PO8e Streaming Interface for the RZ

PO8e Overview

The RZ PO8e interface is an optional interface for RZ processor devices and is designed to transfer high channel-count data to a PCI Express card interface (PO8e) for real-time processing in custom applications. The PO8e card can be in the same computer as the TDT system, or in a dedicated computer.

The RZ connects to the PO8e card via a special DSP (RZDSP-U). This DSP has an interface located on the back panel of the RZ processor and connects to the PO8e via orange fiber optic cables provided with the system.

Data streamed through the PO8e is buffered at several points while the system copies it from the RZ to PC memory. When data is generated on the RZ unit and fed into the PO8e HAL object in Synapse Processing Tree (or the Stream_Remote_MC macro in RPvdsEx), this data is placed in a 10000 sample (per channel) FIFO buffer on the RZ processor. Data from this FIFO is transferred over the fiber optic link to the PO8e PCI Express card.

Note: There is a 10 kHz data streaming limit when streaming from the RZ processor to the PO8.

A shared library is provided (PO8eStreaming) along with a C/C++ interface for writing custom applications to collect data from the PO8e card in real-time. In the PO8eStreaming library a dedicated software thread actively attempts to read from the PCI Express card and places the transferred data into a RAM buffer. This structure allows the application program to query the API when convenient and read data in larger blocks. The RAM buffer is limited only by available memory, though the programmer should consume the data as soon as possible since this interface can transfer at rates up to 12 MB/second.

PO8e Installation

Synapse has a built-in object for the Processing Tree to stream data to the PO8e. This must be added to your Hardware Rig in Synapse and then simply connect the desired output stream to the PO8e object. See the Synapse manual for more information.

For RPvdsEx circuit design, the TDT drivers installs the PO8e circuit macro here:

C:\TDT\RPvdsEx\Macros\Device\PO8e_Streamer\

The PO8eStreaming libraries and examples can be found in:

C:\TDT\RPvdsEx\Examples\PO8e\

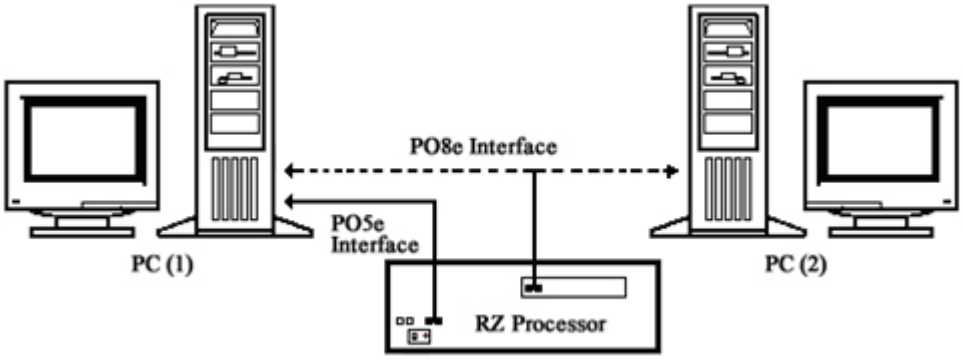
PO8e Hardware Requirements

Basic requirements include a paired fiber optic cable, an RZ processor equipped with the RZDSP-U card.

The PO8e requires a Windows or Linux computer with a PCI Express slot.

Setting-Up Your Hardware with the PO8e

In order to setup the RZ PO8e interface, connect the fiber optic cable from the RZ back panel to the PO8e card installed in the computer. The PO8e can be installed in the same computer as the PO5/e card or in a separate computer. For more information on setting up or configuring the RZ processor see the *System 3 Installation Guide.*



PO8e Connection Diagram

The diagram above illustrates the possible PO8e connections from the RZ processor to the TDT PC (1) or to a separate PC (2).

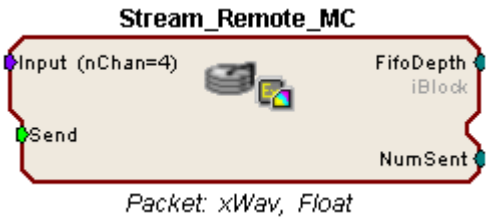
PO8e Circuit Design

Access to the PO8e interface is provided through the RPvdsEx macros named Stream_Remote_MC. This macro operates on multi-channel data and can be configured to specify the number of channels and data type.

Stream_Remote_MC Macro

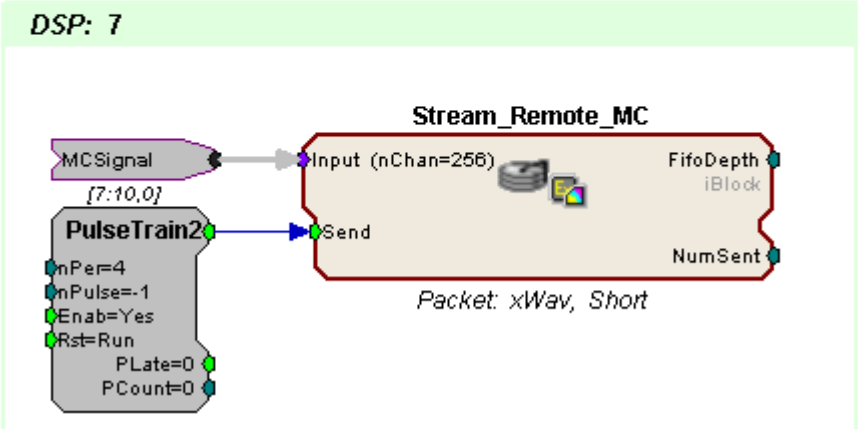
The Stream_Remote_MC macro is used to send data from the RZ to the PO8e card. All data is organized into packets according to the number of words (specified by the packet size) set in the macro setup properties dialog.

The macro accepts a multi-channel data stream as well as a logic input that tells the macro to send out a packet.



Sending Data Construct

Data is sent whenever the "Send" input receives a rising trigger (logic high (1)). Up to 256 channels can be sent on each Send signal. This occurs in one sample period. If the number of channels is greater than 256, data is sent in blocks and grouped together on the PO8e card's buffer.



In this circuit, 256 channels of data in Short format are sent to the PO8e card every fourth sample. The CoreSweepControl macro is required in any circuit using the Stream_Remote_MC macro. The Stream_Remote_MC macro must be placed on the special DSP that is physically connected to the PO8e card (DSP #7 in this case).

Note: To modify the number of channels sent and the data format, edit the parameters found in the Stream_Remote_MC macro setup properties.

About PO8e Streaming

PO8eStreaming is a library of methods for accessing data on one or several PO8e interfaces through a custom Windows or Linux application.

Both C and C++ interfaces are provided to this library. The C interface creates a pointer to a connected card, and then that pointer is passed to each subsequent function.

Users should be mindful of using good 'closed loop' access when working with PO8eStreaming. This means always releasing any open connections to PO8e cards.

A typical PO8e access session for a client consists of five main steps:

1. Run the circuit on the RZ device that streams to the PO8e card.
2. Call connectToCard to get a pointer to an available PO8e card.
3. Call startCollecting to begin reading from PO8e card.
4. Perform any number of buffer operations.
5. Call releaseCard to release the card object from memory.

Organization of PO8e Streaming Methods

PO8eStreaming methods can be divided into three basic groups:

- Setup and Control -- The methods in this group are used to setup access to any PO8e card(s) in the system.
- Hardware Data Access -- The methods in this group are used to read data from PO8e card(s).
- Hardware Information Retrieval -- The methods in this group are used to access information pertaining to current data stream, including number of channels and sample size in bytes.

Setup and Control *PO8e*

cardCount

Description: cardCount returns the number of PO8e cards detected in the system. Call this first to determine the possible values for the "index" passed to the constructor.

C++ prototype:
```
static int cardCount();
```

C prototype:
```
int cardCount();
```

Returns: The number of PO8e cards in the system.

Sample Code

C++
```
int totalCards = PO8e::cardCount();
```

C
```
int totalCards = cardCount();
```

connectToCard

Description: Returns a pointer to the specified card index. Note that the index will be consistent across system boots and is dependent on the PCIe bus layout, so if you move the cards between slots their respective indices can change.

C++ prototype:
```
static PO8e* connectToCard(unsigned int cardIndex = 0);
```

C prototype:
```
void* connectToCard(unsigned int cardIndex = 0);
```

Arguments: *cardIndex* Specify the target card by index.

Returns: Pointer to PO8e instance.

Sample Code This code sample creates a PO8e object pointing to the first card identified in the system.

C++
```
PO8e *card = PO8e::connectToCard(0);
```

C
```
void *card = connectToCard(0);
```

releaseCard

Description: Free the PO8e card objects through this interface. It is done this way to ensure that in Windows the objects are freed from the correct heap context.

C++ prototype:
```
static void releaseCard(PO8e *card);
```

C prototype:
```
void releaseCard(void* card);
```

Arguments:

card Pointer to PO8e object.

Sample Code This code sample releases the card object memory.

C++
```
PO8e::releaseCard(card)
```

C
```
releaseCard(card)
```


startCollecting

Description: Call this to start collecting a data stream from the PO8e card. Collected data will be buffered as needed.

C++ prototype:
```
bool startCollecting(bool detectStops = true);
```

C prototype:
```
bool startCollecting(void* card, bool detectStops = true);
```

Arguments:

detectStops Tell the PO8e to detect when the stream from the RZ is stopped.

Returns:

pointer Pointer to PO8e instance.

Sample Code

Description: This code sample tells an existing PO8e object to begin collecting data.

C++
```
card->startCollecting(true);
```

C
```
startCollecting(card, true);
```

stopCollecting

Description: Call this to stop collecting a data stream from the PO8e card.

C++ prototype:
```
void stopCollecting();
```

C prototype:
```
void stopCollecting(void* card);
```

Sample Code

Description: This code sample stops data collection on a PO8e object.

C++
```
card->stopCollecting();
```

C
```
stopCollecting(card);
```

waitForDataReady

Description: This function provides a means to efficiently wait for data to arrive from the RZ unit.

C++ prototype:
```
size_t waitForDataReady(int timeout = 0xFFFFFFFF);
```

C prototype:
```
int waitForDataReady(void* card, int timeout = 0xFFFFFFFF);
```

Arguments:

int	*timeout*	Maximum duration (in ms) to wait for streaming to begin.

Sample Code

Description: This code sample blocks execution until buffered data is ready on the card.

C++
```
card->waitForDataReady();
```

C
```
waitForDataReady(card);
```

samplesReady

Description: Returns the number of samples (per channel) that are currently buffered.

C++ prototype:
```
size_t samplesReady(bool *stopped = 0);
```

C prototype:
```
int samplesReady(void* card, bool *stopped = 0);
```

Arguments:

bool pointer	*stopped*	The value pointed to will be set to true if the underlying mechanisms detect that data has stopped flowing.

Sample Code

Description: This code returns the number of samples (per channel) currently buffered on the card and detects if streaming has stopped.

C++
```
bool stopped;
```

```
size_t numSamples = card-
>samplesReady(&stopped);

if (stopped)

    PO8e::releaseCard(card);
```

C

```
bool stopped;

int numSamples = samplesReady(card,
&stopped);

if (stopped)

    releaseCard(card);
```

readChannel

Description: Copy the data buffered for an individual channel. Note that this call does NOT advance the data pointer. Use calls to flushBufferedData to discard the data copied using this function. The user is responsible for ensuring that the buffer is large enough to hold nSamples * dataSampleSize( ) bytes. The optional offsets array should be nSamples long and will be populated with the data offset of each block. This allows a user to detect if the buffer on the RZ unit has overflowed.

C++ prototype:

```
int readChannel(int chanIndex, void *buffer,
int nSamples, int64_t *offsets = NULL);
```

C prototype:

```
int readChannel(void* card, int chanIndex,
void *buffer, int nSamples, int64_t
*offsets);
```

Arguments:

int	*chanIndex* The channel to read data from.
void pointer	*buffer* The location to write buffered data to.
int	*nSamples* The number of samples to read.
in64_t pointer	*offsets* The location to write the buffer indices to.

Returns:

int — Number of samples that were read.

Sample Code

Description: This code sample reads 1 sample from channel 2 and stores it in buff.

C++

```
short buff[8192];

card->readChannel(2, buff, 1);
```

C

```
short buff[8192];

readChannel(card, 2, buff, 1);
```

readBlock

Description: Copy the data buffered for all channels. Note that this call does NOT advance the data pointer. Use calls to flushBufferedData to discard the data copied using this function.

The data will be grouped by channel and the number of samples returned applies to all channels. The user is responsible for ensuring that the buffer is large enough to hold nSamples * numChannels( ) * dataSampleSize( ) bytes. The optional offsets array should be nSamples long and will be populated with the data offset of each block. This allows a user to detect if the buffer on the RZ unit has overflowed.

C++ prototype:
```
int readBlock(void *buffer, int nSamples, int64_t *offsets = NULL);
```

C prototype:
```
int readBlock(void* card, void *buffer, int nSamples, int64_t *offsets);
```

Arguments:

void pointer	*buffer*	The location to write buffered data to.
int	*nSamples*	The number of samples to read.
in64_t pointer	*offsets*	The location to write the buffer indices to.

Returns:

int — Number of samples that were read.

Sample Code

Description: This code sample reads 1 sample from all channels, stores it in a buffer and flushes that data from the card.

C++
```
short buff[1024];
card->readBlock(buff, 1);
card->flushBufferedData(1);
```

C
```
short buff[1024];
readBlock(card, buff, 1);
flushBufferedData(card, 1);
```

flushBufferedData

Description: Releases samples from each buffered channel.

C++ prototype:
```
void flushBufferedData(int numSamples = -1, bool freeBuffers = false);
```

C prototype:
```
void flushBufferedData(void* card, int numSamples = -1, bool freeBuffers = false);
```

Arguments:

int	*numSamples*	Number of samples to release. Passing -1 releases all buffered samples.
bool	*freeBuffers*	Controls the optional freeing of the underlying data buffers.

Sample Code

Description: This code sample flushes one sample from all channels.

C++
```
card->flushBufferedData(1);
```

C
```
flushBufferedData(card, 1);
```

Hardware Information Retrieval

numChannels

Description: Counts the number of channels in the current stream. This value is set in the Stream_Remote_MC macro. Changing the number of channels mid-stream triggers an error condition.

C++ prototype:
```
int numChannels();
```

C prototype:
```
int numChannels(void* card);
```

Returns:

int — Number of channels in the current data stream.

Sample Code

Description: This code determines how many channels are in the current stream.

C++
```
int nChannels = card->numChannels();
```

C
```
int nChannels = numChannels(card);
```

numBlocks

Description: Counts the number of blocks that the current stream is divided into. This value is set in the Stream_Remote_MC macro. Each block will contain the same number of channels, so dividing the value from numChannels( ) by this value will leave no remainder. Changing the number of blocks mid-stream triggers an error condition.

C++ prototype:
```
int numBlocks();
```

C prototype:
```
int numBlocks(void* card);
```

Returns:

int — Number of blocks the current data stream is divided into.

Sample Code

Description: This code determines how many blocks are in the current stream.

C++
```
int nBlocks = card->numBlocks();
```

C
```
int nBlocks = numBlocks(card);
```

dataSampleSize

Description: Returns the size in bytes of each data sample (per channel). This value is set in the Stream_Remote_MC macro. Changing the data type during a stream triggers an error condition.

C++ prototype:
```
int dataSampleSize();
```

C prototype:
```
int dataSampleSize(void* card);
```

Returns:

int Size of each data sample in bytes.

Sample Code

Description: This code determines how many bytes are in each sample.

C++
```
int size = card->dataSampleSize();
```

C
```
int size = dataSampleSize(card);
```

getLastError

Description: This returns the most recent error.

C++ prototype:
```
int getLastError();
```

C prototype:
```
int getLastError(void* card);
```

Returns:

int The most recent error code.

Sample Code

Description: This code determines how many channels are in the current stream.

C++
```
int nChannels = card->getLastError();
```

C
```
int nChannels = getLastError(card);
```

Examples

The example files below are installed with the TDT drivers package.

Files: C:\TDT\RPvdsEx\Examples\PO8e\PO8eTest.rcx, PO8eTest.exe, PO8e.h

Hardware: RZ2 Real-Time Processor

Overview: PO8eTest.exe connects to any PO8e card(s) in the PC, waits for a stream then displays the data rate that each PO8e card is receiving. PO8eTest.rcx streams 256 channels of floats to the PO8e card at 6.1 kHz. Launch PO8eTest.exe first, run the circuit and then set the zBusA trigger high to begin streaming.