# Pionway SDK

**为用户在 Pionway FPGA 模块与计算机之间建立一个灵活便捷的 USB 数据传输通道。**

Pionway SDK 是一个基于 Pionway FPGA 模块的软件开发套件，它在 FPGA 与计算机之间建立了一个灵活便捷的 USB 数据传输通道，并为用户提供了一套简单易用的软件编程接口，使用户在开发 USB 数据传输系统时更简单、高效，从而在很大程度上缩短用户的产品开发周期。

## 版权声明



## 修订记录

| 修订日期 | 修订内容 |
| --- | --- |
| 20170512 | Rev.A 最初版本 |
| 20220201 | Rev.B 修改了模板，增加了 SetTimeout()函数，增加了字节顺序转换表，BTPipe 改为 Block Pipe |

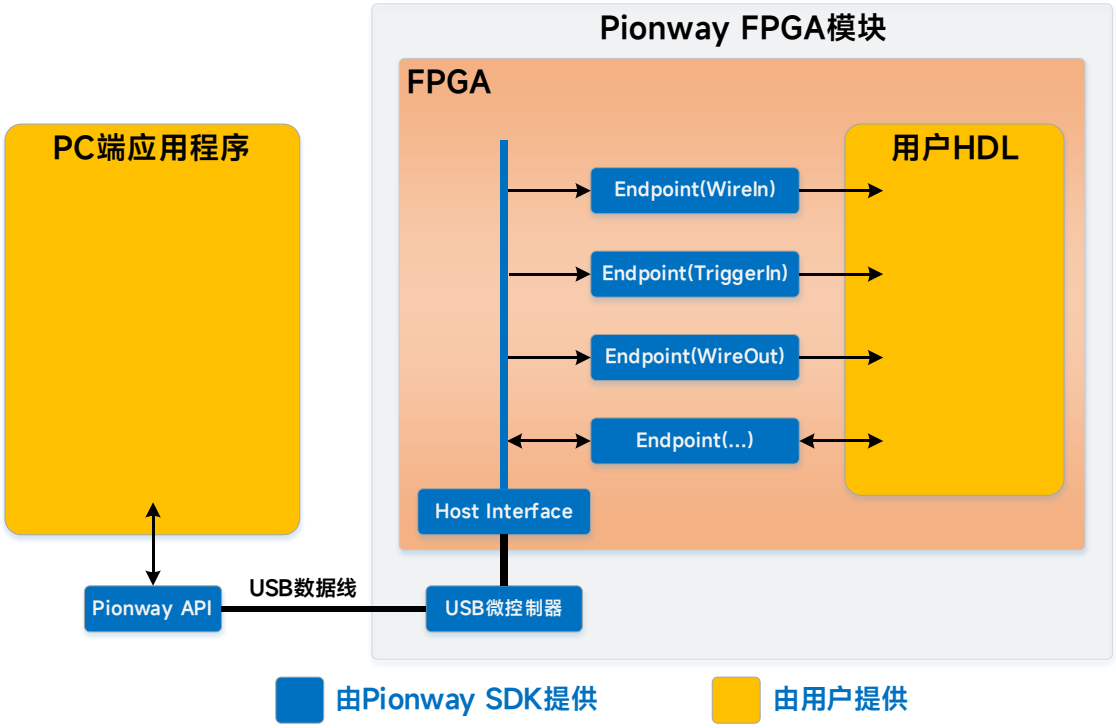
# 目录

# Pionway SDK 概述

Pionway SDK 是基于 Pionway FPGA 模块的软件开发套件，它包括 Pionway HDL、Pionway API、PionwayUSB 应用程序和相关文档，旨在使用户在开发 USB 数据传输系统时更简单、高效、省时、省成本。它在 FPGA 与计算机之间建立了一个灵活便捷的 USB 数据传输通道，并为用户提供了一套简单易用的软件编程接口。用户使用 Pionway SDK 提供的 API 和 HDL 模块分别编写上位机应用程序和 FPGA 程序，并配置 FPGA 和板上外设，就可以实现上位机与 Pionway FPGA 模块之间的通信。Pionway SDK 有以下几类功能：

- 设备发现和枚举
- FPGA 配置
- 使用 wire，trigger，pipe 进行 PC 与 FPGA 之间的通信
- 抽象出了一个基于 USB 接口的通用开发平台

## 系统拓扑



# Pionway HDL

建立在 FPGA 端的若干 HDL 模块，它使 FPGA 能与 USB 微控制器进行通信，并为用户提供编程接口。它由 Host Interface 和 Endpoint 等模块组成。

## Host Interface

使 USB 微控制器能与 FPGA 中的各种 Endpoint 实现通信的模块。

## Endpoint

Endpoint 是在 FPGA 端面向用户的接口，用于连接用户设计中的信号。这些 Endpoint 就像外部引脚，用户只需要将他想控制或传输的信号连接到这些 Endpoint 接口上，然后将这些 Endpoint 连接到一条总线上，并在该总线上放置一个 Host Interface 模块。最后，用户通过在 PC 端应用程序调用特定的 API 就能访问相应的信号。

用户可在任何时刻实例化额外的 Endpoint，以增加 Endpoint 的数量。这些 Endpoint 占用的 FPGA 资源非常少，因此对用户设计的影响非常小。

# Pionway Firmware

即 Pionway 固件，运行在 Pionway FPGA 模块的 USB 微控制器上，它提供 FPGA 与 PC 之间的通信管道。我们会在 Pionway 模块在出厂时下载最新的固件，并持续为用户提供更新。

# Pionway API

专门为 Pionway FPGA 模块设计的上位机应用程序编程接口，它提供了用于上位机与 Pionway HDL 通信的基本函数及一些基本的系统配置函数，用户软件通过调用这些函数可以很轻松地跟 Pionway FPGA 模块中的 FPGA 实现通信。用户可以在 Windows 平台使用 Pionway API，目前支持的语言为 C 和 C++。另外，用户也可以利用第三方软件，如 Matlab 或者 LabVIEW 调用 Pionway 的动态链接库（DLL）。

# Pionway SDK 使用简介

Pionway SDK 的主要作用是在 Pionway FPGA 模块与 PC 之间建立一个便利且有效的数据通道，用户可以很容易地将现有的或新的 FPGA 设计与其建立连接。更重要的是，Pionway SDK 使用户不需要考虑物理接口（USB）部分的具体实现，这在很大程度上减少了用户在产品开发过程中的风险和时间。

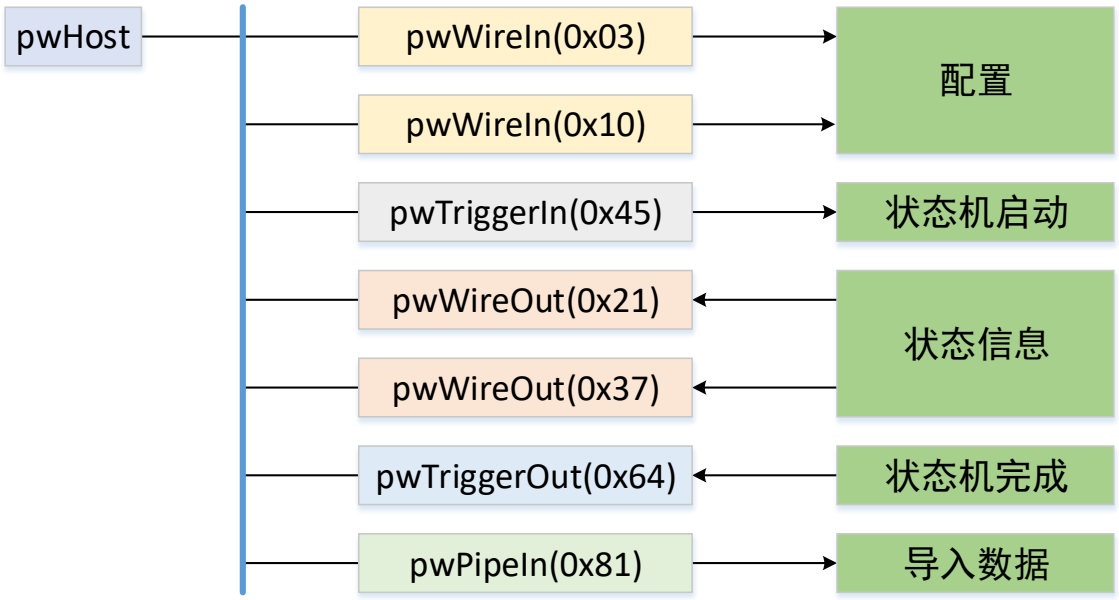
Pionway SDK 在用户的 FPGA 设计中引入了“Endpoint”的概念。一个 Endpoint 是 FPGA 中的一串内部互连，它以某种方式从或向 PC 传输数据。在很多情况下，Endpoint 可以通过用户想要传输的现有信号创建。对于其它情况，用户需要为了实现特定的数据传输而创建 Endpoint。

USB 接口的所有操作都是由上位机发起的。FPGA 端生成上传信号或数据后需要上位机主动查询来接收。

## Endpoint

在 Pionway HDL 中，Endpoint 可以是 Wire、Trigger 或 Pipe，并且数据传输方向可以是 In 或 Out。通过定义，数据传输方向是以 FPGA 为视角，所有向用户设计输入数据的 Endpoint 都是 In 方向，而所有由用户设计输出数据的 Endpoint 都是 Out 方向。在一个设计中，所有的 Endpoint 都是使用 Pionway FPGA HDL 库来实例化，并且共用一个 Host Interface。

下图是一个 FPGA 示例程序的结构框图。pwHost 仅需要实例化一次，并且连接到一些 FPGA 的外部引脚以及一条被所有 Endpoint 共享的总线。这条总线提供了所有 Endpoint 与 Host Interface 之间的数据传输通道。




每个 Endpoint 的实例都有一个关联的地址（括号中所示），从而能够独立于其它 Endpoint 被访问。在这个示例中，两个 Wire In Endpoint 用于系统的配置，两个 Wire Out Endpoint 用于向 PC 传回状态信息，一个 Trigger In Endpoint 用于启动状态机，一个 Trigger Out Endpoint 用于指示状态机的完成，一个 Pipe In Endpoint 用于向 FPGA 导入数据。

下表是对这四种 Endpoint 类型的概述，后文有对它们的详细介绍。

| Endpoint | 同步/异步 | 描述 |
|---|---|---|
| Wire In | 异步 | PC 向用户 HDL 输入一个信号状态。<br>（例如：虚拟按钮或开关） |
| Wire Out | 异步 | 用户 HDL 向 PC 输出一个信号状态。<br>（例如：虚拟 LED 或数码管的显示） |
| Trigger In | 同步 | PC 向用户 HDL 输入一个可以和特定时钟同相的单脉冲信号。（例如：用于启动状态机的按钮） |
| Trigger Out | 同步 | 用户 HDL 通知 PC 有一个特定的事件发生。<br>（例如：状态机发出 Done 信号后 PC 上弹出窗口告知用户可以开始数据传输） |
| Pipe In | 同步 | PC 向用户 HDL 输入批量数据。<br>（例如：存储器的数据下载，流数据） |
| Pipe Out | 同步 | 用户 HDL 向 PC 输出批量数据。（例如：存储器的数据上传，读取计算结果） |
| Block Pipe In | 同步 | 同 Pipe In，增加对数据块流控信号。 |
| Block Pipe Out | 同步 | 同 Pipe Out，增加对数据块流控信号。 |

## Wire

Wire 是 PC 和 HDL Endpoint 之间的异步连接。Wire In 是用户 HDL 的输入，Wire Out 是用户 HDL 的输出。

Wire 可用于连接 LED、数码管、按钮、拨码开关等设备。这些设备通常采用非同步设计，用来传达一些内部信号的当前状态。

为了提高效率，所有的 Wire In 都是同时被捕获和更新的，Wire Out 也是如此。当然，这并不意味着它们是同步传输，而只是在同一时间被更新。因此，32 个 Wire In（或 32 个 Wire Out）是同时传输的。

## Trigger

Trigger 是 PC 和 HDL Endpoint 之间的同步连接。Trigger In 是用户 HDL 的输入，Trigger Out 是用户 HDL 的输出。Trigger 用于启动或发送一个事件发生信号，如一个状态机的开始或结束。

作为用户 HDL 的输入信号，Trigger In 会创建一个单脉冲信号。其中，同步时钟由用户决定，而 HDL 模块负责处理跨时钟域问题。

作为用户 HDL 的输出信号，Trigger Out 会在检测到信号上升沿时将其锁存住，等待 PC 端来查询。这里的上升沿是指从一个时钟周期到下一个时钟周期的电平跳变，不包括毛刺。

## Pipe

Pipe 是 PC 和 HDL Endpoint 之间的同步连接。Pipe 用来传输一系列的字节数，它最常用的是上传或下载存储器数据，也可用于传送流数据。

从用户 HDL 的视角来看，Pipe 总是主设备。也就是说，PC 负责处理 Pipe In 和 Pipe Out 的一切事务。另外，Pipe 必须在 Endpoint 的时钟频率（100.800MHz）下工作。为了跨时钟域的可靠性，建议用户使用 FIFO 作为数据缓冲。用户可使用 Xilinx 的 Core Generator 工具生成合适的 FIFO。

Pipe 的数据入口通常来自从设备（在 Pionway 软件中指用户 HDL），因此，用户需要使用 Trigger 实现数据块传输的有效协议。

Pipe 的数据传输速率取决于 PC 的硬件配置。实际测试的数据传输速率超过 300MB/s。详见后文的性能说明。

## Block Pipe

Block Pipe（Block Pipe）与标准的 Pipe 非常相似，唯一的区别是：Block Pipe 在标准 Pipe 基础上增加了对数据块的流控操作信号。数据块大小可配置为 16 ~ 16384 字节（USB SuperSpeed 模式下）。

FPGA 通过控制“READY”信号（连接到 USB 微控制器）的电平实现节流数据传输，这使得 FPGA 可以在数据可用或待处理之前停止数据传输。

Block Pipe 与标准的 Pipe 有接近的数据传输速率，流控信号使它有了更广泛的应用，并且可以减轻系统的额外开销，缩小控制环路，从而提高系统的实际性能。

# 性能说明

Pionway SDK 包括 FPGA 内部的 HDL 模块、USB 微控制器中的固件和 PC 端的 API。
为了在具体应用中获得最优越的性能，您需要对用到的组件有着深刻的理解，并且了解影响系统性能的各种因素。通过遵循并执行一些简单的策略，您的应用程序将会拥有最高性能并且仍然受益于 Pionway SDK 的易用性和灵活性。

测试平台：Intel Core i7-4790S CPU，Z97 主芯片组，Windows 7 旗舰版-64 位操作系统。影响 USB 性能的因素有很多，包括主板制造商和型号、驱动版本，PC 的负载等。

## Wire 和 trigger

Wire 和 Trigger 提供了 FPGA 与 PC 之间最基本的通信方式。从性能角度看，Wire 每秒可以读写数百次。不论用户是否全部需要，所有的 Wire In 都是同时写入，同理所有的 Wire Out 也是同时读取。

启动 Trigger In 是一个非常快的操作，系统每秒可以操作超过 1000 次(每次调用只写一个 Trigger 的情况下)。Trigger Out 的更新类似于 Wire Out 的读取，所有的 Trigger Out 同时读取。

### Wire 和 trigger 性能测试

CPS = Calls Per Second

| API 调用 | USB 3.0（CPS） |
|---|---|
| UpdateWireIns | 1000+ |
| UpdateWireOuts | 1000+ |
| ActivateTriggerIn | 1000+ |
| UpdateTriggerOuts | 1000+ |

# Pipe（批量传输）

Pipe 是发送或接收批量数据最快的方式。由于每次传输过程中的额外开销，使用长字节传输可以获得最好的性能：每执行一次 Pipe 传输，需要进行多层的操作（固件层、API 层和操作系统层）。因此，建议尽可能一次传输较大的数据量。这通常意味着，您需要在 FPGA 中使用大容量缓冲器，并且建议使用外部 RAM 存储器。

低延迟、高带宽对任何一种协议来说都是一种特别的挑战，USB 也不例外。在这种情况下，这两个目标存在矛盾：试图执行频繁操作和获得高带宽。问题是，建立每次传输的额外开销会占用可用于执行数据传输的时间。

需要注意的是 Windows 不是实时操作系统。它是一种较复杂的操作系统，在任何时候它都有许多拥有更高优先级的进程需要处理。然而，通常情况下，许多简单的操作（如移动窗口）都会急剧降低数据传输带宽。对于依赖传输带宽的应用程序，在设计数据缓冲时，需要仔细考虑。

## 测试性能

写测试：WriteToPipeIn；读测试：ReadFromPipeOut

| 传输长度 | USB 3.0 | |
|---|---|---|
| | 写（MiB/s） | 读(MiB/s) |
| 128 B | 0.04 | 0.02 |
| 256 B | 0.08 | 0.05 |
| 512 B | 0.16 | 0.08 |
| 1.0 kB | 0.24 | 0.2 |
| 4.0 kB | 0.97 | 0.65 |
| 16.0 kB | 5.2 | 3.13 |
| 64.0 kB | 20.8 | 12.5 |
| 256 kB | 62.5 | 41.6 |
| 1.0 MB | 167 | 125 |
| 4.0 MB | 267 | 250 |
| 8.0 MB | 320 | 320 |
| 16.0 MB | 333 | 326 |
| 32.0 MB | 347 | 337 |
| 64.0 MB | 351 | 356 |

## Block Pipe

即 Block-Throttled Pipe，它拥有与标准的 Pipe 相当的性能，不同的是，FPGA 可以进行块级节流数据传输。块大小（block size）用户可以修改，当块大小为 16384Byte 时可获得最高性能。

Block Pipe 是使用小容量缓冲器获得高性能的一种效率更高的方式，因为 FPGA 可以在一个较低的水平协调传输，并且在为每一个缓冲块建立新的传输时不会产生显著的额外开销。

### 测试性能

写测试：WriteToBlockPipeIn；读测试：ReadFromBlockPipeOut。传输长度：16MiB。

| 块大小 (Byte) | USB 3.0 | |
|---|---|---|
| | 写（MiB/s) | 读(MiB/s) |
| 16 | 112 | 68 |
| 64 | 222 | 172 |
| 128 | 258 | 225 |
| 256 | 296 | 275 |
| 512 | 307 | 301 |
| 1024 | 310 | 320 |
| 2048 | 313 | 326 |
| 4096 | 320 | 326 |
| 8192 | 320 | 333 |
| 16384 | 326 | 333 |

## 同步传输

Pionway SDK 不支持 USB 同步传输。虽然同步传输有固定的保证带宽，这对一些要求实现一定性能的系统非常有用，但是这种带宽保证需要付出很大的代价：同步传输不能拥有与 USB 批量传输相同等级的错误检测和错误校正。此外，这种带宽保证只适用于总线而不包括操作系统的性能。

对于 USB 批量传输，在数据传输过程中，当有错误发生时，上位机会请求重新传输丢失的数据包，并且上位机也会重新建立传输，以保证所有数据的传输顺序是正确的。

对于同步传输来说，带宽和延迟要求比数据的准确性更重要。因此，在数据传输过程中可能会丢失一些数据。同步传输是为支持某些对时间要求很高、数据量很大应用要求而提出的，例多媒体设备、音频设备等。然而，如果上位机太忙或者因为某些原因传输中断了，视频丢失了几帧或者音频丢失了几毫秒都将被认为是可接受的。

# API（应用程序编程接口）

Pionway API 包含了能通过 USB 接口实现通信的函数，这些函数经过特别设计，仅适用于 Pionway FPGA 模块，并且用于与模块中的 FPGA 建立连接。Pionway API 还提供了一些函数，用于与 Pionway HDL 模块（Wire，Trigger，Pipe）直接建立连接。这样的抽象化，虽然牺牲了硬件接口的一些灵活性，却为 Pionway 软件带来了很大的灵活性和便利性，从而可以在很大程度上减少用户的产品开发周期（使用户软件与 FPGA 建立连接的开发时间以及相关知识的学习时间）。

Pionway API 以动态链接库（DLL）形式提供，用户可以在 Windows 平台使用，并且支持 C 和 C++。

## API 参考手册

本用户手册提供的 API 文档给出了关于如何使用 Pionway API 的总体概述，关于具体的调用方法和参数的详细信息请查看 API 参考手册。

可以在以下网址找到 API 参考手册：https://pionway.com/products/pionway-sdk.html。

## 示例程序

通常情况下，要学会如何使用一个编程接口的最好方法就是看它的应用实例。我们建议您去仔细研究我们的所有示例程序，以了解如何使用 Pionway SDK 构建应用程序。如果您在设计中遇到问题，请试着重新查看我们的示例程序。

# 机制

Pionway API 以动态链接库形式提供，您需要将它包含在您的应用程序中。该动态链接库的接口是 C，但我们提供的是 C++封装，这让它看起来好像是您的应用程序中的一个 C++类。

目前，该库只包含了一个类，您需要在您的代码中实例化。如下表所示。

| 类 | 描述 |
| --- | --- |
| CPionway | 用于查找设备、配置设备以及与设备通信的基本类。该类分为 3 组函数：设备管理、设备交互、设备配置和 FPGA 通信。 |

# 加载库

Pionway API 是一个动态链接库，它必须在您的应用程序运行时被加载。

## 静态加载

以 Microsoft Visual Studio 为例，将 PionwayDLL.h、Pionway.lib 和 Pionway.dll 这三个文件拷贝至工程目录下（.vcxproj 所在目录），然后在工程中包含 PionwayDLL.h 即可。

# CPionway 类

它是 Pionway API 的主体，可分为 3 组函数：设备交互、设备配置和 FPGA 通信。
在一个典型应用中，您的软件需要执行以下步骤：

1. 创建一个 CPionway 类。
2. 使用设备交互函数，查找将要进行通信的 XMS 设备（Pionway FPGA 模块）并打开。
3. 使用 ConfigureFPGA(...)函数为 FPGA 下载配置文件。
4. 使用 FPGA 通信函数，实现与 FPGA 通信。

## 设备交互

尽管 Pionway API 封装了硬件接口的底层细节，但事实上，该模块仍是一个 USB 设备，因此必须遵守 USB 设备的相关规范。这些函数提供了查找所有已连接的 Pionway 设备的方法，查询每个设备的特定信息，并最终打开某个特定设备用于通信。下表是对这些函数的概述，为简洁起见，我们已将参数省略，详情请参阅 API 参考手册。

| 方法 | 描述 |
|---|---|
| GetDeviceCount | 获取已连接的 Pionway 设备数量，包括未打开的所有设备。 |
| GetDeviceListModel | 获取一个已连接的 Pionway 设备的型号。 |
| GetDeviceListSerial | 获取一个已连接的 Pionway 设备的序列号。 |
| OpenBySerial | 打开一个特定序列号的 Pionway 设备用于通信。 |
| GetDeviceInfo | 获取一个已连接的 Pionway 设备的所有信息，包括设备 ID，设备序列号，产品型号等。 |
| SetTimeout | 用于设置 API 的操作超时时间，默认是 10000ms。 |
| Close | 关闭一个 Pionway 设备 |
| IsOpen | 查询 Pionway 设备的打开情况 |

## 设备配置

一旦一个可用设备已被打开，这些函数可用于配置设备的相关特性，如下表所示。

| 方法 | 描述 |
| --- | --- |
| SetDeviceID | 允许用户设置设备 ID。 |
| ConfigureFPGA | 下载配置文件到 FPGA。 |
| FlashEraseSector | 擦除用户 Flash 中的一个扇区。 |
| FlashWrite | 向用户 Flash 写入数据。 |
| FlashRead | 从用户 Flash 读取数据。 |

## FPGA 通信

一旦 FPGA 已被配置，应用程序和 FPGA 之间的通信将通过一些函数完成。FPGA 与 Pionway 设备上的 USB 微控制器直接连接。这些函数通过该连接进行通信，并要求在配置 FPGA 时实例化 Pionway HDL 模块 pwHost。

下表是对这些函数的简要说明，后面将会对这些 API 与 Pionway HDL 模块的通信方法做详细描述。

| 方法 | 描述 |
| --- | --- |
| IsPionwayEnabled | 检查 pwHost 是否在 FPGA 配置中已被实例化。 |
| UpdateWireIns | 同时更新所有 Wire In 的值（PC 向 FPGA）。 |
| SetWireInValue | 设置一个 Wire In 的值。需要后续调用 UpdateWireIns。 |
| GetWireInValue | 获取一个 Wire In 的值。 |
| UpdateWireOuts | 同时捕获所有 Wire Out 的值（FPGA 向 PC）并在内部存储这些值。 |
| GetWireOutValue | 获取一个 Wire Out 的值。需要在之前调用 UpdateWireOuts。 |
| ActivateTriggerIn | 触发某个特定的 Trigger In（PC 向 FPGA）。 |
| UpdateTriggerOuts | 同时获取所有 Trigger Out 的值（FPGA 向 PC）并记录被触发的 Trigger Out（自从上次查询开始）。 |
| IsTriggered | 查询一个 Trigger Out 是否被触发（自上次调用 UpdateTriggerOuts 起）。 |
| WriteToPipeIn | 向 Pipe In 写入数据（字节数组）。 |
| ReadFromPipeOut | 从 Pipe Out 读取数据（字节数组）。 |
| WriteToBlockPipeIn | 向 Block Pipe In 写入数据。 |
| ReadFromBlockPipeOut | 从 Block Pipe Out 读取数据。 |

# 系统 Flash

所有 Pionway 设备都有一个与 USB 微控制器连接的非易失性 Flash 存储器，用于存储设备固件和用户数据。Pionway API 包含用于控制及访问该 Flash 存储器的函数。该 Flash 的可用存储空间和地址与设备相关。本手册中关于 Flash 的描述适用于所有 Pionway 设备。

# API 通信

三种类型的 Endpoint 类型（Wire，Trigger，Pipe）提供了一种 PC 与 FPGA 通信的途径，每种类型的 Endpoint 分别适用于特定的数据传输类型，并且都有各自的用法和规则。

## 阻塞 API

所有的 Pionway API 函数都是阻塞类型。这意味着，每次调用会在返回之前结束。因此，您可以放心，如果连续执行两个用于更新 FPGA 上的寄存器的 API 调用，第一次调用的更新会在第二次调用开始之前完成。

## Wire

Wire 用于在 PC 与 PFGA 之间传输异步信号状态。Host Interface 最多支持连接 32 个 Wire In 和 32 个 Wire Out。为了节省带宽，所有的 Wire In 和 Wire Out 会在同一时间更新。

上位机（PC）调用 UpdateWireIns()，所有 Wire In 会在同一时间更新（向 FPGA 写入）。在此之前，应用程序通过调用 SetWireInValue()设置 Wire In 的新值。SetWireInValue()只是简单地在 API 内部的更新 Wire In 的值。然后，UpdateWireIns()将这些值传送至 FPGA。

同样地，上位机调用 UpdateWireOuts()，所有 Wire Out 会在同一时间更新（从 FPGA 读取）。该调用读取所有 Wire Out 的值，并将这些值存储在 API 内部中。然后通过调用 GetWireOutValue()读取这些值。

## Trigger

Trigger 用于在 PC 与 FPGA 之间传输单一事件。Trigger In 用于上位机在任意 FPGA 时钟下向 FPGA 传送单脉冲触发信号。Trigger Out 用于 FPGA 向上位机传送触发信号或其它单一事件指示信号。

Trigger 的读取和更新方式类似于 Wire。所有的 Trigger In 会在同一时间更新（向 FPGA 写入），而所有的 Trigger Out 会在同一时间读取（从 FPGA 读取）。

上位机调用 UpdateTriggerOuts()从 FPGA 读取 Trigger Out 的值。随后调用 IsTriggered()来进行检测，如果自上次调用 UpdateTriggerOuts()起该 Trigger Out 被触发，则返回真。

## Pipe

Pipe 适用于多个字节数据的同步通信。在 Pipe In 和 Pipe Out 中，上位机（PC）总是主设备。因此，FPGA 必须在任意时刻都能接收或发送数据。而 Wire、Trigger 和 FIFO 可以用来协调传输过程，使其更加灵活。

当上位机调用 WriteToPipeIn()使数据从上位机被写入至 Pipe In 时，设备驱动会根据底层协议需要打包数据。一旦数据传输开始，它会持续到数据传输完成为止，因此，在此之前，FPGA 必须做好接收所有数据的准备。

当上位机调用 ReadFromPipeOut()使数据由上位机从 Pipe Out 读取时，设备驱动将再次根据需要打包数据。数据传输会从开始持续到数据传输完成为止，因此，在此之前，FPGA 必须按照要求为 Pipe Out 准备好数据。

### 字节顺序

Pipe 的数据在 PC 端（API）是通过 USB 以 8 位字传输的，而 FPGA 端则是 32 位字。建议在 FPGA 端做 Byte Swap 操作。

| PC | | FPGA |
|---|---|---|
| Byte0 | Bit 0 | Bit 24 |
| | Bit 1 | Bit 25 |
| | Bit 2 | Bit 26 |
| | Bit 3 | Bit 27 |
| | Bit 4 | Bit 28 |
| | Bit 5 | Bit 29 |
| | Bit 6 | Bit 30 |
| | Bit 7 | Bit 31 |
| Byte1 | Bit 0 | Bit 16 |
| | Bit 1 | Bit 17 |
| | Bit 2 | Bit 18 |
| | Bit 3 | Bit 19 |
| | Bit 4 | Bit 20 |
| | Bit 5 | Bit 21 |
| | Bit 6 | Bit 22 |
| | Bit 7 | Bit 23 |

| PC | | FPGA |
|---|---|---|
| Byte2 | Bit 0 | Bit 8 |
| | Bit 1 | Bit 9 |
| | Bit 2 | Bit 10 |
| | Bit 3 | Bit 11 |
| | Bit 4 | Bit 12 |
| | Bit 5 | Bit 13 |
| | Bit 6 | Bit 14 |
| | Bit 7 | Bit 15 |
| Byte3 | Bit 0 | Bit 0 |
| | Bit 1 | Bit 1 |
| | Bit 2 | Bit 2 |
| | Bit 3 | Bit 3 |
| | Bit 4 | Bit 4 |
| | Bit 5 | Bit 5 |
| | Bit 6 | Bit 6 |
| | Bit 7 | Bit 7 |

### BT-Pipe

BT-Pipe 等同于拥有可指定传输块大小（block size）的 Pipe。FPGA 根据用户通过参数指定的传输块大小（16，32...16384 字节）发送或接收数据。当 USB 工作于全速（Full Speed）模式时，块大小不能大于 64 字节；当 USB 工作于高速（High Speed）模式时，块大小不能大于 1024 字节；当 USB 工作于超高速（Super Speed）模式时，块大小不能大于 16384 字节。

由于 FPGA 可以通过拉低 EP_READY 信号（使 EP_READY 信号失效）来暂时中断传输，因此 BT-Pipe 的读写操作可能会因超时而失败。

## 32 位和 64 位架构

Pionway API 分为 Windows 32 位和 64 位架构。

如果您的 PC 安装了 32 位版本的 Windows 操作系统，那么，您只需要 32 位的 API，因为 32 位的 Windows 操作系统不能运行 64 位的 API。

如果您的 PC 安装了 64 位版本的 Windows 操作系统，那么，您的应用程序既可以使用 32 位的 API，也可使用 64 位的 API。您选择的 Pionway API 应该符合您正在使用的应用程序架构。通常情况下，32 位应用程序安装在“Program Files (x86)”目录下，而 64 位应用程序安装在“Program Files”目录下。

| | Windows（32 位） | Windows（64 位） |
|---|---|---|
| **32 位应用程序** | 32 位 API | 32 位 API |
| **64 位应用程序** | - | 64 位 API |

# HDL 模块

使用 Pionway API 与 FPGA 通信，需要在您的顶层 HDL 实例化至少一个 Pionway HDL 模块。这些模块可以方便快捷地加入到现有的或新的设计中，并负责处理它与 Pionway API 通信的复杂工作。

Host Interface 是使 USB 微控制器能与 FPGA 中的各种 Endpoint 实现通信的模块。FPGA 中的一些引脚与 USB 微控制器连接，而 Host Interface 则直接与这些引脚连接。它是 Pionway API 到用户设计的入口点。

Endpoint 与 Host Interface 发出的一条共享的控制总线连接。这条内部总线用于将所有来自和去往（即 In 和 Out）Host Interface 的 Endpoint 连接在一起。Pionway API 通过 Endpoint 的地址来选择与之通信的 Endpoint，因此，每个 Endpoint 必须拥有自己唯一的地址才能正常工作。

## Endpoint 类型

Pionway 软件支持三种基本的 Endpoint 类型：Wire，Trigger，Pipe。每个 Endpoint 都可以是输入（从 PC 到 FPGA），也可以是输出（从 FPGA 到 PC）。每个端点类型都有一个特定的地址范围，并且必须正确使用。这些地址是在您实例化 Endpoint 时指定的。

Endpoint 在您的 HDL 设计中实例化，并连接到 pwHost 的目标端口。每个 Endpoint 还有一个或多个端口用于连接您设计中的各种信号，这取决于具体的 Endpoint。

Endpoint 通过一条共享的总线连接到 Host Interface 上。为了正确地在 PC 与目标 Endpoint 之间传送信号，每个 Endpoint 必须被分配一个唯一的 8 位地址。由于性能原因（为了最小化 USB 事务），每个 Endpoint 类型都分别被分配了地址范围，如下表所示。当给您的 Endpoint 分配地址时，请务必遵循这些范围。

| Endpoint 地址 | 地址范围 | 同步/异步 | 数据类型 |
|---|---|---|---|
| Wire In | 0x00 - 0x1F | 异步 | 信号状态 |
| Wire Out | 0x20 - 0x3F | 异步 | 信号状态 |
| Trigger In | 0x40 - 0x5F | 同步 | 单触发信号 |
| Trigger Out | 0x60 - 0x7F | 同步 | 单触发信号 |

| | | | |
|---|---|---|---|
| Pipe In | 0x80 - 0x9F | 同步 | 多字节数据 |
| Pipe Out | 0xA0 - 0xBF | 同步 | 多字节数据 |
| Block Pipe In | 0xC0 - 0xDF | 同步 | 多字节数据 |
| Block Pipe Out | 0xE0 - 0xFF | 同步 | 多字节数据 |

### Endpoint 地址

Endpoint 地址是通过 Endpoint 实例中的一个 8 位附加输入端口分配的。对每种类型的 Endpoint 进行实例化的示例将在以下章节展示。

## Endpoint 数据宽度

对于接口为 USB 3.0 的 Pionway 设备，Endpoint（包括 Wire、Trigger 和 Pipe）的数据宽度为 32 位。

## Host Interface 时钟频率

Host Interface 是来自上位机的从接口。对于接口为 USB 3.0 的 Pionway 设备，它工作在一个固定的时钟频率，该时钟频率为 100.800 MHz（时钟周期为 9.92 ns）。

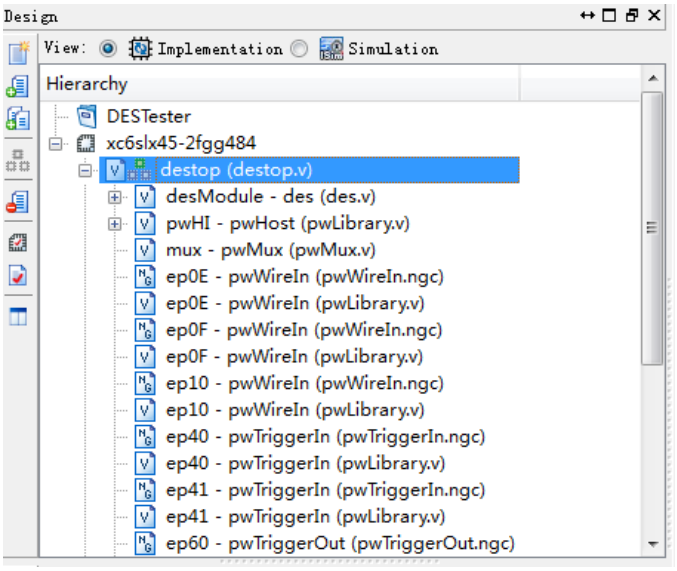
## 使用 Pionway HDL 模块建立工程

Pionway HDL 模块是以预综合文件形式提供，它会包含在您的设计中。下表列出了这些文件，并描述了它们包含的内容。默认情况下，这些文件被安装在“D:\Program Files\Pionway\PionwayUSB\PionwayHDL”目录下。在这个目录下有多个子目录，包含了为不同设备建立的 HDL 模块。请选择适用于您的设备的 Library 和 NGC 文件。

| 文件名 | 描述 |
|---|---|
| pwLibrary.v | 包含所有模块的 Verilog 文件，用于 Verilog 工程。 |
| pwLibrary.vhd | 包含所有模块的 VHDL 文件，用于 VHDL 工程。 |
| pwCore.ngc | Host Interface 的预综合模块。 |
| pwWireIn.ngc | Wire In 的预综合模块。 |
| pwWireOut.ngc | Wire Out 的预综合模块。 |
| pwTriggerIn.ngc | Trigger In 的预综合模块。 |
| pwTriggerOut.ngc | Trigger Out 的预综合模块。 |
| pwPipeIn.ngc | Pipe In 的预综合模块。 |
| pwPipeOut.ngc | Pipe Out 的预综合模块。 |
| pwBlockPipeIn.ngc | BT-Pipe In 的预综合模块。 |
| pwBlockPipeOut.ngc | BT-Pipe Out 的预综合模块。 |

Host Interface 是由两部分组成：一个已被预综合的核心部分和一个封装部分（pwLibrary.v 或 pwLibrary.vhd）。其中，封装部分包含了核心部分和连接 FPGA 引脚所需的 IOB。

当您做一个新的设计时，您需要将 pwLibrary.v 或 pwLibrary.vhd 文件复制到包含您的其它源文件的目录下，并将其添加到您的工程中。该文件会像您的其它模块一样被综合，只是对于那些已被预综合的模块来说，它的 HDL 代码大多只是一个声明。当它被正确添加到一个项目中时，项目导航器会像下图一样列出源文件：




您也可以将您用到的模块对应的预综合文件（*.ngc）复制到您的工程目录中。对于没用到的模块，您不需要复制相应的文件。为了建立 FPGA 配置文件，这些 NGC 文件会在 Xilinx 工具进行转换步骤时用到。

## FPGA 资源需求

在设计 Pionway HDL 模块时，我们尽可能地使它们少占用 FPGA 的内部资源。对于每个模块的资源需求如下表所示。请记住，这些需求是在 Endpoint 的所有位都使用的情况下，即各模块的所有地址都被使用。在很多情况下，布局布线工具将优化并删除未使用的组件。

| 资源 | Slice FF | 4-in LUT | Block RAM |
|---|---|---|---|
| Host Interface | 33 | 49 | 0 |
| Wire In | 16 | 14 | 0 |
| Wire Out | 8 | 5 | 0 |
| Trigger In | 32 | 21 | 0 |
| Trigger Out | 27 | 15 | 0 |
| Pipe In | 9 | 10 | 0 |

| | | | |
|---|---|---|---|
| Pipe Out | 0 | 6 | 0 |
| BT-Pipe In | 119 | 70 | 0 |
| BT-Pipe Out | 58 | 44 | 0 |

# MUX

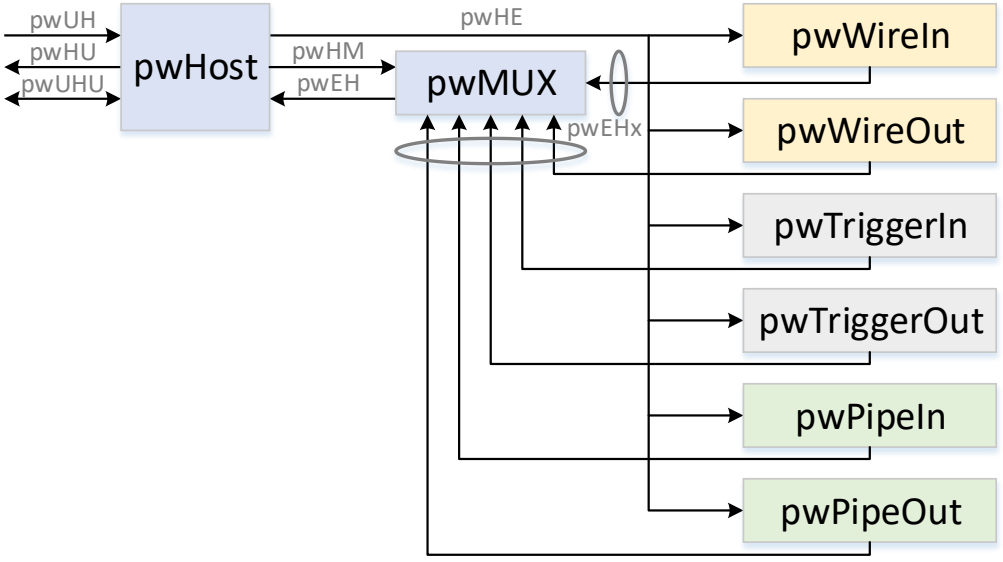
所有 Endpoint 通过 pwEMx 总线连接到挂在 Host Interface 上的 MUX。MUX 是在 pwLibrary.v 或 pwLibrary.vhd 文件中以一个参数化辅助模块形式提供。请参考我们提供的示例程序，以了解如何实例化该模块。

# Host Interface

Host Interface 是 USB 微控制器与 FPGA 实现通信的入口，相当于 Endpoint 连接上位机的接口。它包含让 USB 微控制器与您设计中的所有 Endpoint 通信的逻辑。在任何设计中，必须实例化一个唯一的 Host Interface 才能使用 Endpoint。

Host Interface 是在您的设计中一起被综合的一个唯一模块。它包含一个已被预综合的核心部分和连接 FPGA 引脚所需的 IOB。

以下框图阐述了 pwHost、MUX 及各种 Endpoint 之间的结构关系。



## pwHost

在您的设计中，为了使用 Endpoint，pwHost 必须被实例化，并且仅需一次。以下信号必须连接到 FPGA 引脚（与 USB 微控制器连接）。对于特定产品的引脚位置列表，请参阅该产品的用户手册。

| 信号 | 方向 | 描述 |
|---|---|---|
| pwUH[5：0] | 输入 | Host Interface 的输入信号。 |
| pwHU[3：0] | 输出 | Host Interface 的输出信号。 |
| pwUHU[31：0] | 输入/输出 | Host Interface 的输入/输出信号。 |

pwHost 的剩余端口连接共享总线 pwMUX 及 Endpoint。这些信号被统称为目标接口总线。

| 信号 | 方向 | 描述 |
|---|---|---|
| pwMH[68：0] | 输入 | 连接 pwMUX 的数据信号。 |
| pwHM[1: 0] | 输出 | 连接 pwMUX 的控制信号。 |
| pwHE[99：0] | 输出 | 连接 Endpoint 的控制信号。 |
| pwClk | 输出 | Host Interface 时钟的缓冲复制。 |

使用 VHDL 或 Verilog 对 pwHost 实例化都很简单。您可在您的顶层 HDL 中使用下面的模板。

Verilog 实例化：

```
pwHost pwHI (.pwUH(pwUH), .pwHU(pwHU), .pwUHU(pwUHU),
.pwMH(pwMH), .pwHM(pwHM), .pwHE(pwHE), .pwClk(pwClk))
```

VHDL 实例化：

```
pwHI: pwHost port map (pwUH => pwUH, pwHU => pwHU, pwUHU => pwUHU,
pwMH => pwMH, pwHM => pwHM, pwHE => pwHE, pwClk => pwClk)
```

每个 Endpoint 连接到 Host Interface 上的 163 个目标接口引脚。信号方向是以 Endpoint 为视角。

| 信号 | 方向 | 描述 |
|---|---|---|
| pwHE[99：0] | 输入 | 接口控制（Host Interface 到 Endpoint）。 |
| pwEM[68：0] | 输出 | 接口控制（Endpoint 到 Mux）。 |

这些信号存在于每一个端点。在以下各个 Endpoint 的信号列表中，我们已将这些共同的信号省略了。

# Endpoint

在 Pionway HDL 中，Endpoint 是在 FPGA 与 PC 之间实现通信的基本单元，它分为三种类型：Wire，Trigger 和 Pipe。在实际使用过程中，必须对使用到的 Endpoint 进行实例化。以下将对各类型 Endpoint 的实例化方法进行详细介绍。

## pwWireIn

pwWireIn 提供一条名为 ep_dataout[31 :0]的 32 位输出总线。该总线的引脚以 wire 的形式连接到您的设计中，并作为从 API 至用户 HDL 的异步连接。

当上位机更新 Wire In 时，它会向 Wire In 写入新值，然后在同一时间更新这些值。因此，虽然 Wire 是异步 Endpoint，但它们会同时被更新（同步于 Host Interface 的时钟）。

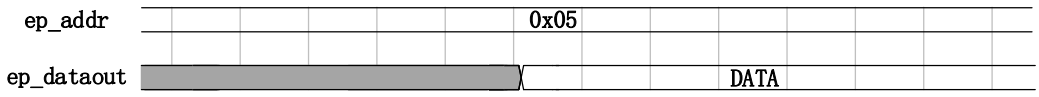
| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_dataout[31：0] | 输出 | Wire 值输出。（从 PC 发送至用户 HDL） |

Verilog 实例化：

```
pwWireIn wire05 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'h05), .ep_dataout(ep05data))
```

VHDL 实例化：

```
wire05: pwWireIn port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x"05", ep_dataout => ep05data)
```



## pwWireOut

pwWireOut 提供一条名为 ep_datain[31 :0]的 32 位输入总线。这些引脚上的信号会在每次上位机更新它的 Wire 值时被读取。事实上，所有的 Wire 会被同时捕获（同步于 Host Interface 的时钟）并被顺序读取。

| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_datain[31：0] | 输入 | Wire 值输入。（从用户 HDL 发送至 PC） |

Verilog 实例化：

```
pwWireOut wire21 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'h21), .ep_datain(ep21data))
```

VHDL 实例化：

```
wire21: pwWireOut port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x"21", ep_datain => ep21data)
```

## pwTriggerIn

pwTriggerIn 提供 ep_clk 和 ep_trigger 作为接口信号，它在 ep_trigger 上的任一位产生一个单周期触发脉冲信号，该脉冲同步于时钟信号 ep_clk。因此，该脉冲信号的周期不必一定是 Host Interface 的周期。该模块会自动处理跨时钟域的问题。

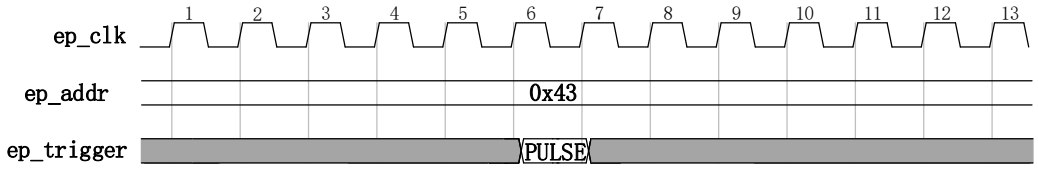
| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_clk | 输入 | Trigger In 应同步的时钟。 |
| ep_trigger[31：0] | 输出 | Trigger In 的独立输出信号。（由 PC 至用户 HDL） |

Verilog 实例化：

```
pwTriggerIn trigIn43 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'h43), ep_clk(clk2), .ep_trigger(ep43trig))
```

VHDL 实例化：

```
trigIn43: pwTriggerIn port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x"43", ep_clk=>clk2, ep_trigger => ep43trig)
```




## pwTriggerOut

用户 HDL 通过此模块向 PC 发送触发信号。Ep_trigger[31：0]包含 32 个独立触发信号，这些信号由 ep_clk 控制采集。如果某个 Trigger Out（ep_trigger[x]）在 ep_clk 的上升沿被检测为高电平，那么该 Trigger Out 将被置位。当下次 PC 检查 Trigger Out 的值时，所有 Trigger Out 的值将被清除。

| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_clk | 输入 | Trigger Out 应同步的时钟。 |
| ep_trigger[31：0] | 输入 | Trigger Out 的独立输入信号。（由用户 HDL 至 PC） |

Verilog 实例化：

```
pwTriggerOut trigOut62 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'h62), ep_clk(clk2), .ep_trigger(ep62trig))
```

VHDL 实例化：

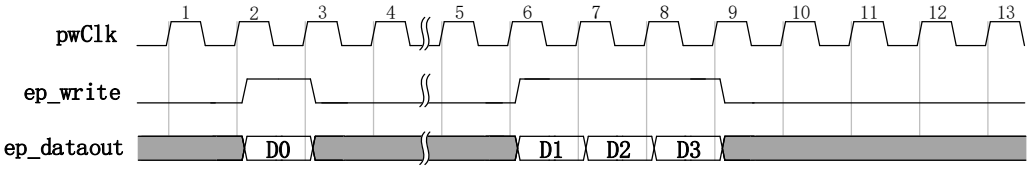
```
trigOut62: pwTriggerOut port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x"62", ep_clk=>clk2, ep_trigger => ep62trig)
```

## pwPipeIn

pwPipeIn 提供了一种由 PC 向用户 HDL 同步传输多字节数据的方法。PC 是主设备，而用户 HDL 则负责接收数据，因为这些数据是通过 Pipe 传输（高达 100MHz）。ep_write 信号是高电平有效信号，当用户 HDL 即将通过 ep_dataout[31：0]接收数据时，该信号被拉高。ep_write 信号可能会连续几个周期高电平，在这样的情况下，ep_dataout[31：0]会一直随时钟输出数据。

pwPipeIn 接口实现起来非常简单，但是它要求用户 HDL 能够对传入的 Pipe 数据立即作出响应。如果用户 HDL 能够跟上该吞吐量，但需要以块形式处理数据，那么使用 FIFO（通过 Xilinx CORE Generator 工具生成）配合 pwPipeIn 是一种很好的解决方案。或者，您也可使用 pwBlockPipeIn 替代。

以下时序图说明了 pwPipeIn 如何将数据提供给用户 HDL。当 ep_write 处于高电平状态，并且 pwClk 为上升沿时，ep_dataout 的数据为有效数据。请注意，在这个例子中，一共传输了 4 个字。需要注意的是，在传输过程中，ep_write 有可能跳转为低电平，即无效状态。这主要会发生在长字节传输（大于 256 个字）。




| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_dataout[31：0] | 输出 | Pipe 数据输出。（由 PC 发送至用户 HDL） |
| ep_write | 输出 | 写信号，高电平有效。当该信号为高电平时，用户 HDL 需要捕获数据。 |

Verilog 实例化：

```
pwPipeIn pipeIn81 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'h81), ep_dataout(ep81pipe), .ep_write(ep81write))
```

VHDL 实例化：

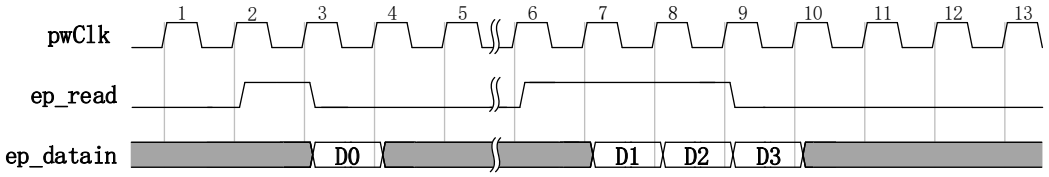
```
pipeIn81: pwPipeIn port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x" 81", ep_dataout => ep81pipe, ep_write => ep81write)
```

## pwPipeOut

pwPipeOut 提供了一种由用户 HDL 向 PC 传输同步多字节数据的方法。因为 PC 是主设备，所以所有的读操作（用户 HDL 端）都是由 PC 发起。因此，每当 ep_read 被拉高时，数据必须准备好。

pwPipeOut 接口实现起来非常简单，但是它同样要求用户 HDL 能够对上位机的读操作请求有一定的响应速度。如果用户 HDL 能够跟上该吞吐量，但需要以块形式处理数据，那么使用 FIFO（通过 Xilinx CORE Generator 工具生成）配合 pwPipeOut 是一种很好的解决方案。或者，您也可使用 pwBlockPipeOut 替代。

以下时序图说明了用户 HDL 需要如何使用 ep_datain 有效数据响应 ep_read。当 ep_read 处于高电平状态，并且 pwClk 为上升沿时，用户 HDL 必须在下一个时钟的上升沿使用 ep_datain 有效数据作出响应，但受制于建立和保持时间（FPGA CLB 时序文档里的 $T_{AS}$ 和 $T_{AH}$）。当然，这些时间也受制于您的 HDL 实现中特定的布线和逻辑。请注意，在这个例子中，一共传输了 4 个字。需要注意的是，在传输过程中 ep_read 有可能跳转为低电平，即无效状态。这主要会发生在长字节传输（大于 256 个字）。




| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_datain[31：0] | 输入 | Pipe 数据输入。（由用户 HDL 发送至 PC） |
| ep_read | 输出 | 读信号，高电平有效。当该信号为高电平时，用户 HDL 必须在下个时钟周期准备好并提供数据。 |

Verilog 实例化：

```
pwPipeOut pipeOutA2 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'h A2), ep_datain(epA2pipe), .ep_read(epA2read))
```

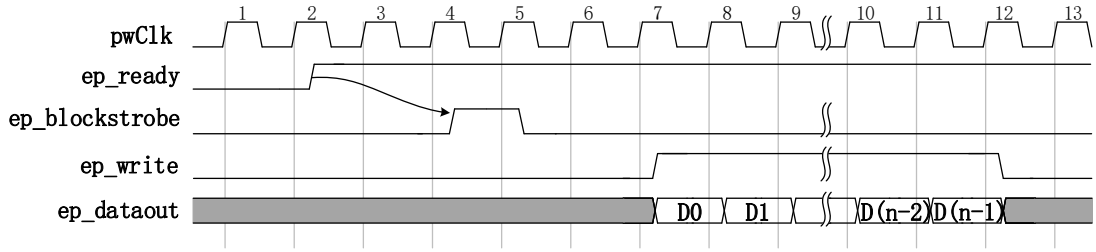
VHDL 实例化：

```
pipeOutA2: pwPipeOut port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x" A2", ep_datain => epA2pipe, ep_read => epA2read)
```

## pwBlockPipeIn

pwBlockPipeIn 类似于 pwPipeIn，只是增加了两个信号，ep_blockstrobe 和 ep_ready，用来处理数据块传输。PC 仍然是主设备，但 FPGA 控制 ep_ready 信号。当 ep_ready 有效时，PC 就可以向该模块传输一个完整的块数据；当 ep_ready 无效时，上位机将不能向该模块传输数据。

例如，ep_ready 可以连接到一个 FIFO 上的容量标志位上。当 FIFO 有可用完整的块空间时，使 ep_ready 有效来表示该 FIFO 可以接收一个完整的块数据。




| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_dataout[31：0] | 输出 | Pipe 数据输出。（由 PC 发送至用户 HDL） |
| ep_write | 输出 | 写信号，高电平有效。当该信号为高电平时，用户 HDL 需要捕获数据。 |
| ep_blockstrobe | 输出 | 块选通信号，高电平有效。该信号会在一个块数据被写入之前持续高电平一个周期。 |
| ep_ready | 输入 | 准备信号，高电平有效。用户 HDL 需要在准备好接收一个完整的块数据时将该信号置为高电平。 |

Verilog 实例化：

```
pwBlockPipeIn pipeInC7 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'HC7), ep_dataout(epC7pipe), .ep_write(epC7write),
    ep_blockstrobe(epC7strobe), ep_ready(epC7ready))
```

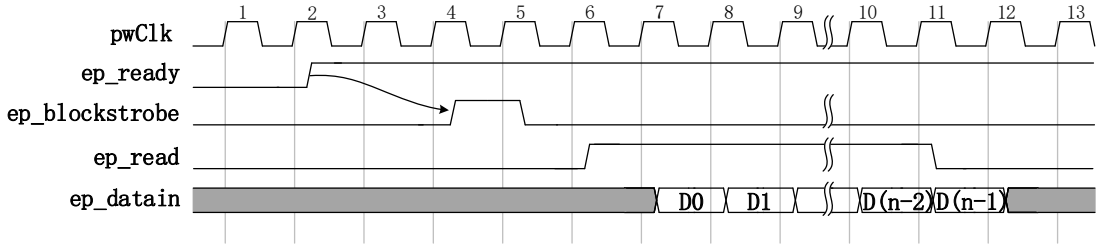
VHDL 实例化：

```
pipeInC7: pwBlockPipeIn port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x" C7", ep_dataout => epC7pipe, ep_write => epC7write,
    ep_blockstrobe => epC7strobe, ep_ready => epC7ready)
```

## pwBlockPipeOut

pwBlockPipeOut 类似于 pwPipeOut，只是增加了两个信号，ep_blockstrobe 和 ep_ready，用来处理数据块传输。PC 仍然是主设备，但 FPGA 控制 ep_ready 信号。当 ep_ready 有效时，PC 就可以从该模块读取一个完整的块数据；当 ep_ready 无效时，上位机将不能从该模块读取数据。

例如，ep_ready 可以连接到一个 FIFO 上的容量标志位上。当 FIFO 有可用完整的块数据时，使 ep_ready 有效来表示该 FIFO 有一个完整的块数据可以读取。




| 信号 | 方向 | 描述 |
|---|---|---|
| ep_addr[7：0] | 输入 | Endpoint 地址。 |
| ep_datain[31：0] | 输入 | Pipe 数据输入。（由用户 HDL 发送至 PC） |
| ep_read | 输出 | 读信号，高电平有效。当该信号为高电平时，用户 HDL 需要在下个时钟之前提供要发送至 PC 的数据。 |
| ep_blockstrobe | 输出 | 块选通信号，高电平有效。该信号会在一个块数据被读取之前持续高电平一个周期。 |
| ep_ready | 输入 | 准备信号，高电平有效。用户 HDL 需要在准备好发送一个完整的块数据时将该信号置为高电平。 |

Verilog 实例化：

```
pwBlockPipeOut pipeOutE5 (.pwHE(pwHE), .pwEM(pwEM),
    .ep_addr(8'hE5), ep_datain(epE5pipe), .ep_read(epE5read),
    ep_blockstrobe(epE5strobe), ep_ready(epE5ready))
```

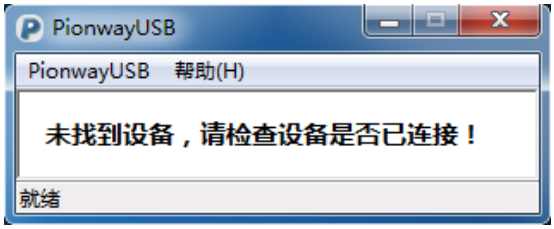
VHDL 实例化：

```
pipeOutE5: pwBlockPipeOut port map (pwHE => pwHE, pwEM => pwEM,
    ep_addr => x" E5", ep_datain => epE5pipe, ep_write => epE5write,
    ep_blockstrobe => epE5strobe, ep_ready => epE5ready)
```

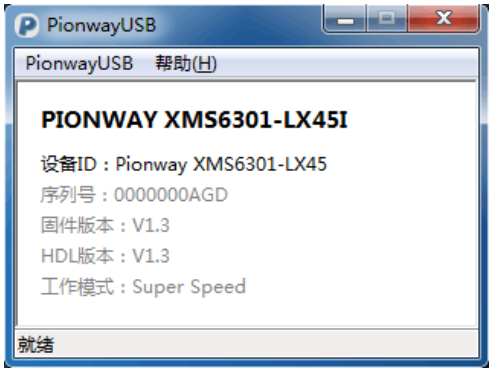
# PionwayUSB 应用程序简介

PionwayUSB 的主要功能有显示设备信息、修改设备 ID、更新设备固件、配置 FPGA 和烧写 FPGA Flash 等功能。

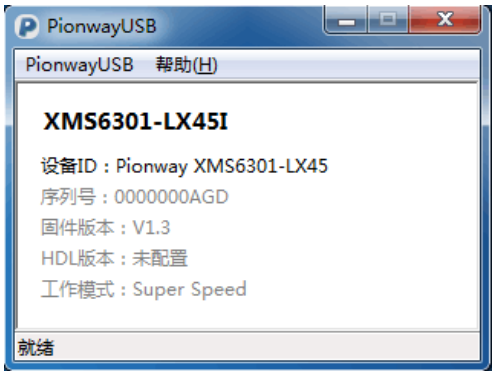
当计算机没有连接 Pionway 设备时，程序将显示“未找到设备，请检查设备是否已连接”，如下图所示。




当计算机已连接 Pionway 设备时，程序将显示设备的相关信息，如下图所示。显示的信息分别为产品名称、设备 ID、产品序列号、设备固件版本、设备 HDL 版本和 USB 工作模式。

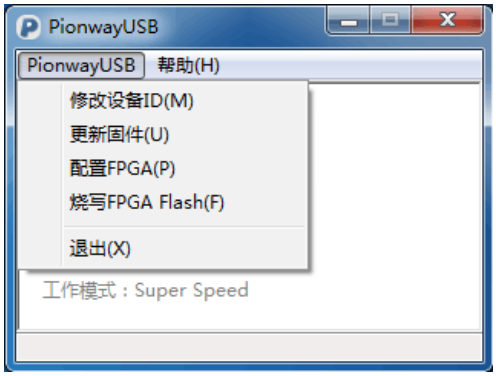



当 Pionway HDL 未配置时，HDL 版本将显示“未配置”，如下图所示。



当有多个 Pionway 设备连接计算机时，程序将自动获取设备列表中的第一个设备并显示其相关信息。因此，为了避免混乱，我们建议您在使用时仅连接一个设备。

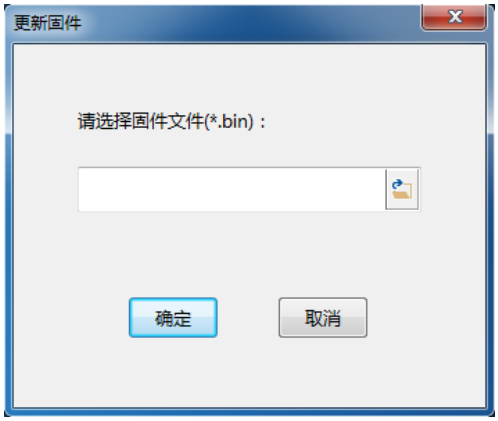
当有设备连接时，您就可以进行修改设备 ID、更新设备固件、配置 FPGA 和烧写 FPGA SPI Flash 操作了，如下图所示。点击菜单中的任意一项将会弹出相应的对话框。




## 修改设备 ID


在设备 ID 输入框中输入不超过 32 位的字符串，点击确定，如果设备 ID 修改成功，状态栏将提示“设备 ID 修改成功“，否则将会弹出错误信息提示框。

## 更新设备固件



选择固件文件（.bin 格式）或输入完整路径，点击确定，如果更新成功，状态栏将提示“固件更新成功，重启设备即可生效”，否则将会弹出错误信息提示框。

## 配置 PFGA


选择配置文件（.bit 格式）或输入完整路径，点击确定，如果配置成功，状态栏将提示“FPGA 配置成功”，否则将会弹出错误信息提示框。该功能只有在未开启 FPGA SPI Flash 的烧写模式时才能使用。

## 烧写 FPGA Flash

该功能用来开启 FPGA SPI Flash 的烧写模式。如果开启成功，将会弹出一个信息提示框，如下图所示，否则，将会弹出错误信息提示框。当烧写模式开启之后，配置 FPGA 功能暂时不可用，烧写完成后，您需要重启设备才能继续使用该功能。