(机器翻译成中文)

Xillybus host application programming guide for Windows

Xillybus Ltd.
www.xillybus.com

Version 3.0

本文档已由计算机自动翻译，可能会导致语言不清晰。 与原始文件相比，该文件也可能略微过时。

如果可能，请参阅英文文档。

This document has been automatically translated from English by a computer, which may result in unclear language. This document may also be slightly outdated in relation to the original.

If possible, please refer to the document in English.

1 介绍 **5**

2 同步 streams 与异步 streams **6**

2.1 概述 6

2.2 异步 streams 的动机 6

2.3 Streams 从 FPGA 到 host 7

2.4 Streams 从 host 到 FPGA 7

2.5 不确定性与 latency 9

3 I/O 编程实践 **10**

3.1 概述 10

3.2 读取数据指南 11

3.3 数据写入指南 13

3.4 在异步 downstreams 上执行 flush 15

3.5 使用 Microsoft 的原生 API 16

3.6 监控driver的buffers中的数据量 18

3.7 XillyUSB：需要监控物理data link的质量 18

4 高速连续 I/O **20**

4.1 基础知识 20

4.2 大 driver 的 buffers 21

4.3 user space 中的 RAM buffers 22

4.4 为什么不只是 Windows pipes? 23

4.5 fifo.c 演示应用程序概述 23

4.6 fifo.c改装笔记 24

4.7 RAM FIFO 功能 24

4.7.1 fifo_init() 25

4.7.2 fifo_destroy() 25

4.7.3 fifo_request_drain() 25

4.7.4 fifo_drained() 26

4.7.5 fifo_request_write() 26

4.7.6 fifo_wrote() 27
4.7.7 fifo_done() 27
4.7.8 FIFO_BACKOFF define variable 27

5 循环 frame buffers 29
5.1 介绍 29
5.2 改编 FIFO 示例代码 29
5.3 丢弃和重复 frames 30

6 具体的编程技术 32
6.1 Seekable streams 32
6.2 双向同步 streams 34
6.3 分组通信 34
6.4 模拟 hardware interrupts 35
6.5 Coprocessing/ Hardware acceleration 36

7 Hibernation 38

A 内部结构：streams 是如何实现的 40
A.1 介绍 40
A.2 “Classic” DMA 与 Xillybus 40
A.3 FPGA 至 host (upstream) 41
A.3.1 概述 41
A.3.2 #1 阶段：Application logic 到中间 FIFO 41
A.3.3 #2 阶段：中间 FIFO 到 DMA buffer 42
A.3.4 Stage #3：DMA buffer 到用户软件应用程序 42
A.3.5 buffers部分满载交接条件 43
A.3.6 例子 44
A.3.7 实际结论 45
A.4 Host 至 FPGA (downstream) 46
A.4.1 概述 46

A.4.2 Stage #1：DMA buffer 的用户软件应用程序 46
A.4.3 #2 阶段：DMA buffer 到中间 FIFO 47
A.4.4 #3 阶段：中间 FIFO 到 application logic 48
A.4.5 一个例子 48
A.4.6 实际结论 48

1 介绍

Xillybus 旨在为 Windows host 提供一个简单而知名的界面，具有自然和预期的行为。 host driver 生成 system objects，其行为类似于 named pipes，并声明自己是这样的。它们像任何文件一样被打开、读取和写入，但在进程之间的行为很像 pipes 或 TCP/IP streams。对于运行在host上的程序来说，区别在于stream的另一端不是另一个进程（通过网络或在同一台计算机上），而是FPGA中的一个FIFO。就像 TCP/IP stream 一样，Xillybus stream 设计用于高速数据传输以及偶尔到达或发送的单个字节。

由于与 Xillybus 的接口都是通过像任何文件一样访问的对象，因此通常可以使用任何实用的编程语言，而不需要特殊的模块、扩展或任何其他适应。如果可以使用所选语言打开文件，则可以使用该文件通过 Xillybus 访问 FPGA。

一个 driver binary 支持任何 Xillybus IP core 配置：driver 会在初始化设备时自动检测 streams 及其属性，并相应地创建 device files。这些 device files 作为 `\\.\xillybus_something`（或 `\\.\xillyusb_00_something` 与 XillyUSB ）访问。

在操作过程中，FPGA 和 host 之间的握手协议会产生连续 data stream 的错觉。在幕后，driver 的 buffers 被填充和处理。与用于 TCP/IP streaming 的技术类似的技术用于确保有效利用 buffers，同时保持对小块数据的响应能力。

2 同步 streams 与异步 streams

2.1 概述

每个 Xillybus stream 都有一个标志，用于确定其行为是同步还是异步。该标志的值在 FPGA 的 logic 中是固定的。

当 stream 被标记为异步时，只要相应的 device file 处于打开状态，就可以在 FPGA 和 host 的 kernel driver 之间进行数据通信，而无需 user space software 的参与。

异步 streams 有更好的性能，特别是在数据流是连续的时候。同步 streams 更易于处理，并且是需要在 user space application 的操作与 FPGA 中发生的操作之间紧密同步时的首选。

在 IP Core Factory 中生成的自定义 IP cores 中，使每个 stream 同步或异步之间的选择自动基于有关 stream 预期用途的信息，如启用 “autoset internals” 时工具用户声明的那样。如果 autoset 选项被关闭，用户会明确地做出这个选择。

无论哪种方式，包含在从 IP Core Factory 下载的包中的 “readme” 文件指定每个 stream 的同步或异步标志（以及其他属性）。

在所有 demo bundles 中，与 xillybus_read_* 和 xillybus_write_* 相关的 streams 都是异步的。 xillybus_mem_8 是 seekable，因此是同步的。当使用 XillyUSB 时，同样适用于各个 xillyusb_* 文件。

2.2 异步 streams 的动机

Linux 和 Microsoft Windows 等多任务操作系统基于 CPU 时间共享：进程获取 CPU 的时间片，通过一些调度算法决定在任何给定时刻哪个进程获取 CPU。

尽管可以为进程设置优先级，但不能保证进程将连续运行，或者 preemption 的周期具有有限的持续时间，即使在多处理器计算机上也不行。操作系统的基本假设是任何

进程都可以接受任何 CPU starvation 周期。面向实时的应用程序（例如声音应用程序和视频播放器）对此问题没有明确的解决方案。相反，它们依赖于操作系统的典型实际行为，并通过 I/O 缓冲来弥补 preemption periods。

异步 streams 通过允许数据在应用程序处于 preempted 或忙于其他任务时连续流动来解决此问题。接下来将讨论这对 streams 在任一方向上的确切意义。

2.3 Streams 从 FPGA 到 host

在 upstream 方向（FPGA 到 host），如果 stream 是异步的，FPGA 中的 IP core 会尽可能尝试填充 host driver 的 buffers。也就是说，当文件打开时，数据可用并且那些 buffers 中有可用空间。

另一方面，如果 stream 是同步的，只有当 host 上的 user application software 有一个从 file descriptor 读取数据的未决请求时，IP core 才会从 user application logic（通常从 FIFO）获取数据。换句话说，当 user application software 处于 read() 函数调用的中间时。

在高带宽应用中应避免使用同步 streams，主要有以下两个原因：

- 当应用程序是 preempted 或执行其他操作时，数据流会中断，因此物理通道在某些时间段内保持未使用状态。在大多数情况下，这会导致带宽性能显着降低。
- 在这些时间间隔内，FPGA 中的 FIFO 上可能会出现 overflow。例如，如果它的填充率为 100 MB/sec，则带有 2 kByte 的典型 FPGA FIFO 在 0.02 ms 左右从空变为满。实际上，这意味着 user space program 的任何 preemption 都可能导致 FPGA 中的 FIFO 的 overflow。

尽管存在这些缺点，但当 FPGA 收集数据的时间很重要时，同步 streams 还是很有用的。特别是，类似内存的接口需要一个同步接口。

Xillybus IP core 在 FPGA 上从 application logic 接收到的数据可以立即被 host 的 user space application 读取，无论 stream 是同步的还是异步的。

2.4 Streams 从 host 到 FPGA

在 downstream 方向（host 到 FPGA），stream 是异步的意味着 host application 的 write() 函数调用将在大多数时间立即返回。更准确地说，如果数据可以完全存储在 driver 的 buffers 中，则对写入 device file 的函数的调用将立即返回。然后数据以 user application logic 在 FPGA 上请求的速率传输到 FPGA，host 的 application software 不参与。

XillyUSB 和其他 Xillybus IP cores 之间存在细微差别，关于将数据发送到 FPGA 的速度，代表异步 streams 到 FPGA。

对于基于 PCIe 或 AXI 的 IP cores，只有在以下情况之一发生时才会将数据发送到 FPGA：

- 当前的 DMA buffer 已满（每个 buffers 有几个 stream ）。
- application software 在 device file 上明确请求 flush（参见 3.4 段）
- file descriptor 正在关闭。
- 如果在特定时间（通常是 10 ms ）内没有向 stream 写入任何内容，则计时器到期，强制执行自动 flush。

使用 XillyUSB stream，数据几乎可以立即发送。更准确地说，driver 尝试将固定大小（通常为 64 kB）的 USB transfers 排队，但如果有数据要传输，则将排队较小的传输，并且相关 stream 没有其他传输排队。因此，对于每个 stream，永远不会有超过一个小于固定大小的排队传输，但只要有数据要传输，总是至少有一个传输在进行中。这导致了 USB 传输的有效使用以及对短数据段的快速响应。

总而言之，所有 IP cores（XillyUSB 和其他 Xillybus IP cores）上的异步 streams 的行为大致相同，XillyUSB 对短数据段的响应时间更快（没有 10 ms 延迟）。

另一方面，如果 stream 是同步的，对写入 device file 的低级函数的调用将不会返回，直到所有数据都到达 FPGA 中的 user application 的 logic。在典型的应用中，这意味着当对write()的函数调用返回时，表明数据已经到达FPGA中与IP core相连的FIFO。

重要的:

更高级别的 *I/O* 功能，例如 *fwrite()*，涉及由 *library functions* 创建的 *buffer layer*。因此，*fwrite()* 和类似的函数可能会在数据到达 *FPGA* 之前返回，即使对于同步 *streams* 也是如此。

在高带宽应用中应避免使用同步 streams，主要有以下两个原因：

- 当应用程序是 preempted 或执行其他操作时，数据流会中断，因此物理通道在某些时间段内保持未使用状态。在大多数情况下，这会严重影响带宽性能。
- 在这些时间间隔内，FPGA 中的 FIFO 上可能会出现 underflow。例如，如果它的排水率为 100 MB/sec，典型的 FPGA FIFO 和 2 kByte 在 0.02 ms 左右从满到空。实际上，这意味着 user space program 的任何 preemption 都可能导致 FPGA 中的 FIFO 的 underflow。

尽管有这些缺点，但同步 streams 在应用程序知道数据已到达 FPGA 很重要时很有用。当 stream 用于传输必须在其他操作（例如硬件配置）发生之前执行的命令时就是这种情况。

2.5 不确定性与 latency

一个常见的错误是为了数据之间的同步而要求异步 streams 上的 latency 较低。例如，如果应用程序是调制解调器，通常自然需要在接收和发送的样本之间进行同步。

这通常会导致对 design 的误解，基于同步中的不确定性必然小于总 latency 的概念。为了保持较低的不确定性，latency 以及 buffers 都尽可能地小，从而导致整个系统对 real-time 的要求很困难。

使用 Xillybus，同步很容易做到完美（在单个样本的级别），如 6.2 段中所述。因此，latency 的限制源于对到达数据快速响应的需要（如果有这种需要）。

例如，对于调制解调器，最大 latency 会影响应用程序的数据源对发送给它的数据的响应速度。在相机应用中，host 可以对相机进行编程以调整 shutter speed 以补偿不断变化的光照条件。较大的 latency 到达的数据会减慢 control loop 的速度。

这些是需要考虑的真正考虑因素，而且它们通常比那些因对 latency 混合不确定性的误解而产生的考虑要严格得多。

3 I/O 编程实践

3.1 概述

Xillybus 适用于任何能够访问文件的编程语言，任何用于访问文件的 API 都适用。

在本指南中，重点介绍了基于 _open()、_read()、_write() 和 _close() 等功能的低级经典 C API。这些带下划线前缀的函数的行为与不带下划线的函数完全相同（即 _read() 和 read() 是相同的函数）。但是，当使用非下划线功能时，Microsoft compiler 可能会发出警告。

Microsoft API（即 CreateFile()、ReadFile() 等）也可以使用（参见 3.5 段落），但由于以下原因，本指南中大多忽略了它：

- Windows API 提供的额外功能不是必需的（即使此 API 提供更准确的错误报告）。
- 经典的 C API 是众所周知的，更简单，更容易被程序员采用。
- 经典的 C API 不太可能跨 Windows 的不同版本进行修改。
- 经典的 C API 使代码可移植。

低层集比其他众所周知的集（例如 fopen()、fwrite()、fprintf() 等）被选择，因为低层 API 的功能没有额外的 buffers 层。这些 buffers 可以对性能产生积极影响，但使用它们无法控制 I/O 的实际操作。

当数据不断传输并且预计软件操作和 I/O 与硬件之间没有直接关系时，这一点就不那么重要了。

额外的 buffer 层也会引起混淆，看起来好像没有软件错误。例如，对 fwrite() 的函数调用只能将数据存储在 RAM buffer 中，而不执行任何 I/O 操作，直到文件关闭。没有

意识到这一点的开发人员可能会误以为 fwrite() 失败是因为 FPGA 端没有发生任何事情，而实际上数据正在 buffer 中等待。

本节介绍推荐的 UNIX 编程实践，使用低级 C run-time library 函数。为了完整起见，这里给出了详细说明，因为对于这些实践中的任何一个都没有任何特定于 Xillybus 的内容。

代码片段取自 Getting started with Xillybus on a Windows host中描述的演示应用程序。这些示例中的 device file 名称是用于 PCIe 的 Xillybus IP core 名称。

以下示例针对 Xillybus 的 PCIe 变体给出。它们也适用于 XillyUSB，但文件名前缀是 xillyusb_00_* 而不是 xillybus_*。如果连接了多个 XillyUSB 设备，则 “00” 部分将替换为发现设备时空闲的最低索引，即 01、02 等。一旦为设备分配了索引，只要它保持不变，它就不会改变连接的。

这些示例的典型 header files 是

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

3.2 读取数据指南

假设变量声明如下：

```
int fd, rc;
unsigned char *buf;
```

device file用低级open打开（file descriptor是integer格式）：

```
fd = _open("\\\\.\\xillybus_ourdevice", O_RDONLY | _O_BINARY);

if (fd < 0) {
  perror("Failed to open devfile");
  exit(1);
}
```

文件名中的 `\\\\.\\` 前缀在 backslashes 的 unescaping 之后变为 `\\.\`。

注意 `_O_BINARY` 标志，它告诉 Windows 将 stream 视为非文本数据。在没有此标志的情况下，Windows 将转换 newline 字符并将 CTRL-Z (0x1a) 视为 EOF (end of

file)。

如果 device file 已打开以供另一个进程读取（可根据要求提供非独占文件打开），则会发出 “Device or resource busy” (errno = EBUSY) 错误。如果出现 “No such device” (errno = ENODEV)，则很可能是尝试打开只写 stream。

成功打开文件并且 `buf` 指向内存中分配的 buffer 后，读取数据：

```
while (1) {
  rc = _read(fd, buf, numbytes);
```

`numbytes` 是要读取的最大字节数。

返回值 `rc` 包含实际读取的字节数（如果函数调用异常完成，则为负值）。

请注意，如果 `numbytes` 中请求的数据量可用，_read() 总是立即返回。否则，如果有任何可用数据，它将在大约 10 ms 之后返回。如果根本没有数据可用，_read() 会休眠，直到它可以返回数据。

driver 在 IP core 已从 FPGA 中的 application logic 接收数据的意义上检查数据的可用性。 DMA buffers 的机制对函数 _read() 的调用者是透明的，并且不会因为 DMA buffer 未满而延迟向 _read() 函数调用传递数据，如附录 A.3.5 部分所述。

> **重要的:**
>
> 不能保证从文件中读取所有请求的字节，即使 *_read()* 成功返回也是如此。如果完成的数据量不令人满意，则调用者有责任对 *_read()* 进行另一个函数调用。

对 _read() 的函数调用后应检查其返回值，如下所示（“continue” 和 “break” 语句假定 while 循环上下文）：

```
  if ((rc < 0) && (errno == EINTR))
    continue;

  if (rc < 0) {
    perror("read() failed");
    break;
  }

  if (rc == 0) {
    fprintf(stderr, "Reached read EOF.\n");
    break;
  }

  // do something with "rc" bytes of data
}
```

第一个 if 语句检查 _read() 是否因为 signal 而过早返回。这是进程从操作系统接收 signal 的结果。

这实际上不是错误，而是强制 driver 立即将控制权返回给应用程序的条件。使用 EINTR 错误号只是告诉函数调用者没有读取数据的一种方式。程序以 “continue” 语句响应，从而重新尝试使用相同的参数调用函数 _read()。

如果 signal 到达时 buffer 中有数据，driver 会返回 rc 中已经读取的字节数。应用程序不会知道 signal 已经到达，并且根据 UNIX 编程约定，它没有理由关心：如果 signal 需要操作（例如 SIGINT 由键盘上的 CTRL-C 产生），则该操作的责任在操作系统，或注册的 signal handler。

请注意，某些 signals 不应该对执行流程产生任何影响，因此如果没有如上所示检测到 signals，则程序可能会无缘无故地突然报告错误。

处理 EINTR 场景对于允许进程停止（与 CTRL-Z 一样）并正确恢复也是必要的。

请注意，signals 属于 UNIX 的世界，因此尽管上面对它们进行了所有说明，但尚不清楚它们是否会出现在 Windows 计算机上。无论哪种方式，相关的 if 语句都是无用的，但在最坏的情况下是无害的。

如果在报告用户可读的错误消息后发生真正的错误，则第二个 if 语句终止循环。

第三个 if 语句检测是否已到达 end of file，这由返回值 0 指示。当从 Xillybus device file 读取数据时，发生这种情况的唯一原因是 application logic 抬高了 stream 的 _eof 引脚（这是 FPGA 上 IP core 接口的一部分）。

3.3 数据写入指南

假设变量声明如下：

```
int fd, rc;
unsigned char *buf;
```

device file用低级_open打开（file descriptor是integer格式）：

```
fd = _open("\\\\.\\xillybus_ourdevice", O_WRONLY | _O_BINARY);

if (fd < 0) {
  perror("Failed to open devfile");
  exit(1);
}
```

文件名中的 \\\\.\\ 前缀在 backslashes 的 unescaping 之后变为 \\.\。

注意 _O_BINARY 标志，它告诉 Windows 将 stream 视为非文本数据。在没有此标志的情况下，Windows 将转换 newline 字符并将 CTRL-Z (0x1a) 视为 EOF (end of

file)。

如果 device file 已打开以供另一个进程写入（可根据要求提供非独占文件打开），则会发出 “Device or resource busy” (errno = EBUSY) 错误。如果出现 “No such device” (errno = ENODEV)，则很可能是尝试打开只读 stream。

成功打开文件并且 `buf` 指向内存中分配的 buffer 后，写入数据：

```
while (1) {
  rc = _write(fd, buf, numbytes);
```

`numbytes` 是要写入的最大字节数。

返回值 `rc` 包含实际写入的字节数（如果函数调用异常完成，则为负值）。

> **重要的:**
>
> 无法保证所有请求的字节都已写入文件，即使 *_write()* 成功返回也是如此。如果完成的数据量不令人满意，则调用者有责任对 *_write()* 进行另一个函数调用。

对 _write() 的函数调用后应检查其返回值，如下所示（“continue” 和 “break” 语句假定 while 循环上下文）：

```
  if ((rc < 0) && (errno == EINTR))
    continue;

  if (rc < 0) {
    perror("write() failed");
    break;
  }

  if (rc == 0) {
    fprintf(stderr, "Reached write EOF (?!)\n");
    break;
  }

  // do something with "rc" bytes of data
}
```

第一个 if 语句检查 _write() 是否因为 signal 而过早返回。这是进程从操作系统接收 signal 的结果。

这实际上不是错误，而是强制 driver 立即将控制权返回给应用程序的条件。使用 EINTR 错误号只是告诉函数调用者没有写入数据的一种方式。程序以 “continue” 语句响应，从而重新尝试使用相同的参数调用函数 _write()。

如果在 signal 到达之前写入了一些数据，driver 将返回 `rc` 中已经写入的字节数。应

用程序不会知道 signal 已经到达，并且根据 UNIX 编程约定，它没有理由关心：如果 signal 需要操作（例如 SIGINT 由键盘上的 CTRL-C 产生），则该操作的责任在操作系统，或注册的 signal handler。

请注意，某些 signals 不应该对执行流程产生任何影响，因此如果没有如上所示检测到 signals，则程序可能会无缘无故地突然报告错误。

处理 EINTR 场景对于允许进程停止（与 CTRL-Z 一样）并正确恢复也是必要的。

请注意，signals 属于 UNIX 的世界，因此尽管上面对它们进行了所有说明，但尚不清楚它们是否会出现在 Windows 计算机上。无论哪种方式，相关的 if 语句都是无用的，但在最坏的情况下是无害的。

如果在报告用户可写错误消息后发生真正的错误，则第二个 if 语句终止循环。

第三个 if 语句检测是否已到达 end of file，这由返回值 0 指示。写入 Xillybus device file 时，永远不会发生这种情况。

3.4 在异步 downstreams 上执行 flush

如段落 2.4 中所述，写入 PCIe / AXI IP core 上的异步 stream 的数据不一定会立即发送到 FPGA，除非 DMA buffer 已满（有几个 DMA buffers ）。此行为通过确保使用分配的 buffer 空间来提高性能。这也提高了在 PCIe / AXI bus 上发送数据包的效率。

正如已经提到的，XillyUSB IP cores 几乎立即发送数据，即使 stream 是异步的，因为 USB 接口有一个有效的安排。因此，只有在涉及等待传输完成时，执行 flush 才对 XillyUSB IP cores 有意义。

Streams 到 FPGA 在关闭 file descriptor 时会自动经历 flush，但这是不能依赖的尽力而为机制。对 _close() 的函数调用被延迟，直到所有数据都到达 FPGA，其方式类似于 write() 函数调用在同步 streams 上被延迟的方式。显着的区别是 _close() 等待 flush 完成最多一秒钟。如果此时 flush 还没有完成，_close() 无论如何都会返回，并在 Event Log 中发出警告消息。但是请注意，在一些罕见的情况下，关闭 file descriptor 时，剩余数据的最后几个字可能会在没有任何警告的情况下丢失。

也可以通过使用长度为零的 buffer 调用函数 _write() 来显式请求异步 stream 的 flush，即

```
while (1) {
  rc = _write(fd, NULL, 0);

  if ((rc < 0) && (errno == EINTR))
    continue; // Interrupted. Try again.

  if (rc < 0) {
    perror("flushing failed");
    break;
  }

  break; // Flush successful
}
```

请注意以下事项：

- 当 count 为零时，_write() 函数调用应该做什么没有明确的系统定义，将选择权留给每个 device driver。 flushing 的这种方法特定于 Xillybus。
- 与 _close() 不同，如上所示的 _write() 会立即返回，而不管 FPGA 上何时使用数据。
- 正因为如此，这种_write()对XillyUSB毫无意义。它没有任何关系，实际上也没有任何作用：无论如何，数据实际上是立即发送的，_write() 函数调用在任何情况下都不会等待。
- 由于没有从 buffer 读取数据，因此 _write() 函数调用中的 buffer 参数可以采用任何值，包括 NULL，如上所示。
- 将更高级别的 API 与零长度的 buffer 一起使用，可能根本没有任何效果。例如，调用函数 fwrite() 写入零字节可能只是简单地返回而什么都不做，因为该函数通常所做的是将数据添加到由 C run-time library 创建的 buffer。
- fflush() 无关：它执行更高级别 buffer 的 flush，但不向低级别 driver 发送 flush 命令。
- 不需要在另一个方向（从 FPGA 到 host ）对 streams 执行 flush ，也没有办法这样做。这是因为当 host 尝试读取数据即将使进程进入睡眠状态（即 block ）时，会自动执行此类 streams 的 flush 。

3.5 使用 Microsoft 的原生 API

虽然不推荐使用，但为了完整起见，还是给出了一个使用 Microsoft 的原生 API 从

stream 读取的示例。完整代码在演示应用程序包中以 winstreamread.c 的形式提供（请参阅 Getting started with Xillybus on a Windows host）。

首先，让我们定义一个用于打印错误的辅助函数。它是 Windows 的 perror() 的对应物：它的目的是将错误代码转换为人类可读的消息。

它接受一个字符串来描述尝试的操作，以及错误代码。作为响应，该函数打印给定的字符串、错误代码和由 Windows 翻译的人类可读的错误描述。

```
void errorprint(char *what, DWORD dw) {
  LPVOID lpMsgBuf;

  FormatMessage(
                FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM |
                FORMAT_MESSAGE_IGNORE_INSERTS,
                NULL,
                dw,
                MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                (LPTSTR) &lpMsgBuf,
                0, NULL );


  fprintf(stderr, "%s: Error=%08x:\n%s\n",
          what, dw, lpMsgBuf);

  LocalFree(lpMsgBuf);
}
```

假定以下变量声明：

```
HANDLE fh;
DWORD rc;
```

使用对 CreateFile() 的函数调用打开文件，尽管它的名称不一定创建文件，但会打开它。

```
fh = CreateFile("\\\\.\\xillybus_ourdevice", // file to open
                GENERIC_READ,                // open for reading
                0,                           // do not share
                NULL,                        // no security
                OPEN_EXISTING,               // existing file only
                FILE_ATTRIBUTE_NORMAL,
                NULL);                       // no attr. template
```

CreateFile() 总是在 “binary mode” 中打开文件。事实上，它不支持称为 “text mode”

的 LF 转换。

文件名中的 `\\\\.\\` 前缀在 backslashes 的 unescaping 之后变为 `\\.\`。

检查打开文件是否成功，并在失败的情况下调用函数 errorprint()（如上定义）。

```
if (fh == INVALID_HANDLE_VALUE) {
  errorprint("Failed to open file", GetLastError());
  return 1;
}
```

成功打开文件并且 `buf` 指向内存中分配的 buffer 后，读取数据：

```
if (!ReadFile(fh,  buf, numbytes, &rc, NULL)) {
    errorprint("ReadFile", GetLastError());
    return 1;
}
```

`numbytes` 是要读取的最大字节数。

ReadFile() 写入的 `rc` 包含实际读取的字节数。

如果为零，则已到达 end of file。当从 Xillybus device file 读取时，发生这种情况的唯一原因是 application logic 已经提升了 stream 的 _eof 引脚（这是 FPGA 上 IP core 接口的一部分）。

```
if (rc == 0) {
  fprintf(stderr, "Reached EOF.\n");
  return 0;
}
```

重要的:

无法保证从文件中读取了所有请求的字节，即使 *ReadFile()* 成功返回也是如此。如果完成的数据量不令人满意，则调用者有责任再次调用函数 *ReadFile()*。

3.6 监控driver的buffers中的数据量

此主题在 Xillybus FPGA designer's guide中名为 "Monitoring the amount of buffered data" 的部分中进行了讨论。

3.7 XillyUSB：需要监控物理data link的质量

与 PCIe 不同，已观察到与 USB 3.0 一起使用的物理 data link 生成 bit errors。这种情况不常见，表明其中一个组件存在问题，很可能是 host 的 USB 端口或电缆。

USB 协议提供了多种机制来克服发生这种情况时的 bit errors，但是这些错误的随机性质使 link protocol 处于很少达到的状态。因此，这可能会揭示 host 的 USB controller 中的错误。这样的错误，在它们存在的范围内，通常是隐藏的，并导致各种奇怪的行为。

因此，如果物理 data link 遭受频繁 bit errors 的困扰，则 USB 连接存在很大的风险，即 USB 连接会卡住、自发断开，或者在极少数情况下，甚至会导致应用程序数据出错。

XillyUSB 通过专用的 device file、`\\.\xillyusb\_NN\_diagnostics` 提供了一种监控物理 data link 健康状况的方法。 showdiagnostics 实用程序（在此 web page 上进行了解释）公开了在此问题上收集的信息。

强烈建议基于 XillyUSB 的应用程序持续监视 showdiagnostics 实用程序显示的前五个计数器（与坏包、检测到的错误和 Recovery 请求有关），并确保它们不会增加。如果他们这样做了，特别是如果他们反复增加，应用软件应该建议纠正措施，可能是以下之一：

- 断开 USB 插头并重新连接到另一个端口。这可能会有所帮助，因为有些主板有不同的端口连接到不同品牌的 USB host 控制器（通常是为了支持更高版本的 USB 3.x 协议）。
- 断开并重新连接同一端口上的 USB 插头。如果 analog signal equalizer（消除物理信号路径引起的衰减和反射）最终处于次优状态，这可能会有所帮助。
- 尝试使用不同的 USB 电缆。

即使在 bit errors 存在的情况下，应用程序也很有可能继续完美运行。因此，最好在考虑到用户可能不会遇到任何明显问题的情况下提出纠正措施建议。

showdiagnotics.pl 实用程序可用作参考代码，因为它的 C 源代码发布在与 executable 相同的 zip 文件中。

请注意，这些问题都不是 XillyUSB 特有的。相反，这些问题很可能会影响任何 USB 3.0 设备，但 XillyUSB 提供了检测它们的方法。此外，值得重申的是，PCIe 链路不存在任何类似问题，这很可能是由于更好地控制了物理连接和信号路由。

4 高速连续 I/O

4.1 基础知识

有四种实践对于在 host 和 FPGA 之间实现高速连续数据流几乎是必不可少的:

- 使用异步 streams
- 确保 driver 的 buffers 足够大以补偿 user space application 的 I/O 操作之间的时间间隔。
- 一旦有可用数据，让 user space application 从 device file 读取数据，或者在 buffers 中有可用空间时立即向其写入数据。
- 当 FPGA 不断插入或排出数据时，切勿关闭和重新打开 device files。

XillyUSB 在保持数据的连续流方面提出了额外的挑战，正如 web page 中所解释的那样。

Xillybus FPGA designer's guide在名为 "Monitoring the amount of buffered data" 的部分中讨论了在任何给定时间监视 driver 的 buffers 中保存的数据量。

上面列表中的第一项，即使用异步 streams，在 2 部分中进行了讨论。第二个和第三个将在本节的其余部分讨论。

要理解第四项，回想一下异步 streams 的优点是数据在 FPGA 和 host 之间运行，无需 user space application 的干预。当文件关闭时，此流程将停止。

特别是对于从 host 到 FPGA 的 stream，关闭文件会强制对 buffers 中的所有数据进行 flush，并且仅在完成后（或一秒后）关闭文件。因此，从关闭文件到再次打开文件（数据写入 file descriptor）之间存在时间间隔，没有数据流。

至于 FPGA 中的 streams，关闭文件会导致 pipe 中从 FPGA 中的 application logic 到 host 中的 user space application 的所有数据丢失（即 FPGA 的 FIFO 和 driver 的

buffers ）。避免这种丢失的唯一方法是在关闭文件之前清空此 pipe 中的所有数据。再一次，在关闭文件和再次打开文件之间存在没有数据流动的时间间隔。

一个常见的错误是使用 EOF 功能来标记数据块（例如完整的 video frames ），并且这样做会迫使 host 在已知边界处关闭并重新打开 device file。然而，这显着增加了 overflow 在 FPGA 的 FIFO 上的风险。

重要的是要记住，操作系统可能会在任何给定时刻（ preemption ）从 user space application 中删除 CPU，因此程序中的后续函数调用之间可能会出现几个甚至几十毫秒的时间间隔。

4.2 大 driver 的 buffers

在 FPGA 和 host 之间以高速率传输数据的最大挑战之一是保持连续流。在涉及 data acquisition 和播放的应用程序中，overflow 或数据不足会导致系统无法正常工作。为避免这种情况，driver 在 host 上分配大 RAM buffers 供自己使用。这些 buffers 弥补了时间间隙，在此期间应用程序无法处理数据传输。

Xillybus 允许分配巨大的 driver 的 buffers，但是这块内存必须从操作系统的 kernel RAM 的池中分配。在某些系统（尤其是 32 位系统）上，这种内存的寻址空间被 Windows 操作系统限制为 1 GB，即使可用的总 RAM 明显更大。在 RAM 小于 1 GB 的系统中，所有内存都可能用于 driver 的 buffers。

使用增强型 host driver 时，可以在 64 位系统上分配更大的 buffers，如本页所述：

http://xillybus.com/doc/huge-dma-buffers/

除了 XillyUSB，driver 的 buffers 在 Xillybus driver 加载时分配（通常在 boot 进程的早期），并且仅在 driver 从 kernel 卸载时（通常在 system shutdown 期间）才释放。当 buffers 很大时，这通常意味着 kernel 的 RAM 池的很大一部分被 driver 的 buffers 占用。这是一个相当合理的设置，因为使用这些 buffers 的应用程序很可能是运行它的机器的主要用途。

巨大的 buffers 的一个潜在问题是它们占据了物理 RAM 的连续段。这与在userspace程序中分配的buffer相反，buffer在virtual address space中是连续的，但可以遍布物理内存，甚至根本不占用任何物理RAM。

随着操作系统的运行，可用内存池变得碎片化。这就是为什么 Xillybus driver 会尽快分配其 buffers，并且即使在不积极使用时也会保留它们。由于同样的原因，尝试卸载 driver 并在稍后阶段重新加载它可能会失败。

XillyUSB 有不同的内存分配方法，它更能容忍物理内存碎片。这也是它的driver在打开一个device file时为其buffers分配RAM，在文件关闭时释放它的原因之一。

但是，应采取预防措施以避免 kernel RAM 短缺。 IP Core Factory 的自动内存分配 (

"autoset internals") 算法旨在不消耗超过 50% 的相关内存池，即 512 MB，基于现代 PC 安装的 RAM 超过 1 GB 的假设。达到 75% 也可能是安全的，这可以通过手动设置 buffer 尺寸来完成。

buffers 的过度分配可能会导致系统不稳定。特别是，当操作系统无法从 kernel pool 分配 RAM 时，它很可能会随机终止进程。

4.3 user space 中的 RAM buffers

对于在 32 位机器上需要 buffers 大于 512 MB 的应用程序，建议在 user space RAM 中做一些缓冲。在 64 位机器上，此选项很少相关，除非所需的 buffer 大小非常大，而不是 2 的幂 (2^N)。例如，通过 Xillybus DMA buffers 无法为 stream 提供 62 GB 的 buffer，但可以通过 user space RAM 实现。

I/O 的连续性问题可以通过在 user space application 中分配一个巨大的 buffer 来解决，这似乎违反直觉。事实上，当操作系统饿死 CPU 的应用程序时，这种解决方案也无济于事。但是，如果操作系统的 scheduler 设计得相当好并且优先级设置正确，那么即使在负载较重的计算机上，user space application 也会经常获得其 CPU 片。

注意 buffer 的第一次填充很重要：当 user space application 请求内存时，现代操作系统不会分配任何物理 RAM。相反，他们只是设置 memory page tables 来反映内存分配。只有当应用程序尝试使用它时才会分配实际的物理内存。这是节省资源的绝妙方法，但会对 data acquisition 应用程序产生灾难性影响：例如，考虑当数据开始从数据源涌入时会发生什么。应用程序将数据写入刚刚分配的buffer，但是每次访问一个新的memory page，操作系统都需要获取一个新的physical memory page。如果碰巧有空闲的物理 RAM，或者如果有一种快速释放物理内存的方法（例如，已经与磁盘同步的 disk buffers），这种内存调整可能会被忽视。但是在没有物理 RAM 的直接来源的情况下，可能必须进行磁盘操作（RAM swapping 到磁盘或 flushing disk buffers），这可能会使应用程序停止太长时间。

真正的坏消息是，获取初始数据负载的能力取决于整个系统的状态。因此，通常可以运行的程序可能会突然失败，因为其他一些程序只是在同一台计算机上做了一些数据密集型的事情。

自然的解决方案是内存锁定：VirtualLock() 告诉操作系统一定的（虚拟）内存块必须保存在物理 RAM 中。这会强制立即分配物理内存，因此如果需要磁盘操作来完成函数调用，则可能需要一些时间才能返回。

操作系统不愿意锁定大块 RAM，因为这会影响其整体性能。调用函数 SetProcessWorkingSetSize() 来提高 RAM 锁定限制通常是 VirtualLock() 成功所必需的。

4.4 为什么不只是 Windows pipes?

Windows 标准 API 支持（至少）两个用于创建 pipes 的函数：CreatePipe() 和 CreateNamedPipe()。这两个函数允许调用者确定 buffer 的大小，所以从表面上看，他们可以完成这项工作。不幸的是，Windows 将请求的 buffer 大小视为建议，如 CreatePipe() 和 CreateNamedPipe() 的网页中所述。

4.5 fifo.c 演示应用程序概述

在 Linux 和 Windows 可以下载的演示应用程序中，有一个叫做 “fifo.c”。这是一个如何使用两个 threads 实现 RAM FIFO 的示例，已在 32 位和 64 位平台上进行了测试。

有关演示应用程序的更多信息，请参阅 Getting started with Xillybus on a Windows host。

请注意，与文档中的其他任何地方不同，本节中的 “FIFO” 一词是指 host 上的 RAM buffer，而不是 FPGA 中的 FIFO。

这个程序的目的是测试快速的 streams，其中一个 RAM FIFO 是维护一个巨大的 RAM buffer 所必需的。换句话说，如果您需要一个比 16 GB 更小的 buffer，那么您很可能不需要这个程序。

它还可以用作自定义应用程序中修改和采用的基础。它的设计没有 mutexes，因此没有 thread 会因为另一个 thread 持有 lock 而进入睡眠状态。当然，当 FIFO 的状态需要时（例如，从空的 FIFO 请求读取），睡眠 (blocking) 确实会发生。

这种没有 mutexes 的实现需要仔细使用 API 函数，因为它们不是 reentrant。但是，使用一台 thread 读取和一台 thread 写入没有问题。

要将 data acquisition 从 device file 运行到具有 128 MB 的 buffer 的磁盘文件中，请键入以下内容：

```
> fifo 134217728 \\.\xillybus_async > dumpfile
```

如果没有给出文件名作为第二个参数，则程序从 standard input 读取。

该程序创建了三个 threads：

- read_thread() 从 standard input（或命令行中给出的文件）读取数据并将数据写入 FIFO
- write_thread() 从 FIFO 读取并写入 standard output
- status_thread() 反复向 standard error 打印一条状态行

第三个 thread 没有功能意义，可以淘汰。也可以在主 thread 中运行其中一种读/写功能。例如，在 data acquisition 应用程序中，可能很自然地只启动 read_thread() 将数据从 file descriptor 移动到 FIFO，但在主应用程序的 thread 中消耗来自 FIFO 的数据。

4.6 fifo.c改装笔记

如果要修改程序，请记住以下几点：

- fifo_* 功能不是 reentrant。当每个 thread 使用一组其他 thread 不使用的功能（这是自然使用）时，使用它们是安全的。
- 函数 fifo_init() 可能需要一些时间才能返回，并且应该在打开异步 Xillybus device file 之前调用。
- 在应用程序中读取的 thread 和写入的 thread 总是尝试在其 I/O 请求中允许的最大字节数。这在某些情况下可能会出现问题，例如当 I/O 源是 /dev/zero 而目标是 /dev/null 时。两者都将在一次尝试中完成整个请求，因此 FIFO 将从完全空到完全充满，然后再一次。在这种情况下，限制调用 I/O 函数时请求的字节数更为明智。

4.7 RAM FIFO 功能

除了修改 fifo.c 示例外，还可以采用源代码中的一组函数。

fifo.c 文件中有一段 FIFO API 功能明显不同。这些函数可以在自定义应用程序中使用，遵循示例并根据下面的函数描述。

重要的:

尽管 *fifo_** 函数旨在用于 *multi-threaded* 环境，但这些函数**不是可重入**的。这意味着一个 *thread* 应该调用与从 *FIFO* 读取相关的函数，而另一个 *thread* 应该进行写入，因此每个 *thread* 调用其单独的一组函数。

除了初始化器、销毁器和 thread join 帮助器外，API 有四个读写函数，每个方向两个。这些函数都没有真正访问 FIFO 中的数据。它们仅维护 FIFO 的状态并提供执行读取、写入、内存复制等所需的信息。

预期的执行过程如下：从 FIFO 读取的 thread 调用函数 fifo_request_drain()，该函数返回有关可以读取多少字节的信息，以及可以从中读取数据的 pointer。如果 FIFO 为空，thread 将休眠直到数据到达。

然后，用户应用程序对指向的数据进行所需的任何使用。在消耗完部分或全部数据（写入文件、复制数据、运行某种算法等）后，它调用函数 fifo_drained() 通知 FIFO API 实际消耗了多少字节。 API 释放 FIFO 中相关的内存部分。如果写入的 thread 由于 FIFO 已满而处于休眠状态，则它会被唤醒。

请注意，读取的 thread 不要求特定数量的字节。相反，fifo_request_drain() 告诉应用程序可以消耗多少字节，应用程序会报告它在 fifo_drained() 中选择消耗的字节数。

至于相反的方向，采取类似的做法：写的thread调用函数fifo_request_write()。此函数返回可写入 FIFO 的字节数，如果 FIFO 已满则休眠。用户应用程序将所需的字节数（但不超过 fifo_request_write() 允许的字节数）写入它从 fifo_request_write() 获得的地址，然后将它所做的事情报告给 fifo_wrote()。

我们现在将详细介绍这些功能中的每一个。

4.7.1 fifo_init()

fifo_init(struct xillyfifo *fifo, unsigned int size)–这个函数初始化FIFO的信息结构并为FIFO分配内存。它还尝试将 FIFO 的 virtual memory 锁定到物理 RAM，使其准备好立即快速写入并防止它成为 swapped to disk。

fifo_init() 为 `size` 字节的 buffer 分配内存。 `size` 可以是任何 integer （即不必是 2 的幂， 2^N ），但建议使用系统认为 `int` 的倍数。

请注意，此函数可能需要几秒钟才能返回：对大部分物理 RAM 的请求可能会强制操作系统将其他进程的 RAM pages 交换到磁盘，或强制 disk cache flushing。在这两种情况下，fifo_init() 可能必须等待大量数据写入磁盘才能返回。

该函数在成功时返回零，否则返回非零。

4.7.2 fifo_destroy()

fifo_destroy(struct xillyfifo *fifo)——解锁后释放FIFO的内存，释放thread synchronization的资源。这个函数应该在主程序退出时调用，因为即使在当前的 Windows 实现中 thread synchronization 资源是自动释放的，但它们的 API 并不能保证这一点。

此函数是 void 类型（因此不返回任何内容）。

4.7.3 fifo_request_drain()

fifo_request_drain(struct xillyfifo *fifo, struct xillyinfo *info)– 提供 pointer 从 FIFO 读取数据作为 info->addr，并通知在 info->bytes 中从 pointer 开始可以读取多少字节。

`info` 结构**不能**与用于对 fifo_request_write() 的函数调用相同的结构。每个 thread 都应该为这个结构维护一个自己的局部变量。

重要的:

返回的字节数**并不**表示在 *FIFO* 中还有多少数据要读取：它也可能反映了直到 *FIFO* 的内存 *buffer* 结束时剩余的字节数。因此，当 *pointer* 接近 *buffer* 的末尾时，数字可能会显着降低。

该函数还将 fifo->position 设置为将 FIFO 的当前读取位置指示为 0 和 size-1 之间的值，其中 `size` 是赋予 fifo_init() 的值。非零 fifo->slept 表示调用时 FIFO 为空。

该函数返回允许读取的字节数（与 info->taken 相同）。但如果函数 fifo_done() 已被调用，而 FIFO 为空，则 fifo_request_drain() 返回零。

4.7.4 fifo_drained()

fifo_drained(struct xillyfifo *fifo, unsigned int req_bytes)——这个函数改变FIFO的状态来反映req_bytes字节的消耗。如果 fifo_request_write() 因为 FIFO 已满而处于休眠状态，它将被唤醒。

重要的:

*req_bytes***没有健全性检查**。用户应用程序有责任确保 *req_bytes* 不大于对 *fifo_request_drain()* 的最后一次函数调用返回的 *info->bytes*。

此函数是 void 类型（因此不返回任何内容）。

4.7.5 fifo_request_write()

fifo_request_write(struct xillyfifo *fifo, struct xillyinfo *info)– 提供 pointer 以将数据写入 FIFO 作为 info->addr，并通知 info->bytes 从 pointer 开始可以写入多少字节。

`info` 结构**不能**与用于对 fifo_request_drain() 的函数调用相同的结构。每个 thread 都应该为这个结构维护一个自己的局部变量。

重要的:

返回的字节数**并不**表示在 *FIFO* 中还有多少数据要写入：它也可能反映到 *FIFO* 的内存 *buffer* 结束时剩余的字节数。因此，当 *pointer* 接近 *buffer* 的末尾时，数字可能会显着降低。

该函数还将 fifo->position 设置为将 FIFO 的当前写入位置指示为 0 和 size-1 之间的值，其中 `size` 是赋予 fifo_init() 的值。非零 fifo->slept 表示 FIFO 在调用时已满。

该函数返回允许写入的字节数（与 info->taken 相同）。但如果函数 fifo_done() 已被调用，fifo_request_write() 将返回零，即使 FIFO 未满（没有意义将数据写入永远不会被读取的 FIFO）。

4.7.6 fifo_wrote()

fifo_wrote(struct xillyfifo *fifo, unsigned int req_bytes)——这个函数改变 FIFO 的状态以反映 req_bytes 字节的插入。如果 fifo_request_drain() 因为 FIFO 为空而处于休眠状态，它将被唤醒。

重要的:

*req_bytes***没有健全性检查**。用户应用程序有责任确保 *req_bytes* 不大于对 *fifo_request_write()* 的最后一次函数调用返回的 *info->bytes*。

此函数是 void 类型（因此不返回任何内容）。

4.7.7 fifo_done()

fifo_done(struct xillyfifo *fifo)——这个函数是可选的，如果threads（读或写）中的任何一个已经完成，它可以帮助应用程序优雅地退出。它只是在 FIFO 的结构中设置一个标志，并在两个 threads 处于睡眠状态时唤醒它们。通过这样做，如果 FIFO 为空，fifo_request_drain() 将返回零而不是休眠，并且 fifo_request_write() 无论如何都将返回零。

这样，这些函数的调用者就知道 FIFO 没有更多用处了，可能会充当必要的角色，这很可能会停止 thread 的执行。

当提供 pipe 的数据源结束（例如 EOF 到达）或数据消费者不再接受时（例如 broken pipe），调用此函数。

此函数是 void 类型（因此不返回任何内容）。

4.7.8 FIFO_BACKOFF define variable

有时让 FIFO 填满到最后一个字节是不可取的。尽管没有明显的理由避免这种情况，但可能希望在数据写入位置和读取位置之间保持一个小的间隙。

例如，FIFO_BACKOFF 可以设置为 8，因此写入 FIFO 的最后一个字节永远不会与第一个有效字节共享一个 64 位字以供读取。这是一个相当牵强的预防措施，但代价是 8 字节内存的低价。

使用 Xillybus 或 XillyUSB 时不需要此功能。

5 循环 frame buffers

5.1 介绍

在某些应用中，尤其是视频图像的实时处理中，往往需要维护多个buffers，使得每个buffer都有一个固定的大小。在视频处理应用程序中，每个这样的 buffer 都包含一个 frame。这允许根据需要跳过 frames 或多次重播它们。

在 frame grabber 应用程序中，可以通过跳过一个或多个 frames 直到有一个空的 buffer 来处理 overflow 条件。例如，在实时取景应用程序中，当查看窗口移动或调整大小时，可能会出现这种 overflow 情况。像这样丢弃 frames 可以防止来自视频源的连续数据流中断，同时保持较小的 latency。

在 frame replay 应用程序中（例如显示 live output 的屏幕），当没有更新的 frame 显示时，输出图像会重复。这解决了源（例如磁盘）暂时停止的情况，导致显示的图像在短时间内冻结。虽然不是完全优雅，但总比 stream 不同步要好。在许多情况下，图像重复机制虽然有些粗糙，但可以很好地克服 frame rates 的差异，特别是当输出的 frame rate 远高于输入的 frame rate 时（例如 30 fps 到 60 fps ）。

本节讨论如何修改 4.5 中介绍的 FIFO 演示应用程序以管理一组此类 buffers。

5.2 改编 FIFO 示例代码

维护 frame buffers 和 FIFO 的循环集之间有相似之处。事实上，如果 FIFO 中的每个字节都代表一个 frame buffer，那么在 FIFO 中读取或写入某个字节的准备就绪，就等同于对整个 frame buffer 的读取或写入准备。

例如，假设一个 frame grabber 应用程序，其中分配了四个 frame buffers 用于包含接收到的图像数据。进一步假设设置了一个四字节的 FIFO 来帮助管理这四个 frame buffers，如下所示：

接收数据的 thread 从第一个 frame buffer 开始，以循环方式继续到下一个 frame buffer。在开始写入新的 frame buffer 之前，此 thread 会检查四字节 FIFO 是否未满。完成一个 frame buffer 后，它会向 FIFO 写入一个字节，如果 FIFO 未满，则转到下一个字节。

消耗图像数据的 thread 以相同的顺序在 frame buffer 中循环。在尝试从新的 frame buffer 读取之前，它会检查四字节 FIFO 是否为空。当它完成了 frame buffer 并准备进入下一个时，它会从 FIFO 读取一个字节。

通过遵守这个约定，可以保证接收数据的 thread 永远不会超过尚未消费的 frame buffer，并且消费的 thread 永远不会尝试从包含无效数据的 frame buffer 中读取。事实上，FIFO 中的字节数代表了集合中有效 frame buffers 的数量。

请注意，写入和读取的字节值没有区别，因此实际上不需要分配这四个字节的内存并将数据存储在其中。只有 FIFO 的握手机制起作用。

因此，4.7 段中概述的 FIFO API 可以按原样采用：

- 使用 size 参数调用函数 fifo_init() 作为 frame buffers 的数量（回想一下 `size` 可以是任何 integer）。fifo_init() 将为 FIFO 分配和锁定内存，它永远不会被使用（因为每个字节只是象征一个 frame buffer）。这种内存浪费可以忽略不计，但可以删除代码中的相关部分以避免将来混淆。
- 调用函数 fifo_request_drain() 以获取要读取的 frame buffer。info->position 将包含要使用的 frame buffer 的索引（编号从 0 开始）。如果没有 frame buffer 准备好，fifo_request_drain() 将休眠，直到有。
- 从 buffer 读取后，调用函数 fifo_drained()，bytes_req=1。
- thread 以相同的方式调用函数 fifo_request_write() 和 fifo_wrote() 写入 frame buffers。
- FIFO_BACKOFF 应设置为零。frame buffers 的这个特性没有意义。

5.3 丢弃和重复 frames

让我们以 image frames 的连续源为例，它永远不会达到 overflow 的状态，因为数据消费者可能并不总是足够快地收集数据。

这个想法是为了防止 thread 上的阻塞，它将数据从数据源传输到 frame buffers。为此，应为每个传入的 frame 循环以下序列：

- 调用函数 fifo_request_write() 找出要写入的 frame buffer
- 写入 info->position 指向的 frame buffer

- 完成写入后，再次调用函数 fifo_request_write()。此函数调用肯定不会休眠（block），因为自上次调用以来，没有报告 buffer 已写入。
- 如果 fifo_request_write() 刚刚返回一个大于 1 的值，则调用函数 fifo_wrote()（当然是 req_bytes=1）。对 fifo_request_write() 的后续函数调用肯定不会休眠（block），因为有多个 buffer 空闲，并且只消耗了一个。事实上，对 fifo_request_write() 的下一个函数调用可以通过选择下一个 frame buffer 来代替。
- 另一方面，如果 fifo_request_write() 只返回 1，则不要调用函数 fifo_wrote()。相反，在执行下一个循环以接收传入数据时再次使用当前 buffer，或者只是将整个 frame 从数据源排空到没有特定目的地。

由于这种用法可以防止阻塞，因此可以在 fifo_request_write() 的实现中删除 while() 循环，因为它永远不会被调用。通过删除相关的 event 及其初始化和销毁代码，可以进一步减少代码。将它们留在代码中的影响很小，因此这种优化主要是保持代码可读性的问题。

可以采取类似的方法在 thread 上重复 frames 从 FIFO 写入：在调用函数 fifo_drained() 之前再次调用函数 fifo_request_drain()，如果返回小于 2，则重复当前 frame。

6 具体的编程技术

6.1 Seekable streams

同步 Xillybus stream 可以配置为 seekable。 stream 的位置在 FPGA 中以单独的线作为地址呈现给 application logic，因此在 FPGA 中连接内存阵列或 registers 很简单，如 demo bundle 和示例代码所示。

此功能特别适用于在 FPGA 中设置 control registers。 stream 的同步特性确保在低级 I/O 函数返回之前设置 FPGA 中的 register。

下面的代码片段演示了如何将 `len` 字节的数据写入内存中的地址 `address` 或 FPGA 中的 register space，假设这两个变量是预先设置的。

```
int rc, sent;

if (_lseek(fd, address, SEEK_SET) < 0) {
  perror("Failed to seek");
  exit(1);
}

for (sent = 0; sent < len;) {
  rc = _write(fd, buf + sent, len - sent);

  if ((rc < 0) && (errno == EINTR))
    continue;

  if (rc <= 0) {
    perror("Failed to write");
    exit(1);
  }

  sent += rc;
}
```

fd 也被假定为从对 _open() 的函数调用返回的值，其中文件被打开以进行写入或读写，并且 buf 指向包含要写入的数据的 buffer。

此示例是段落 3.3 中所示示例的扩展。

这段代码中唯一特别的是对 _lseek() 的函数调用，它设置了地址。在调用 _lseek() 函数时，只能将 SEEK_SET 选项用作第三个参数。

后续函数调用会根据 I/O stream 的位置更新地址，因此在调用函数 _lseek() 后多次连续写入没有限制。

对于在 FPGA 中作为 16 位或 32 位字访问的 streams，给 lseek() 的地址必须分别是 2 或 4 的倍数。 FPGA 中提供给 application logic 的地址始终保持为 stream 的 I/O 位置（最初为 lseek() ）分别除以 2 或 4。对于更宽的词，同样的对数规则适用。

_tell() 函数可能会返回 stream 中的正确位置（即当前地址），但它不是此信息的可靠来源。如有疑问，请再次调用函数 _lseek()。

_lseek() 可以以相同的方式用于读取数据。请参阅演示应用程序包中的 memwrite.c 和 memread.c（以及它们在 Getting started with Xillybus on a Windows host中的描述）。

6.2 双向同步 streams

在某些应用中，需要同步多个 streams，可能方向相反。例如，可以在 host 上实现无线电传输系统，从连接到 RF 接收器的 A/D converter 接收数字样本。同样，它可能会将数字样本发送到连接到 RF 发射器的 D/A converter。在这种情况下，通常需要生成用于传输的数字样本，以便知道与接收样本相关的传输时间。知道接收信号的准确时间也很重要。

幸运的是，可以使用简单的 FPGA logic 来实现。一个这样的解决方案是忽略接收到的数字样本，直到要传输的第一个样本到达 FPGA:

host 首先打开 stream 以从 FPGA 读取样本。这个 stream 在这个阶段是空闲的，因为 FPGA 丢弃了它的接收样本。然后 host 打开 stream 写入样本以传输到 FPGA，并开始向其写入数据。当第一个样本到达 FPGA 时，它会停止忽略接收到的样本，并开始将它们发送到 host。

因此，将从 FPGA 读取的第一个样本将与写入 FPGA 的第一个样本匹配。因此，host 上的应用程序可以将要传输的任何样本的 timing 与接收的任何样本匹配，只需匹配它们在相应 stream 中的位置即可。 FPGA 中的 latency 以及 A/D 和 D/A 的延迟可能需要稍作修正，但这样的 latency 是恒定的并且是已知的。

streams 必须始终保持连续。 4 部分讨论了如何实现这一点。

如果保持发送和接收之间的相对时间关系就足够了，则该解决方案是令人满意的。当样本需要与外部事件或其他时间参考同步时，可以根据需要调整跳过样本的相同原理以实现所需结果。

Xillybus FPGA designer's guide在名为 “Monitoring the amount of buffered data” 的部分中讨论了在任何给定时间监视 driver 的 buffers 中保存的数据量。

6.3 分组通信

某些应用需要将 data stream 分成不同长度的数据包。建议的解决方案使用两个独立的streams，并且不需要数据的发送者在开始通过通道提交数据包时知道数据包的长度。

简单地通过在一个 stream 上一个接一个地传输它们来解决具有固定且已知长度的数据包的琐碎情况。另一端的接收器只为每个数据包读取固定数量的字。这是 video frame grabber 或 video replay 应用程序中的典型解决方案。

对于可变长度数据包的情况，让我们看一下 upstream 应用程序，其中 FPGA 向 host 发送字节数据包。假设 FPGA 只有在最后一个字节到达时才知道数据包的长度。

FPGA端（即发送端）的实现如下:

- FPGA 将数据包中的所有字节写入第一个 Xillybus stream。
- FPGA 在写入数据包中的第一个字节时重置字节计数器，并在它写入的每个额外字节时递增它。
- 当数据包中的最后一个字节被写入时，FPGA 将计数器的值发送到第二个 Xillybus stream。它包含数据包的长度（减一）。

该解决方案的一个重要属性是 FPGA 在发送之前不需要存储整个数据包。它只是在数据到达时传递它。

host 上的用户应用程序运行如下循环:

- 从第二个 stream 读取一个字，包含下一个数据包中的字节数。
- 如有必要，为请求大小的 buffer 分配内存。
- 将给定数量的字节读入专用于来自第一个 stream 的数据包的 buffer。

请注意，host 在访问数据之前获取要读取的字节数，但 FPGA 以相反的顺序将这些字节数写入 streams。使用单独的 Xillybus streams 可以实现这种逆转。

当数据包从 host 发送到 FPGA 时，类似的安排也适用。使用两颗streams，一颗用于数据，一颗用于字节计数的原则仍然存在。 FPGA 的 application logic 现在可以先从一个 stream 读取字节数，然后再从另一个 stream 获取数据。

这种安排也可扩展为在非数据 stream 中传递其他元数据，例如数据包的目的地或某些网络中的路由（有时不知道，当第一个字节到达时）。

6.4 模拟 hardware interrupts

在小型微控制器项目中，通常使用 hardware interrupts 来提醒软件发生了某些事情，并且软件需要采取一些行动。当软件在 Windows 中作为 userspace 进程运行时，hardware interrupts 是不可能的。

对于基于 Xillybus 的系统，建议的解决方案是分配一个特殊的 stream 来承载消息。在最简单的形式中，hardware interrupt 是通过在专用 stream 上发送一个字节来模拟的。

在 host 端，userspace application 尝试从 stream 读取数据。结果是，当没有 “interrupt” 发出信号时，应用程序会休眠（ blocking ）直到一个字节到达并将其唤醒。应用程序处理该事件，然后尝试从专用 stream 读取另一个字节，因此在必要时再次进入睡眠状态，依此类推。

为了实现主应用程序和 interrupt routine 之间的正确交互，这个专用的 stream 可以由单独的 software thread 或进程读取。通过这种安排，主代码流与从专用消息 stream 读取的 thread 无关，后者根据发送的消息休眠和唤醒。

此方法的一个变体使用传输字节的值来传递有关模拟 interrupt 的性质的信息。此外，如果在实现中有意义的话，每条消息都可以长于一个字节。

这种方法看起来可能会浪费 logic 资源，但 Xillybus 最初的设计目的是不为每个添加的 stream 消耗太多 logic，以便使这样的解决方案变得明智。

6.5 Coprocessing/ Hardware acceleration

Coprocessing（也称为 *hardware acceleration*）是一种技术，它允许应用程序利用 logic fabric 的灵活性来更快、更便宜地执行某些操作，并且比给定的 processor 具有更低的能耗或更高效。无论动机是什么，高效的数据传输流对于使 coprocessing 成为合格的解决方案至关重要。

重要的是要认识到基于 coprocessing 的应用程序中的数据流与常见的编程数据流根本不同。为了说明这种差异，让我们以一个需要计算 floating point 表示形式的数字的平方根的计算机程序为例。

程序员的直接方法是将数字作为参数传递给 sqrt()，调用它，然后等待函数返回。

假设希望在 FPGA 的 logic fabric 中计算平方根。一个常见的错误是将 sqrt() 替换为特殊函数，该函数将计算值发送到 FPGA，等待它完成，然后返回结果。尽管这确实是 sqrt() 的简单替代品，但它很可能会比原来的 sqrt()更慢且效率更低：数据在两个方向上穿过 bus 所需的时间，加上它的时间FPGA 进行计算所需的时间，可能比 sqrt() 所需的 processor cycles 长得多。话虽如此，如果数据流设计正确，在 FPGA 上计算平方根会快得多。

为了克服 bus 和 FPGA 的 logic 强加的 latencies，需要重新组织软件。尤其是具有单个 thread 的程序中的任务需要拆分为两个或多个 threads（或进程）。如果多个 threads 是不可能的或不可取的，则可以使用其他编程技术来模仿 multi-threading 的行为，但编程范式仍然是 multi-threaded。

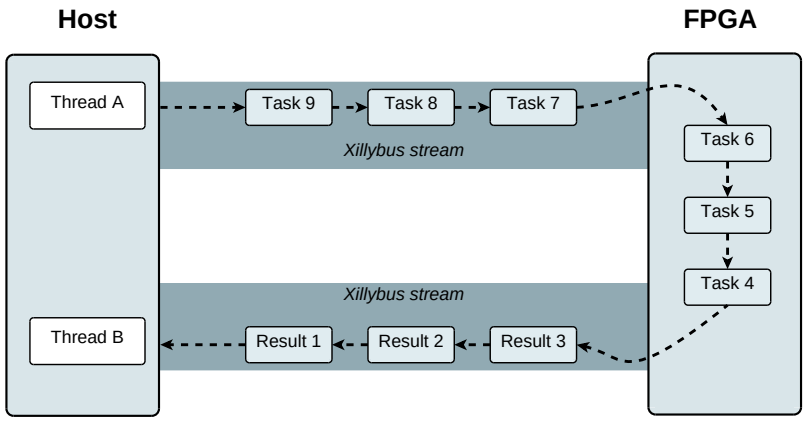
回到 sqrt() 的例子，对这个函数的调用分为两个 threads： 第一个 thread 将用于平方根计算的数据发送到硬件（或表示操作请求的某种其他形式的数据结构）。第二个 thread 从硬件接收结果并从算法中的那个点继续处理。

这在查看单个数据时似乎没有多大意义，但 coprocessing 的动机意味着要处理许多数据项。所以第一个 thread 发送一个数据流进行计算，第二个 thread 接收一个结果流。

pipelining 的这种技术最大限度地减少了硬件 latency 的影响，因为 threads 都没有有

效地等待这个 latency 的时间。相反，latency 会影响两个 threads 之间的处理项目数量——但吞吐量仅取决于两个 threads 和 FPGA logic 的处理能力。

下面的概念图总结了这个想法。



sqrt() 的加速计算是一个相对简单的例子，但它涵盖了使用 coprocessing 的大部分挑战。几乎总是需要重写大部分计算机程序，以便一切都由 pipeline 的数据流驱动。

另一个需要注意的问题是，由于 Xillybus 可与 read() 和 write() 一起使用，因此在将数据项写入 stream 到 FPGA 之前将它们分组以进行计算可能是有益的。同样，尝试在每个 read() 调用中读取多个结果项可能会提高性能。这背后的基本原理是 read() 和 write() 是具有一定开销的 system calls。如果数据元素很小并且以高速率传输，则这些 system call 的开销可能很大。 sqrt() 就是一个很好的例子：double float 通常有 8 个字节长。这种长度的 I/O system call 效率很低，因此为单个 system call 连接多个 double float 元素会有所不同。

还值得一提的是，并非所有应用程序都涉及恒定长度的数据块。例如，使用 coprocessing 计算任意字符串的 hashes （例如 SHA1 ）很可能涉及不同长度的数据元素进行处理。第 6.3 节为此提出了一个解决方案。

7 Hibernation

“Hibernation”、“standby” 和 “sleep” 是计算机省电状态的常用名称。在 hibernation 上，计算机将其 RAM 的内容保存到硬盘，并有效关闭电源。当命令再次启动时，它会迅速恢复到与进入 hibernation 时相同的状态。

对于基于 FPGA 的外围设备，hibernation 需要格外小心，因为 device driver 不知道 FPGA 将要经历什么，尤其是它是否会保持通电状态。

出于这个原因，XillyUSB 设备（即通过 USB 电缆连接的 Xillybus IP core）在计算机进入睡眠状态之前有效地断开连接，就好像它被物理拔出一样。如果计算机再次唤醒时设备存在，则将其视为刚刚连接到计算机。如果应用程序可以容忍这种行为，那么 hibernation 就可以了。请注意，如果 host 连接了多个 XillyUSB 设备，则其文件名中出现的索引在唤醒后可能会发生变化。

本节的其余部分讨论 Xillybus 而非 PCIe。

如果设备由外部电源供电，它将在计算机的其余部分关闭时保持开机状态。但目前尚不清楚 PCIe bus 本身是否会保持通电状态，或者是否会在某个阶段重置。

如果它从计算机的电源中获取电源，它很可能会断电。在这种情况下，FPGA 可能会在电源恢复时从 flash memory 重新加载其 bitstream，并继续作为 bus 上的 PCIe 卡，或者如果它依赖于手动加载 bitstream，它可以从 bus 完全消失。

请注意，如果 PCIe 接口在计算机关闭时突然消失，Windows 可能无法从 hibernation 唤醒。所以如果FPGA依赖于电脑的电源，肯定应该开机自动重新加载bitstream。

Xillybus device driver 应对这种不确定性，当操作系统通知它计算机将要调用 hibernation 时，就像 driver 即将完全卸载一样：它在所有待处理的 I/O 请求上生成错误，强制关闭所有打开的文件立即释放所有资源，移除 device files 并使 FPGA 中的 Xillybus IP core 进入静止状态。

当计算机从 hibernation 恢复时，driver 初始化为在 boot 期间。

尽管该过程的可靠性已得到证实，但仍有两个可能的问题：

- driver 可能无法为其 buffers 分配足够的 RAM，尤其是当 buffer 的总大小超过几兆字节时。当 driver 加载到计算机的 boot 上时，物理映射中的大块连续内存比计算机运行一段时间后更容易分配。 hibernation之后保留了操作系统的内存映射，所以内存的fragmentation也保留了。因此，在系统唤醒期间再次加载 driver 时，在 hibernation 之前释放的连续物理内存块可能不再连续。
- 即使 user space 应用程序应该准备好处理来自任何 I/O 函数调用的错误，由 hibernation 导致的意外错误可能会暴露一个罕见的隐藏错误。

总体结论是，处理 hibernation 的最佳方法是在 PCIe bus 上连接了基于 FPGA 的外围设备的计算机上避免它。这很容易通过设置计算机的省电属性来完成，特别是关闭操作系统可能认为空闲的自动 hibernation。这也将防止计算机由于缺乏人机交互而进入睡眠状态，而实际上它正在执行一项重要的无人值守任务。

A

内部结构：streams 是如何实现的

A.1 介绍

尽管使用 Xillybus 不需要对其实现细节有任何了解，但一些设计人员更愿意了解幕后发生的事情，无论是出于好奇还是为了验证某个解决方案的资格。

本节概述了基于 DMA buffers 创建连续 streams 的主要技术。它适用于 PCIe / AXI 的 Xillybus，但不适用于使用另一种机制的 XillyUSB。

Xillybus 的设计目标是使底层机制对用户透明，并且在很大程度上没有理由了解它们。请在阅读下面的技术细节时牢记这一点，因为为了将 Xillybus 用作 IP core，它们很可能是不必要的。这部分更多的是关于它是如何工作的，而不是关于用户需要知道的事情。

下面有两个主要部分，一个用于 upstream 流程，另一个用于 downstream。由于在两个方向都采用了类似的技术，因此一个部分的大部分内容是另一个部分的重复。

为简单起见，描述集中在异步 streams 上，除非另有说明。 end-of-file 信号以及 non-blocking I/O 的选项在此不作讨论。

A.2 “Classic” DMA 与 Xillybus

传统上，硬件和软件之间的数据传输采用多个具有固定大小的 buffers 的形式。数据以固定长度组织到 buffers 中，可能会也可能不会完全填充。每次 buffer 准备就绪时，都会向另一端发送某种信号。例如，如果硬件已完成对 buffer 的写入，它可能会向 processor 发送 interrupt 以通知软件数据已准备好进行处理。软件使用数据，并通知硬件 buffer 可以再次写入，通常是通过写入一些内存映射的 register。通常，双方都以 round-robin 方式访问 buffers。

Xillybus 在 FPGA 和软件端都向用户界面提供了连续的 stream 传输。在底层，Xillybus

使用了传统的 round-robin 范式和一组 DMA buffers。

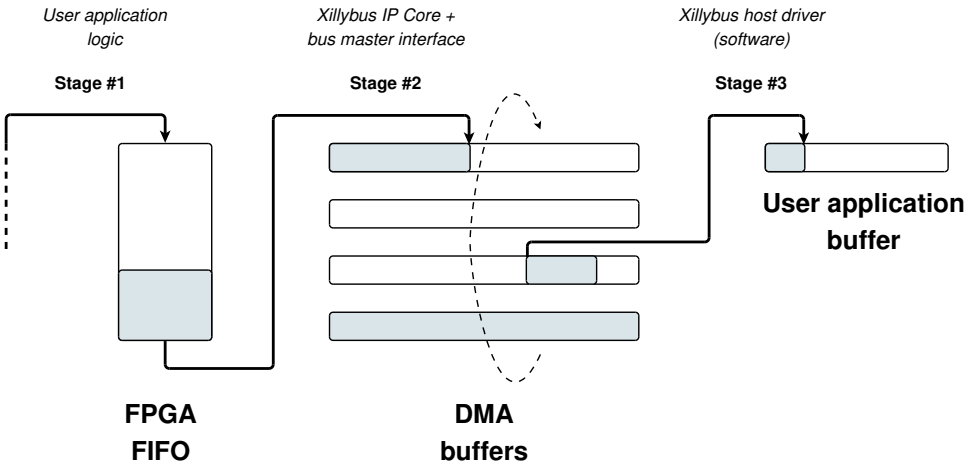
然而，下面描述的技术用于创建连续 stream 的错觉，以便用户可以忽略底层数据传输的存在。特别是，即使应用程序包含以固定大小的块发送数据，也无需将 DMA buffers 的大小与应用程序数据匹配，如下所述。

A.3 FPGA 至 host (upstream)

A.3.1 概述

下图描绘了 FPGA 到 host (upstream) 方向的流动。阴影区域表示在各个存储元件中尚未被消耗的数据。

在此示例中，显示了四个 DMA buffers，即使这些数量可以在 IP Core Factory 中配置。



数据分三个阶段流向 host，如下所述。

A.3.2 #1 阶段：Application logic 到中间 FIFO

FPGA 中的 user application logic 将数据推送到连接 user application logic 和 Xillybus IP core 的 FIFO 中。除了尊重 FIFO 的 “full” 信号以避免 overflow 之外，对何时推送或推送多少数据没有要求。

A.3.3 #2 阶段：中间 FIFO 到 DMA buffer

在这个阶段，Xillybus IP core 将数据从 FIFO 复制到 host 的 RAM 中的一个 DMA buffer。为此，core 使用一些 bus master 接口（PCIe、AXI4 等）将数据直接写入 host 的内存，而无需 host 的 processor 的干预。

在 host 的 RAM 中分配了一个 DMA buffers 池。每个 DMA buffer 的生命周期就像在许多类似的设置中一样：一开始，所有 DMA buffers 都是空的，在概念上属于硬件。硬件以round-robin的方式向buffers写数据：当对某个buffer写完后，通知hostbuffer可以使用了（buffer可以说是交给host了），之后它继续写在下面的buffer上。然后 host 可能会消耗 buffer 中交给它的数据，之后它会通知硬件 buffer 可以再次写入（host 将 buffer 返回给硬件）。

#2 阶段的数据流由 FIFO 的 “empty” 信号和 DMA buffers 池中的可用空间控制。当 Xillybus IP core 检测到来自 FIFO 的低 “empty” 信号，并且某些 DMA buffer 中还有空间时，它会从 FIFO 获取数据并将其写入 DMA buffer。当 FIFO 再次变空，或者任何 DMA buffer 中没有空间时，IP core 的内部状态机会暂时停止获取数据，然后从 DMA buffer 中停止的地方继续。

当数据流停止时，IP core 可能正忙于其他活动，例如代表其他 stream 复制数据（即耗尽另一个中间 FIFO）。因此，在 FIFO 将 “empty” 信号变为低电平的时间与恢复从中获取数据之间可能存在随机延迟。这种延迟各不相同，但总体而言，IP core 保证 512 个字的 FIFO 不会达到 overflow 的状态（只要平均速率在限制范围内）。

每个 DMA buffer 可以在交付给 host 之前完全填充，也可以提交给 host 部分填充。移交部分填充的 buffer 的条件将在后面详细说明（第 A.3.5 节），因为它们需要对软件的行为有所了解。

同步 streams 的情况非常相似，只是 Xillybus IP core 在从中间 FIFO 获取数据之前，会等待对一定量数据的显式请求。

A.3.4 Stage #3：DMA buffer 到用户软件应用程序

此阶段通过响应 read() system calls（或 Microsoft Windows 上的对应 IRP）在 host 上的 Xillybus 的 driver 上实现。根据公认的 API，read() 请求包括由 user application 提供的 buffer，以及 buffer 的大小，这也是要读取的最大字节数。函数调用可能在读取最大字节数（完全实现）或更少后返回。

driver 首先检查移交给它的 DMA buffers，以确定 DMA buffers 中是否有足够的数据供消费，以允许完全满足 read() 请求。如果是这样，它将数据复制到用户的 buffer，可能会将 DMA buffers 返回到硬件，然后从 system call 返回。

否则，用于 read() 函数调用的标准 API 允许 driver 以少于请求的字节数返回，或者等

待（休眠）任何时间段。 driver 被设计为不会在数据很少的情况下经常返回（这可能会导致大量 read() 函数调用每个数据很少，从而浪费 CPU 周期），但也避免了不必要的 latency。如果 DMA buffers 中的数据少于 read() 函数调用所需的数据，那么困境是怎么办：返回部分满足或等待（以及等待多少）。

选择的策略是等待 10 ms 获取更多数据，然后返回可用的任何数据（或者如果没有可用数据，则无限期等待，如标准 API 所要求的那样）。如果 read() 函数调用者始终请求的数据多于可用数据，则这会产生相当快的返回时间，但会将开销限制为每秒 100 次 read() 函数调用。

这并不是说 read() 函数调用必然有 10 ms 的 latency：如果 user space application 事先知道应该准备多少字节，它可能请求不超过该数字。通过这样做，它确保了 latency 在微秒数量级。

然而有一个棘手的部分：host 知道已移交给它的 DMA buffers，但可能存在部分填充的 DMA buffer，而 host 不知道。因此，如果考虑到部分填充的 DMA buffer，实际上可能有足够的数据来完全完成 read() 函数调用。

为了正确处理这种情况，driver 检查丢失的字节数是否适合部分填充的 buffer。如果确实如此，它会通知硬件多少数据就足够了。然后 driver 开始 10 ms 等待。这使硬件有机会立即发送部分填充的 buffer，如果它确实允许完全完成 read() 函数调用。

如果部分填充的 buffer 达到必要的量（可能立即），硬件将其交给 host，然后立即完成 read() 函数调用。

当 10 ms 周期结束时，driver 会返回它所拥有的尽可能多的数据。如果根本没有数据，driver 会向硬件发送一个请求以传递它拥有的任何部分填充的 buffer。目的是在有任何数据时立即返回，因为 10 ms 周期已经结束。

在所有情况下，当 DMA buffer 被完全消耗时，driver 会将其返回给硬件（即通知硬件它可以再次写入）。

关于同步 streams 的几句话： 流程在原理上是相同的，只是在调用 read() 函数调用时数据在 DMA buffers 中永远不可用。这是因为只有在收到指令时，才允许硬件从 FPGA 的 FIFO 复制数据。因此，同步 streams 的 read() 函数调用涉及通知硬件它应该复制的数据量。等待机制保持不变：首先是 10 ms，然后需要任何部分填充的 buffer。

A.3.5 buffers部分满载交接条件

部分满载的buffers交接的案例可以从上面推导出来，为方便起见在此列出。

一般规则是，如果硬件已被告知此类提前提交将导致立即返回 read() 函数调用，则将部分 buffer 移交给 host，这发生在以下三种情况之一：

- host 当前正在处理一个 read() 函数调用，当当前部分填充的 buffer 移交时，该函数调用将完全实现。
- read() 函数调用位于零字节处，并且已达到时间限制（即 10 ms ）。
- 仅在同步 streams 上：当硬件完成获取 host 请求的数量时。

请注意，当 FIFO 变为空时，它本身并不是DMA buffer 提交的原因。

A.3.6 例子

让我们考虑以下 8 位异步 stream 的简单案例。假设 stream 开始时不包含任何数据，之后 FIFO 被单个元素（即一个字节）填充。 host 上的应用程序然后调用 read() 函数，请求一个字节。这是一个可能的事件链：

- Xillybus IP core 检测到低 “empty” 信号，因此从 FIFO 获取单个字节，之后它再次变为空。
- 字节与 DMA 一起写入 DMA buffer 的第一个位置。 host 未收到通知，因为 buffer 未满。
- 在 host 上调用 read() 函数调用，请求一个字节。
- driver 没有 DMA buffer 可从中获取数据：唯一包含数据（一个字节）的 DMA buffer 只有硬件知道。
- driver 检测到它需要的数据量小于 DMA buffer 的大小，因此告诉硬件交出部分填充的 buffer，如果它至少有一个字节。
- driver 开始 10 ms 睡眠，等待某事发生。
- 硬件立即响应，将部分填充的 buffer 移交给 host。
- driver 立即唤醒，将请求的一个字节复制到 read() 函数调用提供的 buffer 中，然后返回。

这个简单的示例演示了 read() 函数调用如何几乎立即返回，即使数据的大小明显小于 DMA buffer。

让我们再看一下这个例子，有一个小的区别：read() 函数调用请求两个字节，尽管只有一个字节被写入 FIFO。顺序如下。

- Xillybus IP core 检测到低 “empty” 信号，因此从 FIFO 获取单个字节，之后它再次变为空。

- 字节与 DMA 一起写入 DMA buffer 的第一个位置。 host 未收到通知，因为 buffer 未满。
- 在 host 上调用 read() 函数调用，请求两个字节。
- driver 没有 DMA buffer 可从中获取数据：唯一包含数据（一个字节）的 DMA buffer 只有硬件知道。
- driver 检测到它需要的数据量小于 DMA buffer 的大小，因此告诉硬件交出部分填充的 buffer，如果它至少有两个字节。
- driver 开始 10 ms 睡眠，等待某事发生。
- 硬件什么也不做，因为它在 DMA buffer 中只有一个字节，但需要两个。
- driver 在 10 ms 之后醒来，什么都没有。它向硬件发送请求以尽快交出部分填充的 buffer，除非它是空的。
- 硬件立即响应，将部分填充的 buffer 移交给 host。
- driver 立即唤醒，将请求的一个字节复制到函数调用者的 buffer 中，然后返回。

第二个示例显示了在实际上只有一个字节时请求两个字节的结果：函数调用仅在 10 ms 之后返回，只有一个字节。但是请注意，在大多数实际情况下，这种延迟不会被注意到。

A.3.7 实际结论

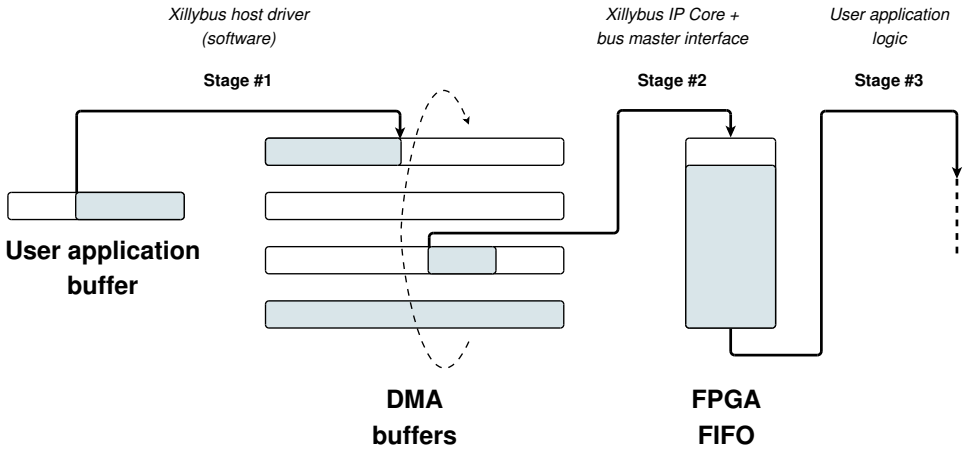
- 即使应用程序级数据总是由 N 字节的块组成，也没有理由以任何方式调整 DMA buffer 大小。 user application software只需要保证read()函数调用请求的数据量完全符合需要，部分buffer机制保证数据已经压入FPGA的FIFO时函数调用返回，latency很低。
- 即使对于连续 streams 的数据，也可以通过使用小的 buffers 进行 read() 函数调用来减少 latency，但代价是增加了操作系统的开销。无论 DMA buffers 的大小如何，latency 仅取决于 read() 函数调用中请求的数据速率和字节数。减小 DMA buffer 大小无济于事，因为如果 read() 函数调用不能完全完成 read() 函数调用，它将继续等待 10 ms。
- 如果 10 ms 是可接受的 latency，则优化没有意义，因为 read() 函数调用保证在此时间段后返回，除非根本没有数据可返回。

A.4 Host 至 FPGA (downstream)

A.4.1 概述

下图描述了 host 到 FPGA (downstream) 方向的数据流。阴影区域表示在各个存储元件中尚未被消耗的数据。

在此示例中，显示了四个 DMA buffers，即使这些数量可以在 IP Core Factory 中配置。



和以前一样，数据从 host 流向 FPGA 分为三个阶段，如下所述。

A.4.2 Stage #1：DMA buffer 的用户软件应用程序

此阶段通过响应 write() system calls（或 Microsoft Windows 上的对应 IRP）在 Xillybus 的 driver（在 host 上）上实现。根据完善的 API，write() 函数调用请求包括由用户应用程序提供的 buffer，以及 buffer 的大小，这也是要写入的最大字节数。函数调用可能在写入最大字节数（完全实现）或更少后返回。

在 host 的 RAM 内存中分配了一个 DMA buffers 池。每个 DMA buffer 的生命周期都像许多类似的设置：一开始，所有 DMA buffers 都是空的，在概念上属于 host。host以round-robin的方式向buffers写入数据：当对某个buffer的写入完成后，通知硬件buffer可以使用（buffer是交给硬件了），之后它继续写在下面的buffer上。然后硬件可能会消耗 buffer 中的数据，之后它会通知 host buffer 可以再次写入（硬件将 buffer 返回给 host）。

Xillybus 的 driver 通过尝试将尽可能多的数据复制到 DMA buffers 来响应 write() 函数调用。当一个DMA buffer被完全填满后，就交给硬件了，即host通知硬件buffer可以被

消费，并保证在硬件将buffer还给host之前不会再次写入。

如果 driver 在 DMA buffer 空间用完之前设法写入了至少一个字节，则 write() 函数调用返回写入的字节数。否则无限期地等待（通过休眠，即 “blocking” ），直到 DMA buffer 可用于写入，然后它将尽可能多的数据写入 DMA buffer 并返回。

请注意，如果 DMA buffer 被部分填充，它不会在 write() 函数调用结束时移交给硬件，因此一个 DMA buffer 中可能存在硬件不知道的数据。 “flush” 操作交出部分填充的 buffer，它发生在以下四种情况中的任何一种情况下：

- 显式 flush，由使用零字节写入的 write() 函数调用引起。这个 write() 函数调用立即返回（即它不等待数据被 FPGA 使用）。
- 在最后一次 write() 函数调用之后 10 ms 启动自动 flush。
- 当文件关闭时，会出现 flush。在这种情况下，close() 函数调用会等待最多一秒钟，直到数据完全被 FPGA 消耗，然后才返回。
- 在同步 streams 上，对 write() 的每个函数调用都以 flush() 结束，它会无限期地等待，直到 FPGA 完全消耗数据。

请注意，具有零长度 buffer 的 write() 函数调用强制显式 flush，确保已写入的所有数据都可用于 FPGA。但是，它不会向应用软件指示 FPGA 何时使用数据。如果需要这种同步，则应使用同步 stream。

A.4.3 #2 阶段：DMA buffer 到中间 FIFO

在这个阶段，Xillybus IP core 将 host 的 RAM 中的 DMA buffers 中的数据复制到 FPGA 中的 FIFO 中。为此，core 使用一些 bus master 接口（PCIe、AXI4 等）直接从 host 的内存中读取数据，而无需 host 的 processor 的干预。

#2 级的数据流由 FIFO 的 “full” 信号和属于 FPGA 的 DMA buffers 池中的数据可用性控制。当 Xillybus IP core 检测到来自 FIFO 的低 “full” 信号，并且某些 DMA buffer 中有数据准备好时，它会从 DMA buffer 获取数据并将其写入 FIFO。当 FIFO 再次变满或 DMA buffers 为空时，IP core 的内部状态机会暂时停止获取数据，然后从 DMA buffer 池中的中断处继续。

当数据流停止时，IP core 可能正忙于其他活动，例如代表其他 stream 复制数据（即填充另一个中间 FIFO）。因此，在 FIFO 将 “full” 信号变为低电平的时间与恢复数据复制之间可能存在随机延迟。这种延迟各不相同，但总体而言，IP core 保证 512 字的 FIFO 足够深。

硬件当然知道部分填充的 DMA buffers，并跟踪每个包含多少数据。

A.4.4 #3 阶段：中间 FIFO 到 application logic

FPGA 中的 user application logic 从连接 user application logic 和 Xillybus IP core 的 FIFO 获取数据。除了尊重 FIFO 的 “empty” 信号以避免 underflow 之外，对何时或获取多少数据没有要求。

A.4.5 一个例子

让我们考虑以下 8 位异步 stream 的简单案例。假设 stream 开始时不包含任何数据，之后 host 的应用程序将单个字节写入 device file。

事件顺序如下：

- driver 的 write() 函数调用是通过写入一个字节的请求来调用的。
- 由于 stream 不包含数据，显然 DMA buffers 中有空间。因此，driver 将字节复制到第一个 DMA buffer 中并返回。
- 10 ms 期间没有任何反应。
- autoflush 机制在 10 ms 之后触发，导致 driver 将 DMA buffer 连同它包含一个字节的信息一起交给硬件。
- Xillybus IP core 从 DMA buffer 读取字节并将其写入中间 FIFO。
- application logic 可以随意从 FIFO 读取字节。

A.4.6 实际结论

- 即使应用程序级数据总是由 N 字节的块组成，也没有理由以任何方式调整 DMA buffer 大小。 user application software 只需要请求 flush 的数据，并在每个块的末尾使用请求零字节的 write() 函数调用。一个微秒数量级的latency就是这样实现的。
- 即使对于连续的 streams 数据，latency 也可以通过使用小的 buffers 进行 write() 函数调用来减少 latency，然后是 flush（零字节的 write() 函数调用），但会增加操作系统的开销。无论 DMA buffers 的大小如何，latency 仅取决于 flush 请求之间的数据速率和数据量。
- 如果事先知道 flush 总是出现在给定的数据块之后，那么减小 DMA buffer 的大小是有意义的，因此没有 DMA buffer 被填充超过某个级别。然而，这样做的唯一好处是在 host 上节省了大量的 RAM，这不太可能是显着的。

- 如果 10 ms 是可接受的 latency，那么优化没有意义，因为 autoflushing 机制在 10 ms 没有活动之后启动。