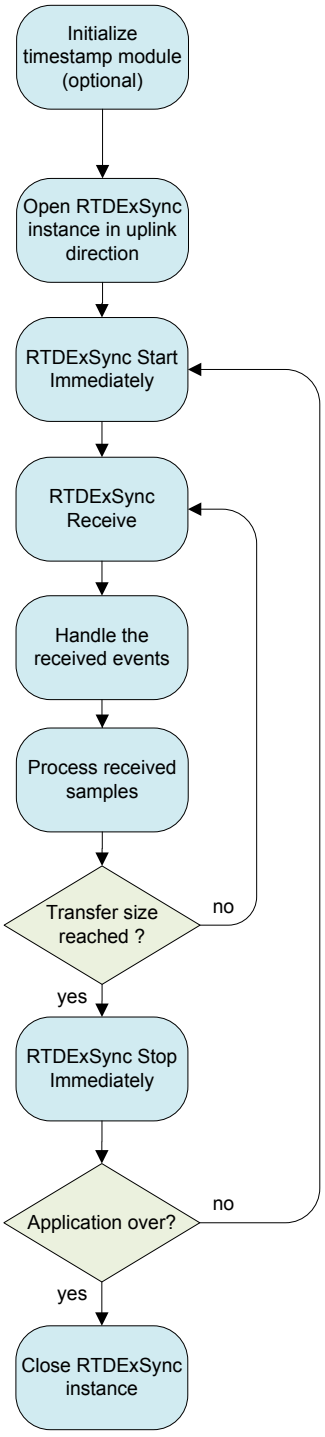

Figure 8 RTDExSync uplink, start and stop immediately flow graph

4.2.3 Send and Stop at Trigger or at Time in the Downlink Direction

The following diagram illustrates the typical flow of an RTDExSync application in **downlink** direction in **Start at trigger** and **Stop at trigger** mode.

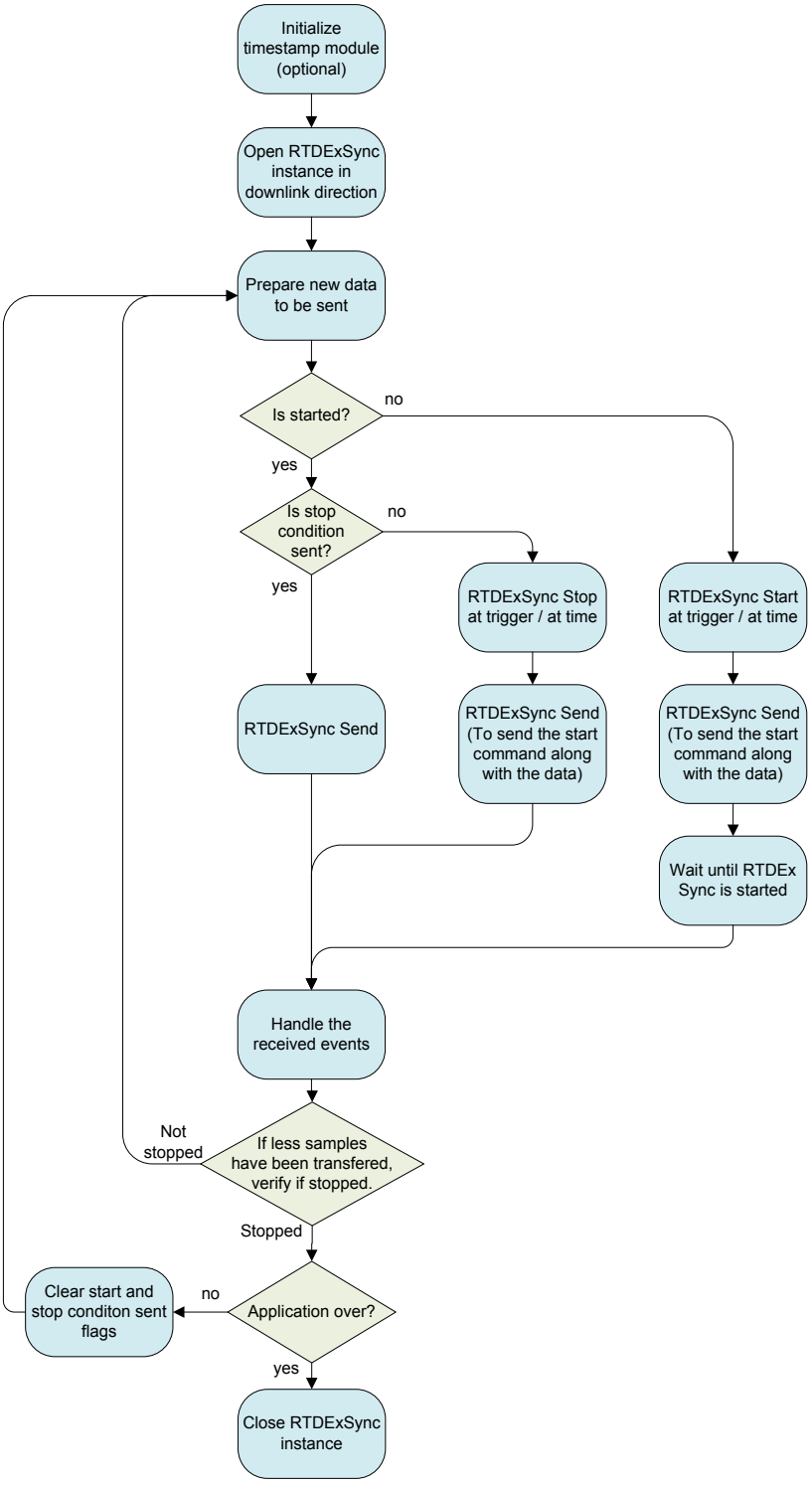

Figure 9 RTDExSync downlink, start and stop at trigger or at time flow graph

The RTDExSync **Send** function can return a lower number of samples than requested when the stop condition has been met. Instead of continuously calling the **IsStopped** function, the user application can only monitor the number of samples sent which is returned by the **Send** function.

Even though it is possible to send new data packets to the platform after the start command is sent and before the start condition is met, this is not shown in this flow chart. Since there is only one global flow control mechanism for all the RTDEx channels, calling multiple **Send** functions on a **Start Pending** channel will block the data pipe for all the channels. If other channels are also trying to send data, chances are these channels will starve. This explains why the **IsStarted** function is called before sending any new data packet.

However, it is recommended to send along with the start condition enough data to almost fill the data FIFO. The host needs time to receive and parse the events notifying it the transfer has started and this will delay the transmission of new data packets. There must be enough data placed inside the FIFO to make sure no underrun occurs between the start condition and the reception of the next data packets. Increasing the FIFO size can help avoiding this type of underrun.

For the **At Time** start or stop condition, is it highly recommended to initialize the timestamp module. If only relative timestamps are required between the uplink and downlink channels, resetting the timestamp module is all that is needed to do during the initialization. When reset, the timestamp module counter is set to 0 and immediately starts counting every user clock period. If an absolute timestamp is required, a PPS signal connected to the platform is required. Then, the host can set the time of the timestamp module between two PPS event. At the next PPS event, the time set will be applied to the timestamp module counter and will start counting every user clock period since this PPS event.

4.2.4 Send and Stop at Trigger or at Time in the Uplink Direction

The following diagram illustrates the typical flow of an RTDExSync application in **uplink** direction in **Start at trigger** and **Stop at trigger** mode.

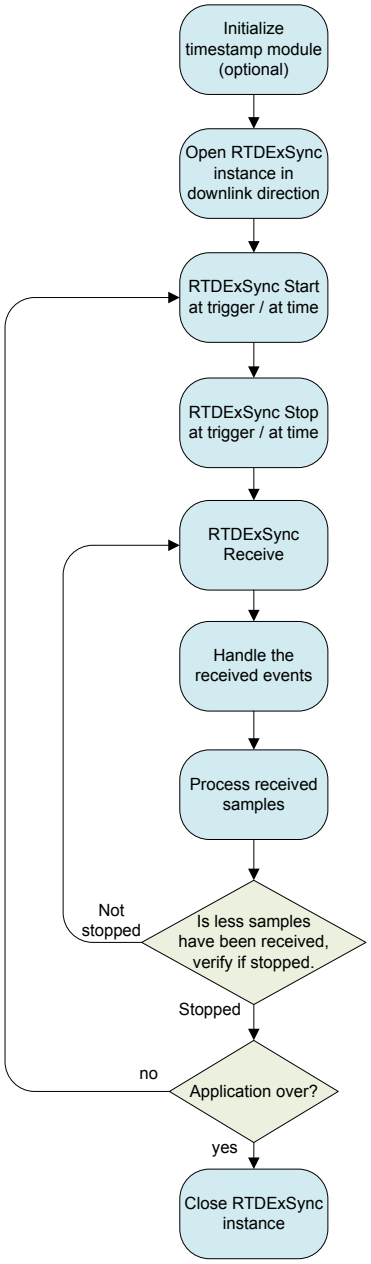

Figure 10 RTDExSync uplink, start and stop at trigger or at time flow graph

The RTDExSync **Receive** function can return a lower number of samples than requested when the stop condition has been met. Instead of continuously calling the **IsStopped** function, the user application can only monitor the number of samples received which is returned by the **Receive** function if argument **blocking** is set to 1.

If the blocking argument is not set, the **Receive** function can timeout and a number of samples lower than requested can be read, make sure to verify the returned value with every call of the function in that mode. In this non-blocking mode, the **IsStopped** function can be called to know if the reason the timeout happened is because the transfer is stopped.

5 Host Applications Using the RTDExSync EAPI Library

The BAS software suite provides two example utility applications that the user can modify. These two applications demonstrate how to stream data to and from the host using RTDEx GigE or PCIe, depending on the media that's instantiated in the FPGA.

- *RxSyncStreaming* is used to stream data from the FPGA to the host.
- *TxSyncStreaming* is used to stream data from the host to the FPGA.

Two examples showcase how these two applications are used and are available in your *%BASROOT%\examples\radio420\host\scripts\rtdexsync* directory. They use the Radio420 FMC. These boards are installed inside a PicoSDR.

1. Radio420_RxSyncStreaming configures the Radio420. Data sampled by the FMC is then streamed using RTDExSync and stored on the host PC.
2. Radio420_TxSyncStreaming transfers data from the host PC to the Radio420 core, then to the Radio420 FMC.

Please refer to the Radio420 - RtdexSync Examples Guide to execute these examples.

5.1.1 RxSyncStreaming description

Usage: *RxSyncStreaming <ip address> <configuration file name>*

- Parameter 1 must be the carrier IP address, ex 192.168.0.101
- Parameter 2 must be the Configuration file name, ex RxSyncStreaming.ini

This application configures the RTDEx core to send data from the FPGA to the host. The application continuously receives the data and saves it to a file, specified in RxSyncStreaming.ini.

The principal function of this application is function `RxSyncStreaming()`.

1. For each carrier specified through the application's command line arguments,
    - The application calls function `connect_cce()` to connect to the carrier's CCE and get a connection state object
2. For each channel,
    - The application opens the RTDExSync channel using function `RTDExSyncOpen()`.
    - Then, it initializes two data queues. These data queues are used to transfer data between the threads that will be responsible for receiving the RTDExSync stream, and writing the samples to a file. Each channel has a "Free buffer" queue, and a "Full buffer" queue.
    - After initializing one mutex (in total, not per channel), the application starts a `SaveTask` thread and a `RxTask` thread. `RxTask` calls `RTDExSyncReceive()` in a loop and passes

the data to the SaveTask thread using the buffer queues initialized before. SaveTask then stores the data to a file.

3. The function then waits for all channel threads to end before returning.

5.1.2 RxSyncStreaming configuration file

Instead of getting its parameters through command line arguments, application *RxSyncStreaming* uses a separate configuration file to gather them. The path to this file is passed as an argument to the application. Details on how Nutaq's configuration files are built are available in the application note "*App Note - Parsing and Creating Configuration Files*" residing in the %BASROOT%*doc**app_notes* repository.

To get a feel of what the *RxSyncStreaming* configuration file looks like, the reader is encouraged to open file *RxSyncStreaming.ini* in %BASROOT%\examples\radio420\host\scripts with a text editor. It is provided with the Radio420 applicative example. The file is similar to this example:

```
RTDExSync_subframesize=8192
RTDExSync_queuesize=100
RTDExSync_numberofsamples=204800000
RTDExSync_framesize=65536
real_time_flag=1

RTDExSync_startmode=2
RTDExSync_stopmode=2
RTDExSync_starttrigger=0
RTDExSync_stoptrigger=0
RTDExSync_starttime=16777216
RTDExSync_stoptime=16778016

[RTDEx_1]
type=RTDExSync
carrier_position=1
channel_number=1
filename=../../bin/rxsync1.bin
```

Configuration files for using RxSyncStreaming must contain all RTDExSync parameters.

First, parameters common to all RTDExSync channels. These parameters are not enclosed within a section:

- `RTDEx_subframesize.` When the media is GigE, this is the Ethernet packet size used to transfer data between the host PC and the FPGA.
- `RTDEx_queuesize.` Depth of the data queues used in RxSyncStreaming
- `RTDEx_numberofsamples.` Size of the transfer to perform before the RTDExSync streaming is stopped, if the stop condition is `TrigMode_Immediately.`
- `RTDEx_framesize.` Refers to the number of bytes to receive (and wait for) each time function RTDExSyncReceive() is called.
- `real_time_flag.` Specifies whether the operating system grants this application the real-time privilege.

- `RTDExSync_startmode.` This is the RTDExSync start mode. This refers to the type of event that makes the first sample be sent from the FPGA to the host PC.
  - `0:` Immediately when `RTDExSyncStart()` is called.
  - `1:` Start on trigger.
  - `2:` Start on specific time (in clock ticks)
- `RTDExSync_stopmode.` This is the `RTDExSync` stop mode. This refers to the type of event that stops the transfer.
  - `0:` Immediately when `RTDExSyncStop()` is called.
  - `1:` Stop on trigger.
  - `2:` Stop on specific time (in clock ticks)
- `RTDExSync_starttrigger.` This is the `RTDExSync` trigger selection, if start mode is by trigger.
  - `0:` Trigger input 0 on RTDExSync Core
  - `1:` Trigger input 1 on RTDExSync Core
  - `2:` Trigger input 2 on RTDExSync Core
  - `3:` Trigger input 3 on RTDExSync Core
- `RTDExSync_stoptrigger`. This is the `RTDExSync` trigger selection, if stop mode is by trigger.
  - `0:` Trigger input 0 on RTDExSync Core
  - `1:` Trigger input 1 on RTDExSync Core
  - `2:` Trigger input 2 on RTDExSync Core
  - `3:` Trigger input 3 on RTDExSync Core
- `RTDExSync_starttime.` This is the start time, in clock ticks, if the start mode is on time.
- `RTDExSync_stoptime.` This is the stop time, in clock ticks, if the stop mode is on time.

Then, parameters specific to each RTDExSync channel. These parameters are contained within a section. Each section gives parameters for one RTDExSync channel.

- `type` (an instance of that parameter must be equal to `RTDExSync`)
- `carrier_position`. This is the carrier number this RTDExSync channel will be instantiated on. This refers to the IP address list given as command line arguments to RxSyncStreaming. For example, a value of "1" means that this RTDExSync channel will run on the carrier configured with the first IP address specified in the list passed to the application.
- `channel_number`. The RTDExSync channel number to use on the carrier.
- `filename`. Name of the file in which received data is to be stored.

5.1.3 TxSyncStreaming description

Usage: *TxSyncStreaming <ip address> <configuration file name>*

- Parameter 1 must be the carrier IP address, ex 192.168.0.101
- Parameter 2 must be the Configuration file name, ex TxSyncStreaming.ini

This application configures the RTDExSync core to receive data from the host to the FPGA. The application continuously reads data from a file and send it to the FPGA through RTDEx.

The principal function of this application is function `TxStreaming()`.

1. For each carrier specified through the application's command line arguments,

- The application calls function `connect_cce()` to connect to the carrier's CCE and get a connection state object

2. For each channel,
    - The application opens the RTDExSync channel using function `RTDExSyncOpen()`.
    - Then, it initializes two data queues. These data queues are used to transfer data between the threads that will be responsible for reading the data from the file, and sending the RTDExSync stream. Each RTDExSync channel has a "Free buffer" queue, and a "Full buffer" queue.
    - After initializing two mutex (in total, not per channel) and a semaphore, the application starts a `ReadTask` thread. This thread continuously reads from the data file and puts the data into a queue. Finally, the application starts `TxTask` which receives the data from the ReadTask thread and calls `RTDExSyncSend()` on the data.
3. The function then waits for all channel threads to end before returning.

5.1.4 TxSyncStreaming configuration file

Instead of getting its parameters through command line arguments, application *TxSyncStreaming* uses a separate configuration file to gather them. The path to this file is passed as an argument to the application. Details on how Nutaq's configuration files are built are available in the application note "*App Note - Parsing and Creating Configuration Files*" residing in the %BASROOT%*doc**app_notes* repository.

To get a feel of what the T*xSyncStreaming* configuration file looks like, the reader is encouraged to open file *TxSyncStreaming.ini* in %BASROOT%\examples\radio420\host\scripts with a text editor. It is provided with the Radio420 applicative example. The file is similar to this example:

```
RTDExSync_subframesize=8192
RTDExSync_queuesize=100
RTDExSync_numberofsamples=204800000
RTDExSync_framesize=65536
real_time_flag=1

RTDExSync_startmode=2
RTDExSync_stopmode=2
RTDExSync_starttrigger=0
RTDExSync_stoptrigger=0
RTDExSync_starttime=16777216
RTDExSync_stoptime=16778016

[RTDEx_1]
type=RTDExSync
carrier_position=1
channel_number=1
filename=../../bin/rxsync1.bin
```

Configuration files for using TxSyncStreaming must contain all RTDExSync parameters.

First, parameters common to all RTDExSync channels. These parameters are not enclosed within a section:

- `RTDEx_subframesize.` When the media is GigE, this is the Ethernet packet size used to transfer data between the host PC and the FPGA.
- `RTDEx_queuesize.` Depth of the data queues used in TxSyncStreaming
- `RTDEx_numberofsamples.` Size of the transfer to perform before the RTDExSync streaming is stopped, if the stop condition is `TrigMode_Immediately.`
- `RTDEx_framesize.` Refers to the number of bytes to receive (and wait for) each time function RTDExSyncReceive() is called.
- `real_time_flag.` Specifies whether the operating system grants this application real-time privilege.
- `RTDExSync_startmode.` This is the RTDExSync start mode. This refers to the type of event that makes the first sample be sent from the host PC to the FPGA user logic.
  - `0:` Immediately when `RTDExSyncStart()` is called.
  - `1:` Start on trigger.
  - `2:` Start on specific time (in clock ticks)
- `RTDExSync_stopmode.` This is the `RTDExSync` stop mode. This refers to the type of event that stops the transfer.
  - `0:` Immediately when `RTDExSyncStop()` is called.
  - `1:` Stop on trigger.
  - `2:` Stop on specific time (in clock ticks)
- `RTDExSync_starttrigger.` This is the `RTDExSync` trigger selection, if start mode is by trigger.
  - `0:` Trigger input 0 on RTDExSync Core
  - `1:` Trigger input 1 on RTDExSync Core
  - `2:` Trigger input 2 on RTDExSync Core
  - `3:` Trigger input 3 on RTDExSync Core
- `RTDExSync_stoptrigger`. This is the `RTDExSync` trigger selection, if stop mode is by trigger.
  - `0:` Trigger input 0 on RTDExSync Core
  - `1:` Trigger input 1 on RTDExSync Core
  - `2:` Trigger input 2 on RTDExSync Core
  - `3:` Trigger input 3 on RTDExSync Core
- `RTDExSync_starttime.` This is the start time, in clock ticks, if the start mode is on time.
- `RTDExSync_stoptime.` This is the stop time, in clock ticks, if the stop mode is on time.

Then, parameters specific to each RTDExSync channel. These parameters are contained within a section. Each section gives parameters for one RTDExSync channel.

- `type` (an instance of that parameter must be equal to `RTDExSync`)
- `carrier_position`. This is the carrier number this RTDExSync channel will be instantiated on. This refers to the IP address list given as command line arguments to TxSyncStreaming. For example, a value of “1” means that this RTDExSync channel will run on the carrier configured with the first IP address specified in the list passed to the application.
- `channel_number`. The RTDExSync channel number to use on the carrier.
- `filename`. Name of the file from which data is to be read.

5.1.5 Exploring the Applications Source Code and Modifying/Rebuilding it

Project folders, allowing the user to modify the source code of the applications are also available. They reside in *%BASROOT%\tools\apps\core*.

If rebuilding is needed, follow the steps described in the section appropriate to the operating system you are using, where <*application to rebuild*> is either

- *RxSyncStreaming,*
- *TxSyncStreaming,*

5.1.5.1 Windows

A folder named *prj_win* which contains the Visual Studio 2012 solution associated with the application is present in the folders listed above.

1. Start Microsoft Visual Studio.
2. On the **File** menu, point to **Open** and click **Project/Solution**.
3. Browse to the *%BASROOT%\tools\apps\core\<application to rebuild>* folder and select the *<application to rebuild>.sln* solution
4. Select the build configuration Release x64.
5. On the **Build** menu, click **Build Solution**.

5.1.5.2 Linux

A folder named *prj_linux* which contains a shell script and a makefile associated with the application is present in the folders listed above. Source files are available in the *inc* and *src* folders

1. Open a terminal and use the *cd* command to browse to the *%BASROOT%\tools\apps\core\<application to rebuild* > repository in the installation.
2. To build the application, run the following command in the Linux terminal.

   sudo ./build_demo.sh

6 RTDExTs Wrapper for Absolute Time

To provide absolute timestamp for **RTDExSync** start and stop conditions and events, a wrapper on top of **RTDExSync** and **Timestamp** host library is made available. To use this library, Nutaq's **RTDExSync** and **Timestamp** modules must be present inside the platform and a PPS must be provided to the system.

The diagram below shows the software architecture of the **RTDExTs** library. It shows that **RTDExTs** is a software wrapper on top of RTDExSync and Timestamp modules.

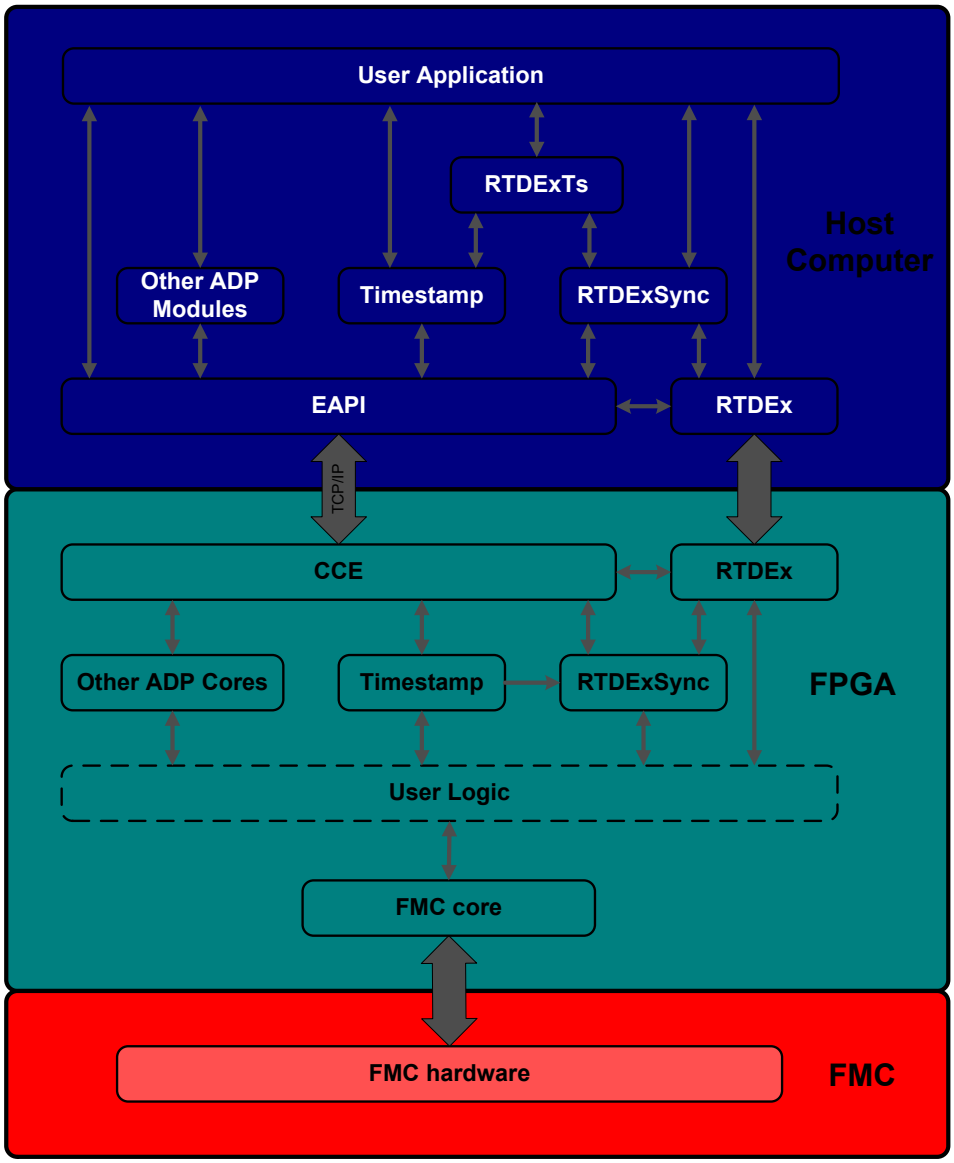

Figure 11 RTDExTs software architecture

For the wrapper to behave properly, it is required that the delay between the PPS events received by the platform and the time of the host computer doesn't exceed 100 milliseconds. This can easily be achieved if the host computer is synchronizing its time with an NTP client application and if the platform is receiving its PPS signal from a GPS receptor.

6.1 RTDExTs Library

The majority of the RTDExTs functions are the same as the functions of the RTDExSync library. The only difference is that the values for time passed or received in arguments are expressed in POSIX time plus a number of nanoseconds instead of a number of clock ticks.

The **Open** function is different than the one of the RTDExSync library since it also needs a handle to the **Timestamp** module. There also are additional functions to correctly handle time of the **Timestamp** module.

Function	Description
adp_result_t RTDExTs_Open(RTDExTs_Handle_t *phRTDExTs, connection_state state, uint8_t u8ChIdx, RTDExTs_Dir_t dir, uint32_t u32RTDExEthBaseAddr, uint32_t u32SubframeSize, uint32_t u32FrameSize, uint32_t u32TsClkFreq, uint32_t bTsInit, uint32_t *pu32MaxIQSamples);	Open RTDExTs instance. It is the same open function than RTDExSync except there are two additional parameters. **u32TsClkFreq** is the frequency of the clock connected to the timestamp module. **bTsInit** a boolean parameter to indicate if the time of the timestamp module will be set in this open function. When several uplink and downlink channels are present, they all share the same **Timestamp** module. All channels must know the clock frequency of the timestamp but only one channel needs to configure it. The **Timestamp** time is sent to the platform when the host time is not near a second transition (±100 ms). If the host time and PPS signal are minimally synchronized, this should assure that the absolute time will be correctly set inside the platform at the next PPS event.
adp_result_t RTDExTs_Start(RTDExTs_Handle_t hRTDExTs, RTDExTs_Mode_t mode, RTDExTs_TrigSel_t trigSel, uint32_t u32secStartTime, uint32_t u32nsecStartTime);	Program the start mode of a new stream of transmission. Same as RTDExSync start function but with POSIX time and nanosecond offset.
adp_result_t RTDExTs_Stop(RTDExTs_Handle_t hRTDExTs, RTDExTs_Mode_t mode, RTDExTs_TrigSel_t trigSel, uint32_t u32secStopTime, uint32_t u32nsecStopTime);	Program the stop mode of a new stream of transmission. Same as RTDExSync start function but with POSIX time and nanosecond offset.
adp_result_t RTDExTs_Send(RTDExTs_Handle_t hRTDExTs, void *pIQSample, uint32_t u32NumSample, uint32_t u32EvtQueueSize, RTDExTs_Event_t *pEvtQueue, uint32_t *pu32EvtCount);	Perform an RTDExSync transfer from the host to the device. Same as RTDExSync send function but with events in POSIX time and nanosecond offset.

Function	Description
adp_result_t RTDExTs_Receive(RTDExTs_Handle_t hRTDExTs, void *pIQSample, uint32_t u32NumSample, uint32_t u32EvtQueueSize, RTDExTs_Event_t *pEvtQueue, uint32_t *pu32EvtCount, int block);	Perform an RTDExSync transfer from the device to the host. Same as RTDExSync receive function but with events in POSIX time and nanosecond offset.
adp_result_t RTDExTs_IsStarted(RTDExTs_Handle_t hRTDExTs, uint32_t u32EvtQueueSize, RTDExTs_Event_t *pEvtQueue, uint32_t *pu32EvtCount, uint32_t u32TimeoutMs);	Check if the RTDExSync channel is started or not. Same as RTDExSync receive function but with events in POSIX time and nanosecond offset.
adp_result_t RTDExTs_IsStopped(RTDExTs_Handle_t hRTDExTs, uint32_t u32EvtQueueSize, RTDExTs_Event_t *pEvtQueue, uint32_t *pu32EvtCount, uint32_t u32TimeoutMs);	Check if the RTDExSync channel is stopped or not. Same as RTDExSync receive function but with events in POSIX time and nanosecond offset.
void RTDExTs_Close(RTDExTs_Handle_t hRTDExSync);	Close the RTDEx channel. Same as RTDExSync close function.
adp_result_t RTDExTs_GetChInfo(connection_state * state, uint8_t u8Channel, int *pPresent, int *pDir);	Get the info about a RTDExSync channel. Same as RTDExSync function.
adp_result_t RTDExTs_ResetCore(connection_state * state);	Reset the RTDExSync FPGA core. Same as RTDExSync function.
adp_result_t RTDExTs_Abort(RTDExTs_Handle_t hRTDExTs);	Abort the current RTDExTs state in order to be able to handle a new start command in a known state. Same as RTDExSync function.
adp_result_t RTDExTs_VerifyTime(RTDExTs_Handle_t hRTDExTs);	Verify that the platform time matches the host computer time. If a PPS event has not occurred since the time setting inside the Open function, after a timeout of 1.5 second, an error will be returned indicating that the PPS is absent. If the time read from the platform does not fit the host time within ±100 milliseconds, an out of sync error will be returned. The RTDExTs library does not make extra effort in order to synchronize the platform time if the host time is not within ±100 milliseconds of the PPS time. It is the user task to make sure the host time and the PPS signal that goes to the platform are synchronized. In order to so, use a GPS receiver to generate the PPS event and synchronize the host computer time to an NTP server.

Function	Description
adp_result_t RTDExTs_SetPropagationDelay(RTDExTs_Handle_t hRTDExTs, int32_t i32nsecDelay);	This function is used to compensate the propagation delay. The propagation delay can be used to compensate for the propagation time in the analog path, the ADC/DAC chips and the user logic algorithm. The propagation delay resolution is limited by the timestamp clock period. The time offset value is truncated to the previously clock period multiple if the specified offset does not fit the current resolution. For **downlink** direction, this value will be used to start and stop before the actual time. This is done to make sure the data converted into analog signal will be outputted at the desired time. For **uplink** direction, this value will be used to start and stop after the actual time. This is done to make sure the analog signal at the platform interface will be acquired at the desired time.
adp_result_t RTDExTs_GetHostPosixTime(uint32_t * pu32sec, uint32_t * pu32nsec);	Get the current host time. This function should not be used to have precise time but to have a good idea of the host time.
adp_result_t RTDExTs_GetHostReadableLocalTime(char * pcTime, uint32_t u32Length);	Get the current host time. This function should not be used to have precise nanoseconds time but to display in a readable format the computer local time. The **u32Length** argument is the **pcTime** maximal length. **pcTime** length should at least be 30 bytes.
adp_result_t RTDExTs_GetDevicePosixTime(RTDExTs_Handle_t hRTDExTs, uint32_t * pu32sec, uint32_t * pu32nsec);	Get the platform time. This function should not be used to have precise time but to have a good idea of the platform time. The communication latency is not compensated and several milliseconds can easily be required in order for the time to be available in the application.
adp_result_t RTDExTs_GetDeviceReadableLocalTime(RTDExTs_Handle_t hRTDExTs, char * pcTime, uint32_t u32Length);	Get the platform time in a readable format and in the host computer local time format. This function should not be used to have precise time but to have a good idea of the platform time. The communication latency is not compensated and several milliseconds can easily be required in order for the time to be available in the application. The **u32Length** argument is the **pcTime** maximal length. **pcTime** length should at least be 30 bytes.
adp_result_t RTDExTs_PosixToReadableLocalTime(uint32_t u32sec, uint32_t u32nsec, char *pcString, uint32_t u32Length);	Convert a POSIX time with nanosecond offset in a readable format and in the host computer local time format. The **u32Length** argument is the **pcTime** maximal length. **pcTime** length should at least be 30 bytes.

adp_result_t
RTDExTs_ReadableLocalTimeToPosix(
char *pcString,
uint32_t * pu32sec,
uint32_t * pu32nsec);
	Convert time expressed in a readable format and in the host computer local time format to POSIX time with nanosecond offset.
	The readable local time should have the following syntax:
	`"2014-11-20 10:22:06.123456789"`
	where the last 9 digit after the '.' represent an offset in nanosecond
	Careful, `"2014-11-20 10:22:06.5"` means 10:22:06 +5 ns (not +500 ms). To express +500 ms make sure remaining zeros are present to express the value in nanosecond:
	`"2014-11-20 10:22:06.500000000"`

Table 10 RTDExTs library functions

6.2 Building a Host Application Using RTDExTs

RTDExTs application initializations are built the same way as the **RTDExSync** applications are built except that the **Timestamp** module does not need to be initialized separately since it is initialized inside the **RTDExTs Open** function. To use the **RTDExTs** wrapper, it is required that a PPS signal is provided to the

platform and that the PPS is minimally synchronized with that the host time within ±100 milliseconds. Also, at least one **RTDExTs** channel, among all used channels, must initialize the **Timestamp** module during its **Open** function by setting its **bTsInit** argument to **1**.

The rest of the applications are similar than **RTDExSync** applications except that the time is expressed in POSIX with nanosecond offset instead of time in clock ticks.

7 Functional Examples

Functional examples for the RTDExSync have been implemented for Nutaq's Perseus AMC carrier. This chapter presents the examples and offers links to the example documentation.

7.1 BSDK Examples

The RTDExSync BSDK examples showcase the RTDExSync design on the Nutaq's Perseus AMC carrier platform using Xilinx's Platform Studio, and Microsoft Visual Studio.

The procedures for the RTDExSync examples are presented in the following documents

- *%BASROOT%\doc\cores\RTDEx\RTDExSync Examples Guide.pdf*

7.2 MBDK Examples

The RTDExSync MBDK examples aim to showcase the RTDExSync design on a Perseus with a Radio420 using Matlab, Xilinx's System Generator, Microsoft's Visual Studio, and Nutaq's Command Line Interface.

The procedures for the RTDExSync examples are presented in the following documents

- *%BASROOT%\doc\cores\RTDEx\RTDExSync Examples Guide.pdf*