XLink Component

User Guide

v1.2 / August 2018

Intel® Movidius™
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Revision History

Date	Version	Description
August 2018	1.2	Updated singleStream link.

Table of Contents

1 Introduction

This user guide explains the structure and functionality of the XLink component inside the Myriad Development Kit and the APIs used. The XLink component's purpose is to unify several communication protocols under one set of APIs.

1.1 Glossary of terms

CMX	Connection Matrix Crossbar
ldscript	linker description script
CNN	Convolutional Neural Network

2 Functionality

2.1 Purpose

The XLink's purpose is to unify communication protocols by allowing the user to call generic API functions. The two protocols that will be hosted by the first version of XLink are:

- **USB** – allows the Myriad chip to communicate through the USB 3.0 protocol. The primary goal of the XLink framework is to let the Myriad chip to be controlled without a JTAG connection, using USB only. It is primarily targeted for USB keys, but it can be used on any development board containing a Myriad chip.

  XLink has been developed with the goal of having the USB protocol separated from the actual operation of the component. It is to be extensible to have an XLink multiple protocol functionality. The framework is initially targeted for CNN applications with USB. The PC side API supports Linux only.
- **PCIe** (*to be implemented*) – Allows communication through the PCI Express protocol.

2.2 Features

The XLink component provides a basic communication protocol between (currently) two nodes. The communication is through USB. In the future, it will be generalized to work on any other protocol.

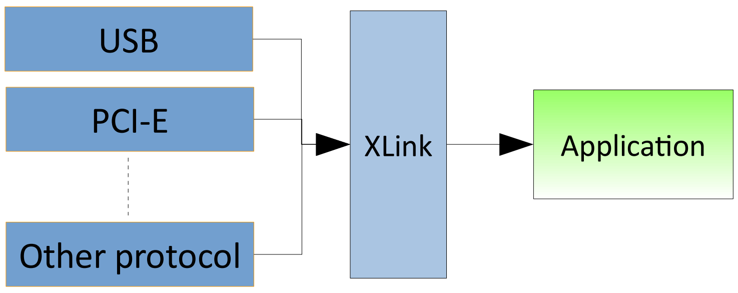


Figure 1: XLink component overview

2.3 Generic Transfer Functionality

Most of the functionality in XLink is not specific to the type of nodes. The generic functionality that XLink supports is as follows:

- Creates multiple communication channels between nodes on a single physical connection.
- Writes to the remote node on a specified channel.
- Reads packets written by the remote node when requested by the application.
- Boot/Reboot Myriad with a custom binary from a PC application.
- Close communication and reset Myriad from the PC application.

2.3.1 Design Features

- **Optimal bus utilization**

  The transfer of the data from one node to the other is done in a two step process. On the call of XLinkWriteData from the first node (N1), the data is written to the stream belonging to the other node (N2). This triggers the transaction as soon as it can after N1 has made the data available. When N2 wants to read the data, it calls XLinkReadData and reads from its stream. This reduces the read latency when N2 wants to read and optimizes the usage of the bus.
- **Extensible to other communication protocols**

  XLink is developed as a general data transfer component that is able to work over USB. The operation of the component has no dependency on USB. It is essentially a component that can be easily separable into (i) a generic data transfer component with a data read/write interface (xLink) and (ii) a USB implementation of the XLink interfaces for PC and Myriad and, soon, (iii) a PCIe implementation for PC and Myriad boards.
- **Generic to different architectures**

  There are still a few outstanding tasks to make XLink truly generic to different architectures. However, this is the design paradigm that was used in the development of XLink.

XLink can be seen as a combination of XLink's core structure and an implementation of a communication protocol (USB, PCIe etc.).

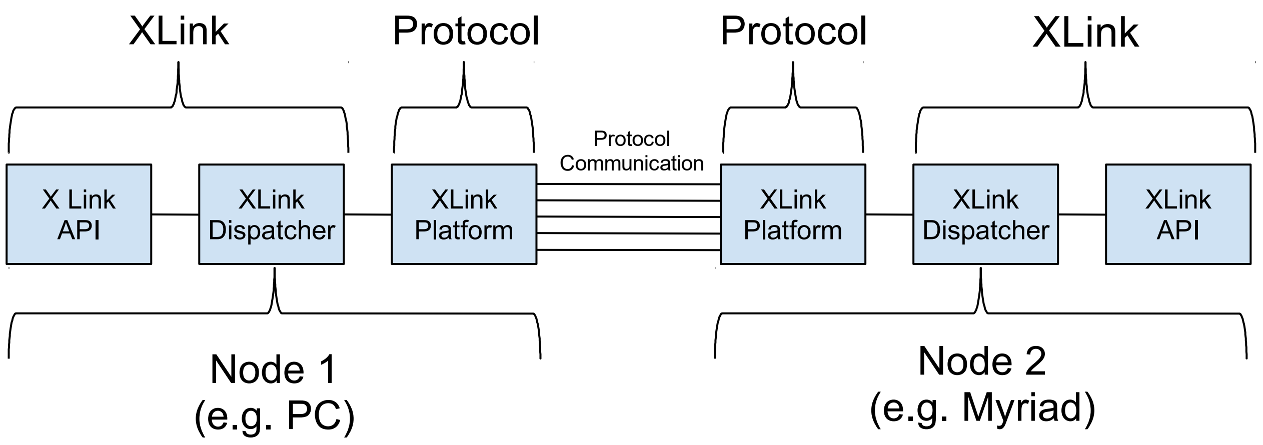


Figure 2: XLink component and XLinkPlatform

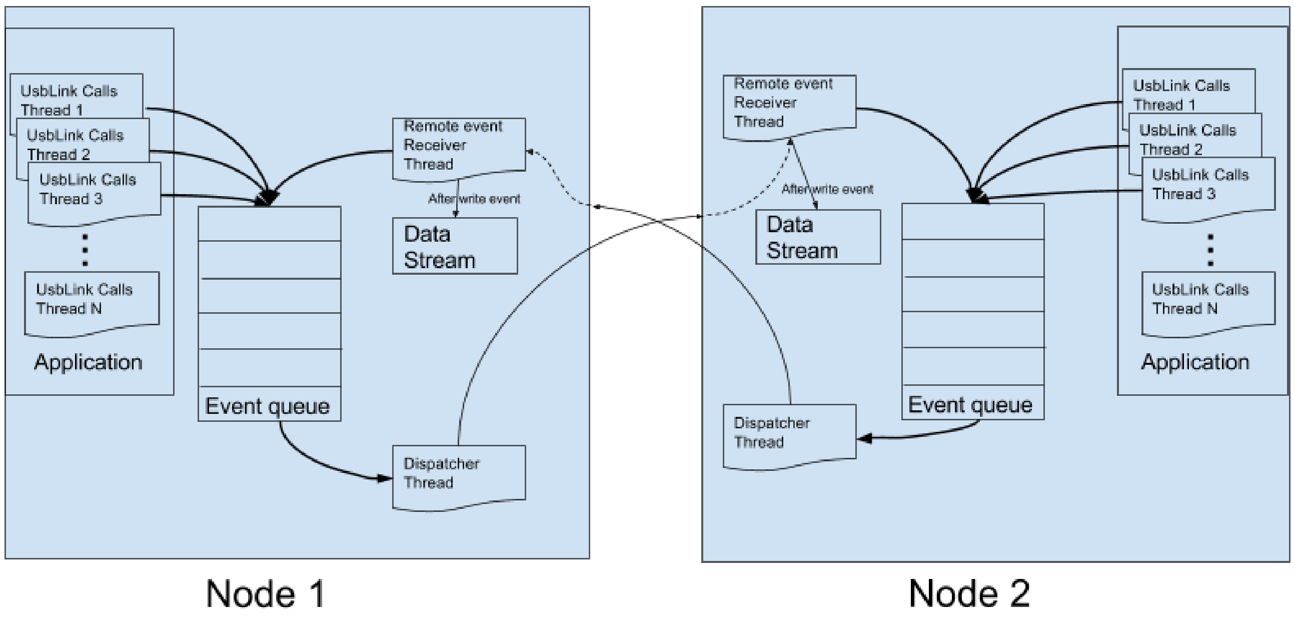


Figure 3: XLink data sharing over USB illustration

The data sharing has a basic error handling too, the Myriad chip being able to reject a symbol setup or query if the following conditions are not met:

- Buffer name invalid (i.e., too long).
- Size too big.
- The Myriad chip is running when the host tries to set or get data.

2.4 USB Protocol implementation

The communication protocol can use one of the following USB classes:

- VSC class
- CDC ACM class

The choice between the two USB classes is made by defining/undefining the macro `USE_USB_VSC`. The USB specific implementation is contained in the file called `UsbLinkPlatform.c` which implements the following functions:

```
int UsbLinkWrite(void* fd, void* data, int size,unsigned int timeout);
int UsbLinkRead(void* fd, void* data, int size, unsigned int timeout);
int UsbLinkPlatformConnect(const char* devPathRead,
                           const char* devPathWrite,
                          void** fd);
int UsbLinkPlatformInit(int loglevel);
int UsbLinkPlatformGetDeviceName(int index,
                                 char* name,
                                 int nameSize);
int UsbLinkPlatformBootRemote(const char* deviceName,
                              const char* binaryPath);
int UsbLinkPlatformResetRemote(void *fd);
void* allocateData(uint32_t size, uint32_t alignment);
void deallocateData(void* ptr, uint32_t size, uint32_t alignment);
```

2.5 Communication

In order to communicate, XLink needs to initialize communication and then open (at least) one stream. When the node calls `XLinkOpenStream`, a stream is opened on the remote node with the following attributes.

- **Name**: a char string.
- **Stream size**: the maximum data that can be stored by to the remote node.

After a buffer is created from node 1, the other node (node 2) can create a buffer with the same name. Then, two way communication between the two nodes is possible.

XLink deals with communication between nodes that involves the transport of fixed length packets between the nodes. If the packet sent is a 'write data' packet, this will be followed by data of the specified length.

The XLink application that runs on a platform runs at least 3 threads:

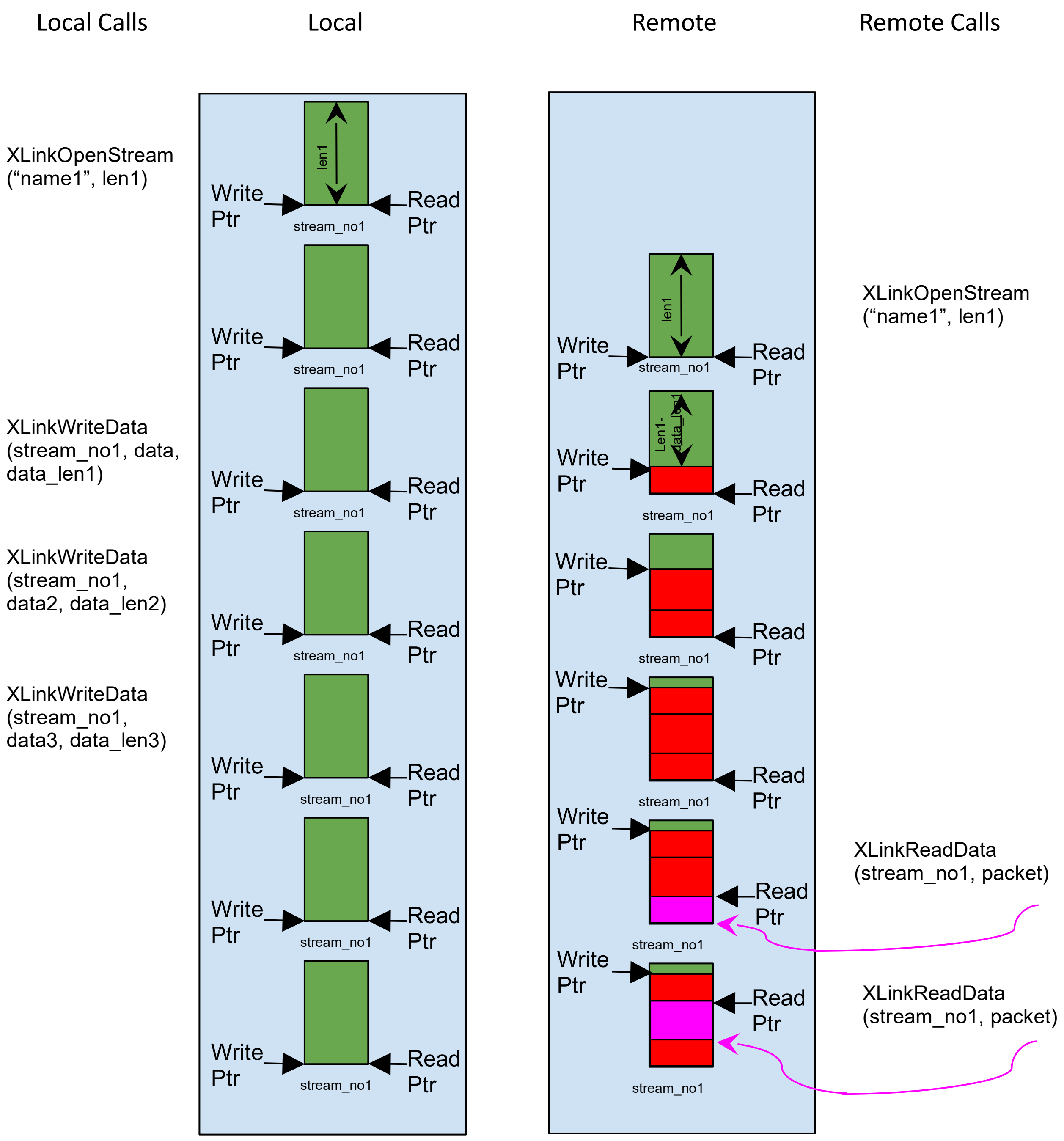
1. **Get Remote Events**: Single thread that handles the events that come in from the remote node. It adds this event to a processing queue. If the incoming event is a 'write data' packet, it will also add the data to the local stream.
2. **Get Local Events** (XLink calls from the local node): There can be multiple threads that add local events to the processing queue. This is dependent on the application. Each API call of XLink adds a local event to the processing queue so it is up to the user to decide how many threads they use to call the XLink API.
3. **Event Dispatcher**: Single thread that loops through the remote and local event queues and generates a response to the events. The response is either sent to the other node or stored in the local node if necessary.

Every time a node gets an event from the local application or from the remote event receiver, it adds it to the queue and does any data transfer (if required) straight away. The dispatcher then acts on the events in the queue in serial order.

2.6 XLink APIs/ Functions

```
// Initializes communication and pings the remote
XLinkError_t XLinkInitialize(XLinkGlobalHandler_t* handler);

// Connects to specific device, starts dispatcher and pings remote
XLinkError_t XlinkConnect(XLinkHandler_t* handler);

// Opens a stream in the remote that can be written to by the local
// Allocates stream_write_size (aligned up to 64 bytes) for that
stream
streamId_t XLinkOpenStream(char* name, int stream_write_size);

// Close stream for any further data transfer
// Stream will be deallocated when all pending data has been released
streamId_t XlinkCloseStream(streamId_t streamId);

/*Future development*/
// Currently useless
XLinkError_t XlinkGetAvailableStreams();
// Currently useless
XLinkError_t XlinkAsyncWriteData();

// Send a package to initiate the writing of data to a remote stream
// Note that the actual size of the written data is ALIGN_UP(size, 64)
XLinkError_t XLinkWriteData(int streamId, void* buffer, int size);

// Read data from local stream. Will only have something if it was
written to by the remote
XLinkError_t XLinkReadData(int streamId, streamPacketDesc_t** packet);

// Release data from stream - This should be called after ReadData
XLinkError_t XLinkReleaseData(int streamId);

// Boot the remote (This is intended as an interface to boot the
Myriad from PC)
XLinkError_t XLinkBootRemote(const char* deviceName,
                                const char* binaryPath);

// Reset the remote and reset local as well
XLinkError_t XLinkResetRemote(linkId_t id);

// Close all and release all memory
XLinkError_t  XlinkResetAll();

// Profiling functions
XLinkError_t XLinkProfStart();
XLinkError_t XLinkProfStop();
XLinkError_t XLinkProfPrint();
```

2.7 Working mechanism


Local Calls
Local
Remote
Remote Calls
XLinkOpenStream ("name1", len1)
XLinkWriteData (stream_no1, data, data_len1)
XLinkWriteData (stream_no1, data2, data_len2)
XLinkWriteData (stream_no1, data3, data_len3)
XLinkOpenStream ("name1", len1)
XLinkReadData (stream_no1, packet)
XLinkReadData (stream_no1, packet)
len1
Len1-data_len1
Write Ptr
Read Ptr
stream_no1

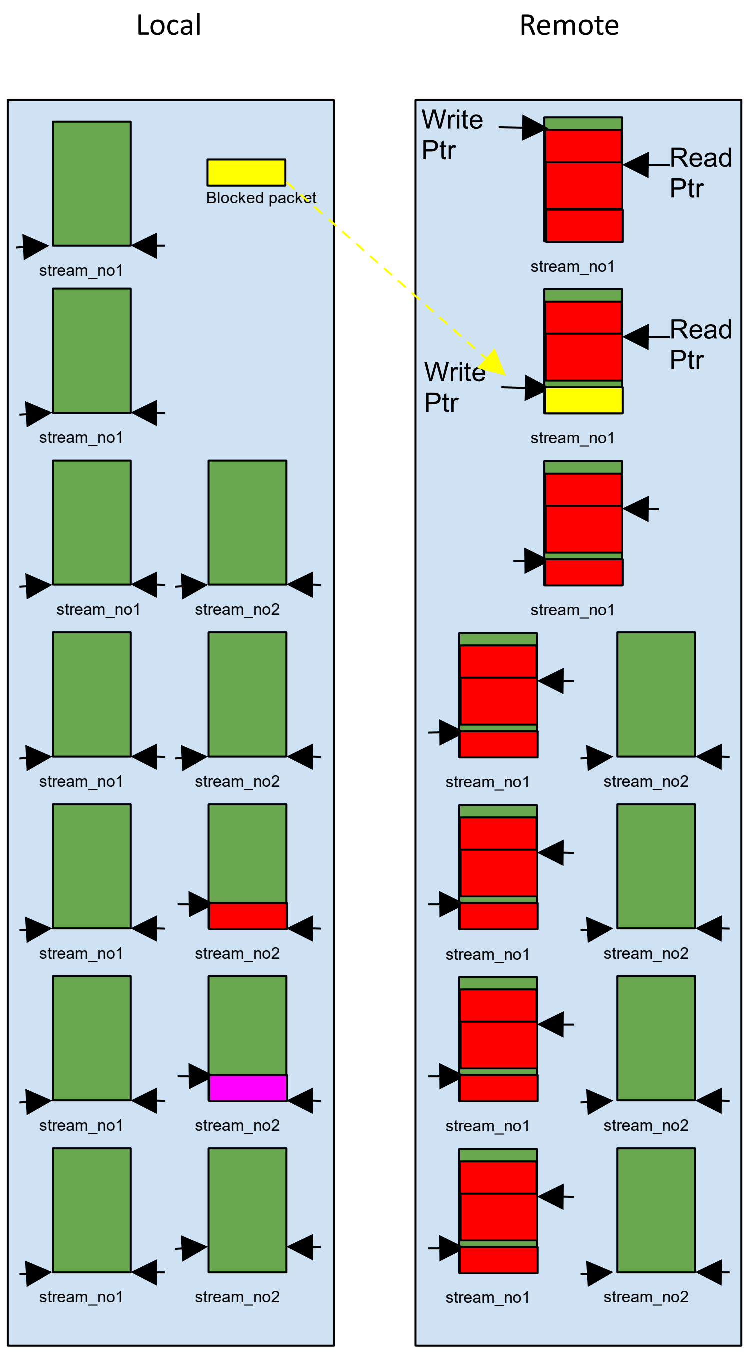
Local Calls

XLinkWriteData (stream_no1, data4, data_len4)

XLinkOpenStream (“name2”)

XLinkReadData (stream_no2, packet)

XLinkReleaseData (stream_no1)



Remote Calls

XLinkReleaseData (stream_no1)

XLinkOpenStream (“name2”)

XLinkWriteData (stream_no2, data5, data_len5)

3 Test App Setup

Build dependency:

- The Linux development `libusb-devel-1` library must be installed to be able to build the PC application.
- The PC application is only Linux compatible.

182 board

- Configure your boot switches for **USB bootmode** (2, 7 up – Myriad 2 | 2, 4, 5 up – Myriad X)
- Connect the USB cable to the 182. You should use USB3 port and cable.

USB stick

- Connect the USB stick to the PC.

Common steps

- Power up and connect the MDK development board.
- Verify with `lsusb` that the device is in bootloader and it's enumerated (03e7:2150 for Myriad 2 or 03e7:2485 for Myriad X).
- Verify that `libusb` is installed, if not run “`install libusbx-devel-1`” command with package manager(may differ depending on your operating system).
- Go to `/etc/udev/rules.d` and create a new file: `98-mcci.rules`.
- Add two lines to the file:
  - `SUBSYSTEM=="usb", ATTRS{idProduct}=="2150", ATTRS{idVendor}=="03e7", MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"`
  - `SUBSYSTEM=="usb", ATTRS{idProduct}=="f63b", ATTRS{idVendor}=="040e", MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"`
  - `SUBSYSTEM=="usb", ATTRS{idProduct}=="f63b", ATTRS{idVendor}=="03e7", MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"`
  - `SUBSYSTEM=="usb", ATTRS{idProduct}=="2485", ATTRS{idVendor}=="03e7", MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"`
- Check if you are in "users" group (`/etc/group`).
- `sudo udevadm control --reload-rules && sudo udevadm trigger`.
- `mdk/examples/HowTo/XLink/singleStream/myriad` -> compile the application with "**make all**".
- Go to `mdk/examples/HowTo/XLink/singleStream/pc` -> compile PC application with “**make**” the run it with “**./XLink**“ command.

Please note that it is important to set up the `udev` rules. You may think that you can use this communication with “`sudo`”, but without the above rules, the device will be detected as a modem and the kernel will send commands to the Myriad chip. This may break the communication or make it run much slower.

4 Usage instructions

4.1 API

1. **Call** `XLinkInitialize` **(from both nodes) to open USB communication and setup:**

   This initializes the protocol and starts USB communication. This fails if it is not able to open USB communication. If the Myriad chip is still starting up, the initialize might fail. This can be called within a loop until it results in `X_LINK_SUCCESS`. The function returns with `USB_LINK_UP` when communication has been established between both sides (a ping is sent and responded to).

2. **Call** `XLinkConnect` **(from both nodes) to connect to specific device:**

   Connects to a specific device and starts the Xlink Dispatcher. It will issue a ping to the remote side of the device.

3. **Call** `XLinkOpenStream` **(from both nodes) with the name and size of the stream:**

   The first call (whichever node calls it) creates a stream on both sides with the given size. The stream ID is returned. The second call simply returns the stream ID of the already created stream (by the first call).

4. **Use XLink as much as required:**
   a. `XLinkWriteData` **OR,**

      Writes the data to a circular FIFO buffer in the scope of the remote dispatcher.

   b. `XLinkReadData` **followed by** `XlinkReleaseData`

      XLinkReadData reads from the circular FIFO buffer in the scope of the local dispatcher.

      XLinkReleaseData releases the data read from the circular buffer. This should be called once the application is finished with the data.

5. **Call** `XLinkResetRemote` **to reset the remote and deinitialize.**

4.2 Constraints

- Maximum number of threads from which XLink can be called: `USB_LINK_MAX_STREAMS`.
- The receiver (reading the data) can only use a maximum of 1 thread for each stream, i.e. receiving from a particular stream cannot be multi-threaded.
- The data is aligned to 64 bytes (the cache line size in the current architecture).
- Stream name has a maximum length of 16 characters.
- There needs to be an end where __PC__ is defined.

4.3 Integration to custom app

To integrate the XLink component into a custom app, you should start with the test application: `examples/HowTo/Xlink/SingleStream`.

➢ **Steps for the Myriad chip part:**

- Add the following to the makefile:

```
MV_SOC_OS = rtems

USE_USB_VSC?=yes
ifeq ($(USE_USB_VSC),yes)
LEON_APP_URC_SOURCES+=$(wildcard common/components/USB_VSC/*leon/*.urc)
CCOPT+= -D'USE_USB_VSC'
ComponentList_LOS+=USB_VSC
MV_USB_PROTOS = protovsc2

else
LEON_APP_URC_SOURCES+=$(wildcard common/components/USB_CDC/*leon/*.urc)
ComponentList_LOS+=USB_CDC
MV_USB_PROTOS = protowmc
endif

ComponentList_LOS +=XLink
```

- Add an **rtems_config** to your application. Make sure that you have the following included in your config:

```
#define CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES 20
```

- Make sure you have the `AUX_CLK_MASK_USB_CTRL_SUSPEND_CLK` aux clock set to 20MHz
- Make sure that your application calls (at least) `XLinkInitialize()`

➢ **Steps for the PC part:**

- Make sure you add the following include directory to your app:

```
common/components/XLink/pc/
common/components/XLink/shared/
common/shared/include/
/usr/include/libusb-1.0
common/swCommon/pcModel/half \
-D'USE_USB_VSC' \
-D __PC__ \
-ggdb
```

- Make sure you add the following sources:

```
C_SOURCES += main.c\
common/components/XLink/pc/XLinkPlatform.cpp \
common/components/XLink/shared/XLink.c \
common/components/XLink/shared/XLinkDispatcher.c \
common/components/XLink/pc/usb_boot.c
CPP_SOURCES=\
      common/components/XLink/pc/UsbLinkPlatform.cpp
```

- All XLink functions will use a handler. The user must first configure the XLink component, telling it which communication protocol it will be using (which possibly means changing `XlinkHandler_t` structure by adding protocol type inside it). The initial values in the handler are important only for the PC version. You need to declare a handler and initialize it with the following:

```
XLinkHandler_t handler = {
      .devicePath = "/dev/ttyACM0",
      .devicePath2 = "/dev/ttyACM1",
};
```

- Make sure that your application calls (at least) `XLinkInitialize()`

➢ **Choosing the HW link type:**

Based on the defined communication protocol to be used, the XLink APIs will call the corresponding platform functions to perform those operations.

`XLink` supports the following three HW interfaces: `USB_VSC`, `USB_CDC` and `USB_LINK_JTAG`.

The functionality can be chosen at build-time by specifying a define both on the PC and the Myriad chip part:

- To use **VSC**, you need to:
  - Add the define `USE_USB_VSC`.
  - Add the `USB_VSC` component to the component list.
  - Define `MV_USB_PROTOS = protovsc2` in the Myriad Makefile.
- To use **CDC**, you need to:
  - Add the `USB_CDC` component to the component list.
  - Add the CDC urc file in the Myriad Makefile `common/components/USB_CDC/*leon/*.urc`).
  - Define `MV_USB_PROTOS = protowmc` in the Myriad Makefile.
- To use **jtag** you need to:
  - Add the `USE_LINK_JTAG` define to the app on both the PC and the Myriad chip sides.
  - After launching the application, create in the Myriad chip debug script the pipe: `pipe create USBLinkPipe -readsym mvUsbLinkTxQueue -writesym mvUsbLinkRxQueue -tcp 5678`.
  - Booting the Myriad chip is not supported in this mode, you need to use the debugger.

4.4 RTEMS Resources Used

4.4.1 Semaphores

- 1 for XLink Initialization (wait for init of other node).
- 1 for add Event (wait for event to be added before another event is added).
- 1 for notifying the dispatcher that there’s a new event.
- 1 per stream created (for threaded writing).
- 1 per thread used to add event (up to 8) (to enable `waitEventComplete`).
- 2 for `USB_VSC` component.
- Total: `5 + NUM_STREAMS + NUM_THREADS_ADDING_EVENTS`.

4.4.2 Threads

- 1 for event receiver.
- 1 for dispatcher.
- Total: 2.