(机器翻译成中文)

Getting started with Xillybus on a Linux host

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 This document may also be slightly outdated in relation to the original.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

这是安装 driver 以在 Linux host上使用 Xillybus / XillyUSB 以及试用 IP core的基本功能的演练指南。

为简单起见，假设 host 是一台具有 compilation 功能的成熟计算机。 embedded 平台的过程类似，但有一些直接的区别（例如 cross compilation）。

本指南假定 FPGA 加载了 demo bundle的 bitstream，包含 Xillybus 或 XillyUSB的 IP core，并且已被 host 识别为外围设备（通过 PCI Express、 AXI bus或 USB 3.x，视情况而定）。

这些文档之一概述了实现此目标的步骤（取决于所选的 FPGA）：

- Getting started with the FPGA demo bundle for Xilinx
- Getting started with the FPGA demo bundle for Intel FPGA
- Getting started with Xillinux for Zynq-7000
- Getting started with Xillinux for Cyclone V SoC (SoCKit)

host driver 生成 device files，其行为类似于 named pipes：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类似于 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。

一个 driver 二进制文件涵盖了 PCIe 和 AXI 传输的所有 Xillybus IP cores 。另一个 driver 被指定为 XillyUSB。 streams 及其属性在加载到 host的操作系统时由 driver 自动检测，并相应地创建 device files 。这些类似 pipe的 device files 显示为 /dev/xillybus_*something* （或 /dev/xillyusb_*something* 与 XillyUSB）。

但请注意， PCIe / AXI 接口有一个 driver ， XillyUSB有另一个。

有关 host 相关主题的更深入信息，请参阅 Xillybus host application programming guide for Linux。

2

安装 host driver

2.1 安装 Xillybus的 driver的阶段

安装 Linux kernel driver 包括以下步骤:

- 检查是否需要安装。如果没有，请跳过下面的其他步骤，可能是最后一步（复制 udev 文件）
- 检查先决条件（已安装 compiler 和 kernel headers ）
- 解压下载的文件
- 运行 kernel module 的 compilation
- 安装 kernel module
- 复制 udev 配置文件以使任何用户都可以访问 Xillybus device files

这是使用命令行界面 (“Terminal”) 完成的。 Appendix A 中简短的 Linux 生存指南可能对那些不太熟悉此界面的人有所帮助。

2.2 你真的需要安装任何东西吗?

2.2.1 一般的

Xillybus （PCIe 或 AXI）的 driver 是大多数 Linux kernels 和发行版的一部分，如下所述。然而，关于安装 udev 文件的段落 2.7 值得一看。

XillyUSB 的 driver 是 5.14版本中 Linux kernel 的一部分，该版本于 2021 年 8 月发布。

2.2.2 预装 driver的Linux 发行版

一些 Linux 发行版预装了 PCIe / AXI Xillybus driver (“out of the box”)，例如：

- Ubuntu 14.04 及更高版本
- 任何最近的 Fedora 发行版
- Xillinux （仅适用于 Zynq 和 Cyclone V SoC 平台）

快速检查 driver 是否安装在当前系统设置中是在 shell prompt处键入：

```
$ modinfo xillybus_core
```

如果安装了 driver ，则会打印有关它的信息。否则它会显示 “modinfo: ERROR: Module xillybus_core not found”。

同样，要检查 XillyUSB的 driver，命令是：

```
$ modinfo xillyusb
```

XillyUSB 无需任何安装即可与 Ubuntu 22.04、 Fedora 35 及更高版本以及派生发行版一起使用。

请注意，如果 Linux 在虚拟机中运行，它不会在 PCIe bus上检测到 Xillybus 。 driver 的操作系统必须在 bare metal上运行。 XillyUSB 可以在虚拟机中工作。

2.7 段落概述了如何永久更改 Xillybus device files 权限，因此任何用户都可以访问它们（与仅 root 用户相反——这在台式计算机上通常是需要的）。

2.2.3 Linux kernels 包括 Xillybus的 driver

Xillybus的 driver （用于 PCIe 和 AXI）从 3.12版本开始包含在官方 Linux kernel 中。在 3.12 和 3.17之间版本的 kernels 中， driver 被包含为 “staging driver”，这是社区完全接受任何新 driver之前的初步阶段。 Xillybus的 driver 在 3.18版本上被承认为 non-staging 。尽管有一些编码风格的变化，但 3.12 kernel 中最早的 driver 与目前可用的 driver 之间几乎没有功能差异。

当加载任何 staging driver 时， kernel 在系统日志中发出 warning ，表示其质量未知。关于 Xillybus，这个 warning 可以放心忽略。

至于 precompiled 并包含在发行版中的 kernels ： Xillybus 可能不包含在 kernel images 或 kernel modules中，因为某些发行版在 kernel 配置中启用了 Xillybus的 driver ，而另一些则没有。

那些自行执行 kernel compilation 的人可以将其配置为包含 driver，而不是将 driver 的单独 compilation 执行为 kernel modules （请参阅第 2.9段）。

但是对于想要坚持使用不支持 Xillybus 的发行版的用户， kernel module 选项通常更容易，如下所述。

如上所述， XillyUSB 的 driver 从版本 5.14开始包含在 Linux kernel 中。

2.3 检查先决条件

有时 Linux 安装缺少 kernel module compilation的基本工具。由于此任务很常见，因此可以在 Web 上找到特定于每个 Linux 发行版的操作指南。

了解这些工具是否存在的最简单方法是尝试运行它们。例如，在 command prompt上键入 “make coffee” 。这是正确的回应：

```
$ make coffee

make: *** No rule to make target 'coffee'.  Stop.
```

尽管这是一个错误，但我们可以看到 “make” 实用程序存在。如果需要安装 GNU make ，这就是我们所期望的：

```
$ make coffee
bash: make: command not found
```

C compiler 也是需要的。类型 “gcc”，具有所需的响应：

```
$ gcc
gcc: no input files
```

再次出现错误消息，但不是 “command not found”。

除了这两个工具之外，还需要安装 kernel headers 。这有点难以检查。知道这些文件丢失的常见方法是当 kernel compilation 失败并显示 header file 丢失的错误时。

在 Fedora、 RHEL、 CentOS 和其他 Red Hat 衍生产品上，此类命令可能会让计算机做好准备：

```
# yum install gcc make kernel-devel-$(uname -r)
```

在 Ubuntu 和其他基于 Debian的发行版上：

```
# apt install gcc make linux-headers-$(uname -r)
```

重要的:

这些安装命令必须作为 *root*发出。敦促那些不熟悉作为 *root* 用户这一概念的人首先了解它，其中包括附录的 *A.4*段落中的其他内容

.

2.4 解压下载的文件

从 Xillybus的站点下载 driver 后，将目录更改为下载文件所在的位置。在 command prompt上，键入（不包括 $ 符号）：

```
$ tar -xzf xillybus.tar.gz
```

对于 XillyUSB driver：

```
$ tar -xzf xillyusb.tar.gz
```

应该没有反应，只是一个新的 command prompt。

2.5 运行 kernel module的 compilation

将目录更改为模块源所在的位置。对于 XillyUSB driver：

```
$ cd xillyusb/driver
```

对于 Xillybus driver：

```
$ cd xillybus/module
```

键入 “make” 以运行模块的 compilation 。会话应如下所示：

```
$ make
make -C /lib/modules/3.10.0/build SUBDIRS=/home/myself/xillybus/module modules
make[1]: Entering directory `/usr/src/kernels/3.10.0'
  CC [M]  /home/myself/xillybus/module/xillybus_core.o
  CC [M]  /home/myself/xillybus/module/xillybus_pcie.o
```

```
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /home/myself/xillybus/module/xillybus_core.mod.o
  LD [M]  /home/myself/xillybus/module/xillybus_core.ko
  CC      /home/myself/xillybus/module/xillybus_pcie.mod.o
  LD [M]  /home/myself/xillybus/module/xillybus_pcie.ko
make[1]: Leaving directory `/usr/src/kernels/3.10.0'
```

细节可能略有不同，但不会出现错误或 warnings 。对于 XillyUSB，仅生成单个模块 xillyusb.ko。

请注意， kernel modules 的 compilation 特定于在 compilation期间运行的 kernel 。

如果打算使用另一个 kernel ，请键入 “make TARGET=kernel-version”，其中 “kernel-version” 是选择的 kernel 版本，如 /lib/modules/中所示。

2.6 安装 kernel 模块

留在同一目录中，成为 root 并键入 “make install”。这可能需要几秒钟，但不应产生任何错误。如果此操作失败，请将 compilation 生成的 *.ko 文件复制到某个现有的 kernel driver 子目录，然后按如下方式运行 depmod。该示例显示了如何复制 PCIe driver的文件，假设相关的 kernel 是 3.10.0：

```
# cp xillybus_core.ko /lib/modules/3.10.0/kernel/drivers/char/
# cp xillybus_pcie.ko /lib/modules/3.10.0/kernel/drivers/char/

# depmod -a
```

安装不会立即将模块加载到 kernel 中。如果发现 Xillybus 外围设备，这将在系统的下一个 boot 上完成。如何手动加载模块见 2.8段。

对于 XillyUSB，不需要 reboot ： 该模块会在下次插入 USB 设备或以其他方式发现时自动加载。

2.7 复制 udev rule 文件

默认情况下， Xillybus device files 只能由其所有者（即 root）访问。让任何用户都可以访问这些文件是很有意义的，这样就可以避免像 root 一样工作。 udev 机制在按照一组规则生成 device files 时更改这些文件权限。

因此，在同一个目录中，仍然是 root，将 udev rule 文件复制到保存所有规则的位置，很可能是 /etc/udev/rules.d/，使用

```
# cp 10-xillybus.rules /etc/udev/rules.d/
```

文件的内容很简单：

```
SUBSYSTEM=="xillybus", MODE="666", OPTIONS="last_rule"
```

这意味着任何由 Xillybus device driver 生成的文件都应该被赋予 permission mode 0666，即任何人都可以读写。

请注意，可以通过更改 udev 文件的内容来保留仅所有者权限并更改所有者的 ID 。

对于 XillyUSB，文件为 10-xillyusb.rules，包含

```
SUBSYSTEM=="xilly*", KERNEL=="xillyusb_*", MODE="0666"
```

2.8 加载和卸载模块

为了加载模块（并开始使用 Xillybus），输入 root：

```
# modprobe xillybus_pcie
```

或者，对于 XillyUSB：

```
# modprobe xillyusb
```

这将使 Xillybus device files 出现（假设 bus上存在相应的硬件）。

请注意，如果在系统执行其 boot 进程时存在 Xillybus PCIe / AXI 外围设备并且已经如上所述安装了 driver ，则不需要这样做。或者在 driver 已安装时将 XillyUSB 设备连接到计算机。

要查看 kernel中的模块列表，请键入 “lsmod”。要从 kernel中移除模块，请键入（对于 PCIe 机箱）

```
# rmmod xillybus_pcie xillybus_core
```

这将使 device files 消失。

如果出现问题，请检查 /var/log/syslog 日志文件中是否有包含单词 “xillybus” 或 “xillyusb”的消息（如果适用）。在这个日志文件中经常可以找到有价值的线索。使用 “dmesg” 命令也可以访问相同的日志信息。

如果不存在 /var/log/syslog 日志文件，则可能是 /var/log/messages 。

2.9 Linux kernel中的Xillybus PCIe driver

如前所述， Xillybus 的 driver 包含在 Linux kernel v3.12.0 及更高版本中。因此，作为单独安装 driver 的替代方案，可以运行整个 kernel 的 compilation ，并将其参数设置为支持 Xillybus。

包含 driver 的 kernel compilation 在功能上等同于段落 2.3 至 2.6中描述的步骤。

要将 Xillybus的 driver 包含在用于 compilation的 kernel 中，应启用其选项参数，作为 kernel modules 或 kernel 本身的一部分。比如 .config 中为 PCIe 接口启用 Xillybus的 driver 的部分，用 kernel module compilation，如下：

```
CONFIG_XILLYBUS=m
CONFIG_XILLYBUS_PCIE=m
```

同样，对于 XillyUSB （ kernel v5.14 及更高版本）：

```
CONFIG_XILLYUSB=m
```

更改此文件的常用方法是使用 kernel的构建环境工具，通常是 “make config”、 “make xconfig” 或 “make gconfig”。后两个是 GUI 工具，允许搜索字符串 “xillybus” 并通过单击复选框启用参数。上面显示的配置参数的文本表示可能有助于验证是否设置了正确的选项。

在 3.18之前的 kernels 上，可能需要在尝试启用 Xillybus之前启用 staging drivers ，从而导致 .config 文件中出现以下行。

```
CONFIG_STAGING=y
```

在 .config 文件中启用 Xillybus的 driver 后，按照惯例运行 kernel compilation 。

从 kernel 5.14开始，激活 Xillybus 和/或 XillyUSB 将自动导致 CONFIG_XILLYBUS_CLASS的启用。这是配置系统的依赖规则的结果。因此，手动设置此标志是不必要的（并且从用于配置 kernel的 GUI 实用程序中是不可能的）。

3

“Hello, world” 测试

3.1 目标

Xillybus 是一个开发套件。因此，最好在真正的 design中使用它来发现它的优点。因此，最初的 FPGA 代码包括最简单的可能实现，作为工作的基础。与 core的两对接口之间有两个 FIFOs 连接。发送到 core 的任何数据都存储在这些 FIFOs中，并循环回 host。 RAM 也连接到 core，演示对内存或 registers的访问。

下面显示的测试仅触发与 FPGA的通信。建议对 FPGA 稍作改动，并将自定义 logic 附加到 FIFOs，以便更好地了解系统的工作原理。

3.2 先决条件

- 从 demo bundle 获得的 bitstream 加载到 FPGA中。这在 Getting started with the FPGA demo bundle for Xilinx 或 Getting started with the FPGA demo bundle for Intel FPGA中有说明。

  对于 Zynq 和 Cyclone V SoC 平台，这是默认情况下的情况，因为 demo bundle 是 Xillinux的一部分。请参见 Getting started with Xillinux for Zynq-7000 或 Getting started with Xillinux for Cyclone V SoC (SoCKit)。

- Xillinux 平台除外：电脑正常执行 boot ， 检测到 PCIe / USB 接口（可以用 “lspci” 或 “lsusb”检查）。

- 您对 UNIX / Linux command-line 界面相当满意。附录 A 可能对此有所帮助。

3.3 琐碎的 loopback 测试

关于这个测试的想法是验证 loopback 确实有效。最简单的方法是使用 “cat”， 它是一

个 UNIX 命令行实用程序。

通过例如双击 “Terminal” 桌面项打开两个 terminal 窗口，或者如果桌面上不存在它，则在菜单中查找它。

在 command prompt 的第一个 terminal 窗口类型上（不要输入美元符号，它是 prompt）：

```
$ cat /dev/xillybus_read_8
```

这使得 “cat” 程序打印出它从 xillybus_read_8 device file读取的任何内容。在这个阶段预计不会发生任何事情。

那些使用 *XillyUSB* 的人会发现 *device file* 是 *xillyusb_00_read_8* 。 *“xillyusb”* 前缀很明显， *“00”* 索引意在允许多个 *USB* 设备连接到同一个 *host*。下面使用 *device files* 的 *PCIe / AXI* 命名约定。

在第二个 terminal 窗口中，键入

```
$ cat > /dev/xillybus_write_8
```

注意 > 重定向字符，它告诉 “cat” 将输入的任何内容发送到 xillybus_write_8。

现在在第二个 terminal上键入一些文本，然后按 ENTER。相同的文本将出现在第一个 terminal中。根据常见的 UNIX 约定，在按下 ENTER 之前，不会向 xillybus_write_8 发送任何内容。

如果在尝试这两个 “cat” 命令时收到错误消息，请检查拼写错误，除非错误是权限被拒绝。在后一种情况下，建议按照 2.7 中的步骤重新加载 driver 模块（或在计算机上执行 reboot ）。

或者，可以以 root 用户的身份与 Xillybus device files 进行交互，这在台式计算机上不太推荐（但在 embedded 平台上很常见）。附录 A.4中有关此内容的更多信息。

否则，请检查 “dmesg” 命令的输出中是否包含单词 “xillybus”的消息，并查看其中是否有任何指示问题。或者，从 /var/log/syslog （或 /var/log/messages，如果分布是 Red Hat、 Fedora、 CentOS 和类似分布之一）获取消息

任一 “cat” 都可以通过 CTRL-C停止。其他琐碎的文件操作也同样适用。例如，当第一个 terminal 仍在读取时，在第二个 terminal中键入以下内容：

```
$ date > /dev/xillybus_write_8
```

只要 FIFO 的两端都连接到 Xillybus的 IP core就可以了。 FPGA 代码中的更改当然会改变上述操作的结果。

请注意， FPGA 内部的 FIFOs 不会受到 overflow 和 underflow的威胁： core 尊重FPGA内部的 'full' 和 'empty' 信号。必要时， Xillybus driver 会强制应用程序等待，直到 FIFO 为 I/O 做好准备（通过 blocking = 强制 user space 程序休眠）。

还有另一对 device files ，它们之间有一个 FIFO ： /dev/xillybus_read_32 和 /dev/xillybus_write_32。这些 device files 以 32 位字粒度工作， FPGA中的 FIFO 也是如此。尝试与上述相同的测试将导致类似的行为，但有一个区别：所有 I/O 都以 4 字节块的形式执行，因此当输入未达到 4 字节块的边界时，最后一个字节将保持未传输状态。

4

host 应用示例

4.1 一般的

包含用于 Xillybus / XillyUSB的 host driver 的捆绑包中有四个或五个简单的 C 程序。demoapps 目录由以下文件组成：

- Makefile – 该文件包含 “make” 实用程序对程序的 compilation使用的规则。
- streamread.c – 从文件中读取，将数据发送到 standard output。
- streamwrite.c – 从 standard input读取数据，发送到文件。
- memread.c – 执行 seek后读取数据。演示如何访问 FPGA中的内存接口。
- memwrite.c – 执行 seek后写入数据。还演示了如何访问 FPGA中的内存接口。

这些程序的目的是展示正确的编码风格并作为编写自定义应用程序的基础。然而，它们都不适合在实际应用中使用，特别是因为它们不适合高带宽数据传输，如 5部分所述。

这些程序在这里没有单独记录，因为它们非常简单并且都符合 UNIX的常见文件访问实践，这在 Xillybus host application programming guide for Linux中有详细讨论。

请注意，这些程序使用普通函数 open()、 read()、 write() 等，而不是 fopen()、fread()、 fwrite() 集，因为后者可能会由于 C 库级别的 data buffers 而导致意外行为。

第五个程序 fifo.c仅与 driver for PCIe捆绑在一起。它演示了用于连续 data streaming的 userspace RAM FIFO 的实现。这个程序很少有用，因为 device file的 RAM buffers 可以配置为为几乎所有场景产生足够的空间。 XillyUSB永远无法达到这些场景，因为它提供的带宽相对较低。因此， fifo.c 不包含在其 driver中。

4.2 编辑和 compilation

熟悉 Linux 环境下 compilation 程序的朋友可以直接跳到下一段： 程序设置没有什么异常。

首先，将目录更改为 C 文件所在的位置：

```
$ cd demoapps
```

要运行所有五个程序的 compilation ，只需在 shell prompt处键入 “make” 。预计将举行以下会议：

```
$ make
gcc  -g -Wall -O3 memwrite.c -o memwrite
gcc  -g -Wall -O3 memread.c -o memread
gcc  -g -Wall -O3 streamread.c -o streamread
gcc  -g -Wall -O3 streamwrite.c -o streamwrite
gcc  -g -Wall -O3 -pthread fifo.c -o fifo
```

以 'gcc' 开头的五行是 compiler的调用，由 “make” 实用程序生成。这些命令可以单独用于任何程序的 compilation （但没有理由这样做。只需使用 “make”） 。

在某些系统上，如果未安装 POSIX threads 的库（例如在某些 Cygwin 安装中） ，第五个 compilation （ fifo.c的）可能会失败。这可以忽略，除非 fifo.c 是感兴趣的程序。

“make” 实用程序仅在必要时运行 compilation 。如果仅更改了一个源文件，则在随后的 “make”调用中，只有该文件将经历 compilation 。因此，正常的程序是编辑感兴趣的源文件，然后根据需要将 “make” 用于 recompilation 。

要删除由 compilation生成的 executables ，请键入 “make clean”。

如上所述， compilation 规则在 Makefile中。它的语法可能被认为有些困难，但幸运的是，它是那些可以在不完全理解它们的情况下进行编辑的文件之一。

给定的 Makefile 仅与当前目录中的文件相关。这意味着可以制作整个目录的副本，并在没有冲突的情况下处理副本。除了其他源文件之外，还可以添加 C 文件并轻松更改规则，以便 “make” 在其上运行 compilation 。

例如，假设 memwrite.c 被复制到一个新文件 mygames.c。这可以通过 GUI 接口或 command line来完成：

```
$ cp memwrite.c mygames.c
```

现在来编辑 Makefile。有许多编辑器和许多方法来调用它们中的每一个。对于快速入门，请使用 gedit 或 xed，它们可以通过桌面上的图标调用，也可以直接在 shell prompt上调用。要编辑 Makefile，只需去

```
$ gedit Makefile &
```

命令末尾的'&'表示不要等到应用程序完成再获取另一个 shell prompt，它适用于启动 GUI 应用程序等。

在 Makefile中，有一句话说

```
APPLICATIONS=memwrite memread streamread streamwrite
```

它实际上只是一个名称列表，它们之间有空格。将 mygames（不带 .c 后缀）添加到列表中。

4.3 执行

段落 3.3 中显示的简单 loopback 示例可以通过两个应用程序完成。在 compilation之后，输入第一个 terminal：

```
$ ./streamread /dev/xillybus_read_8
```

这是从 device file读取的程序。注意选择 executable时需要用 "./" 明确指向当前目录。

然后，在第二个窗口中（假设需要更改目录）：

```
$ cd demoapps
$ ./streamwrite /dev/xillybus_write_8
```

这或多或少类似于 "cat"，只是 "streamwrite" 尝试逐个字符地工作，而不是等待 ENTER。这是在一个名为 config_console()的函数中完成的，可以忽略它：它只是为了打字的直接效果而存在的。

上面的示例与 PCIe / AXI的 Xillybus 有关。对于 XillyUSB， device file 名称前缀略有不同，因此它是例如 xillyusb_00_read_8 而不是 xillybus_read_8。

重要的:

streamread 和 *streamwrite* 以 *128* 字节的块执行它们的 *I/O* ，以保持实现简单。当数据速率很重要时，应使用更大的 *buffers* 。见第 *5.3*段。

4.4 内存接口

memread 和 memwrite 程序更有趣，因为它们演示了如何通过在 device file上调用 lseek() 函数来访问 FPGA 上的内存。请注意，在 demo bundle中，只有 xillybus_mem_8 允许 seeking。它也是唯一可以读写的 device file 。

在写入内存之前，使用 hexdump 实用程序观察当前情况：

```
$ hexdump -C -v -n 32 /dev/xillybus_mem_8
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020
```

输出可能会有所不同：它反映了 FPGA的 RAM ，它可能包含其他值（很可能是因为之前的实验）。

-C 和 -v 标志告诉 hexdump 使用所示的输出格式。“-n 32” 部分只要求前 32 个字节。内存数组只有 32 个字节长，所以没有必要再读下去了。

简要说明发生了什么： hexdump 打开 /dev/xillybus_mem_8 并从中读取 32 个字节。每个允许 lseek() 函数调用的文件在打开时默认从位置 0 开始，因此输出是内存数组中的前 32 个字节。

将地址 3 的内存值更改为 170 （ hex 格式的0xaa ）：

```
$ ./memwrite /dev/xillybus_mem_8 3 170
```

并再次读取整个数组：

```
$ hexdump -C -v -n 32 /dev/xillybus_mem_8
00000000  00 00 00 aa 00 00 00 00  00 00 00 00 00 00 00 00  |...Ãª............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020
```

很明显，它奏效了。

memwrite.c 中的特殊部分是 “lseek(fd, address, SEEK_SET)”。使用此命令，设置FPGA的地址，导致任何后续读取或写入从该位置开始（并随着数据流而增加）。

5

高带宽性能指南

Xillybus的 IP cores 的用户经常进行数据带宽测试，以确保确实满足宣传的数据速率。实现这些速率需要避免可能会显着减慢数据流的障碍，其中大部分是由于 host 处理数据速度不够快造成的。

更重要的是在应用项目中避开这些障碍，以充分利用 IP core的全部功能。

本节是基于最常见错误的指南集合。遵循这些指南应导致带宽测量值等于或略高于公布的值。

到目前为止，抱怨未达到预期带宽的最常见原因是测量不正确。推荐的方法是使用Linux的 “dd” 命令，如下面的 5.3 段所示。

本节中的信息对于 “Getting Started” 指南来说是相对高级的，并且参考了其他文档中解释的主题。尽管如此，在本指南中还是给出了它，因为许多用户在熟悉 IP core的早期阶段就进行了性能测试。

5.1 不要 loopback

demo bundle 中的 logic 实现了一对 streams 之间的数据的 loopback 方向相反。虽然这对初始测试很有帮助，但对于测试性能来说却是一个糟糕的设置。问题是由Xillybus IP core产生的数据传输突发，要么完全填满 loopback FIFO ，要么完全清空。每次发生这种情况时，都会导致 IP core的数据传输短暂停止。这些暂停大大降低了测量的吞吐量。

在实际场景中，使用从 FPGA 到 host的最大数据速率， application logic 填充FPGA的 FIFO 的速度与 Xillybus IP core 消耗它的速度一样快。因此， FIFO 永远不会是空的。同样在另一个方向上， application logic 清空 FIFO 的速度与 IP core 填充它的速度一样快，因此它永远不会装满。

当然，从功能的角度来看， FIFO 在第一种情况下是空的，在第二种情况下是满的，这很好。这只会导致 Xillybus stream 暂时停止。一切正常，只是没有以最大速度运行。

如果需要基于 demo bundle进行性能测试，可以很容易地对其进行修改。例如，为了测试 32 位接口，将 user_r_read_32_empty 和 user_w_write_32_full 信号从 FIFO (fifo_32) 断开，并将它们绑定到常数零。这很可能会导致 FIFO 同时受到 overflow 和 underflow 的影响，从而导致错误的数据，但数据速率将是最佳的。如果希望在此类性能测试中使用应用程序数据，则确保这两个信号永远不会变高的任何安排都可以。

请注意，打开 loopback 可以单独测试每个方向，但这也是同时测试两个方向的正确方法。

5.2 不涉及磁盘或其他存储

磁盘、 solid-state drives 和其他类型的计算机存储通常是无法满足带宽预期的原因。操作系统的 caching 机制增加了混乱，因为它允许短距离突发的快速数据传输，而不涉及底层物理媒体。由于 cache 在现代计算机上可能非常大，因此在磁盘的速度限制变得可见之前，可以传输多个 Gigabytes 数据。这可能会导致用户认为 Xillybus的数据传输出现问题，因为数据速率的这种突然变化没有其他明显的解释。

对于 solid-state drives (flash)，还有一个额外的混淆因素，尤其是在长时间连续写入期间：写入 flash drive 涉及擦除未使用的 blocks。这是因为向 flash memory 写入数据只允许对已擦除的 blocks 进行，即为空。

flash drive 通常有一个 blocks 池已被擦除。这使得写入操作更快，因为有可用空间来写入数据。然而，当没有更多空的 blocks时，速度会显着下降，因为 flash drive 被迫擦除 blocks 并可能重新组织其数据。

由于这些原因，测试 Xillybus的带宽绝不应该涉及任何存储介质，即使快速检查会使它看起来好像介质足够快。一个常见的错误是尝试将数据从 Xillybus stream 复制到磁盘上的一个大文件中，并测量它所花费的时间。即使这个操作在功能上是正确的，性能测量也可能完全错误。

如果存储打算成为应用程序的一部分（例如 data acquisition），建议对介质运行广泛的长期测试，以验证它是否符合预期。短的 benchmark test 可能极具误导性。

5.3 读写大块

每个 read() 和 write() 操作都会在操作系统上生成一个 system call ，这会在 CPU 周期中产生影响。因此，使用足够大的 buffer 尺寸非常重要。这适用于带宽测试以及高性能应用程序。

对于每个 read() 和 write() 函数调用，一个常见的 buffer 大小约为 128 kB 。这并不一定意味着每个这样的函数调用都处理 128 kB，而是这些函数调用最多允许这么多。

重要的是要注意 streamread 和 streamwrite 示例实用程序（请参阅 4部分）不适合测量性能，因为其中的 buffer 大小是 128 字节（不是 kB）。这简化了示例代码，但对于性能测试来说太慢了。

以下 shell 命令可用于快速检查速度（根据需要更换 /dev/xillybus_* names ）：

```
dd if=/dev/zero of=/dev/xillybus_sink bs=128k
dd if=/dev/xillybus_source of=/dev/null bs=128k
```

这些一直运行，直到用 CTRL-C停止。添加 “count=” 参数以针对固定数量的数据运行测试。

5.4 注意 CPU 的消耗

一个常见的错误是高估 CPU在速度方面的能力。与普遍的看法不同，即使是最快的 CPUs 也很难做任何有意义的事情，因为数据比 100-200 MB/s 更快（每个 thread）。计算机程序通常是密集型应用程序的瓶颈，不一定是数据传输。有时， buffer 大小不足（如上所述）也会导致 CPU 消耗过多。

因此，密切关注 CPU 的消耗很重要，例如使用 “top”。尽管如此，确保正确解释这些程序的输出仍然很重要，尤其是在多core 机器上。例如， quad-core 计算机上的 25% CPU 是否意味着低 CPU 消耗，还是特定 thread上的 100%？ 如果使用 “top” ，那取决于程序的版本。

另外需要注意的是， system calls的处理时间是如何测量和显示的：如果操作系统的 overhead 减慢速度，它会出现在给定进程的 CPU 百分比中吗?

一个简单的判断方法是使用 “time” 实用程序。例如，

```
$ time dd if=/dev/zero of=/dev/null bs=128k count=100k
102400+0 records in
102400+0 records out
13421772800 bytes (13 GB) copied, 1.07802 s, 12.5 GB/s

real 0m1.080s
user 0m0.005s
sys 0m1.074s
```

底部 “time” 的输出表明，在此操作所用的 1.080 秒中， processor 在 user space 程序中花费了 5 ms ， 而处理系统调用的时间为 1.074 秒。总结一下，很明显 processor

一直很忙，所以 processor 是瓶颈。这是完全可以预料的，因为在这个例子中没有执行真正的 I/O 。

5.5 不要让读写相互依赖

对于需要双向通信的应用程序，一个常见的错误是编写一个由一个主循环组成的 single-threaded 计算机程序。对于每个循环，向 FPGA写入一个数据块，然后从中读取一个数据块。

如果两个 streams 在功能上是独立的，这可能没问题。然而很多时候这样的程序是为 coprocessing 应用程序编写的，基于这样的误解，即程序应该发送一个块进行处理，然后读回结果，因此每个循环完成一定数量的数据处理。

这种方法不仅效率低下，还可能导致执行卡住（取决于编写的错误程度）。Xillybus host application programming guide for Linux 的第 6.6 节详细阐述了这个主题，并提出了一种更合适的编码技术。

5.6 了解 host的 RAM bandwidth 极限

这主要适用于 embedded 系统和/或使用修订版 XL / XXL IP core时：每个主板（或 embedded 系统）对其 DDR RAM 内存单元的带宽有限。在要求非常苛刻的应用程序中，这可能会成为瓶颈。

请记住，从 FPGA 到 user space 程序的每个数据块都需要两个 RAM 操作：第一个是当数据从 FPGA 发送到 DMA buffer时。第二种是将数据复制到 user space 程序可访问的 buffer 中时。出于类似的原因，当数据以相反方向传输时，也需要两次 RAM 操作。

DMA buffers 和 user space buffers 之间的分离是所有使用 read() 和 write() （或类似函数调用）的 I/O 的操作系统要求。

因此，如果同时在两个方向上测试修订版 XL IP core ，预计每个方向上大约 3.5 GB/s ，这需要 RAM的四倍带宽，即 14 GB/s。并非所有主板都具有此功能，还要记住 host 会同时将 RAM 用于其他事情。

使用 XXL修订版，出于同样的原因，即使是一个方向的简单测试也可能超过 RAM的带宽能力。

5.7 DMA buffers 足够大

这很少是一个问题，但仍然值得一提：如果在 host 上为 DMA buffers 分配的 RAM 空间太小，可能会减慢数据传输，因为 host 被迫将 data stream 分成小块，因此浪费

CPU 循环。

所有 demo bundles 都有足够的 DMA 内存来进行性能测试，同样适用于在 IP Core Factory 生成的 cores ，将所需的 “Expected BW” 设置为实际预期并启用 “Autoset Internals” 。 “Buffering” 应该设置为 10 ms，即使任何选项都可能很好。

一般来说，四个 DMA buffers，总的 RAM 空间对应于 10 ms 的数据在请求的速率，对于带宽测试的目的是足够的。

5.8 使用正确的数据宽度

很明显，任何 stream 的最大数据速率都受到 stream的数据宽度和 bus_clk的频率的限制。但除此之外，32 位 streams 应该在 A版本的 IP cores 上使用，因为 8 位和 16 位 streams 消耗的 PCIe 带宽比它们实际用于有效载荷数据传输的带宽要多。

B 及更高版本的IP cores 允许选择比 PCIe 模块的数据路径更宽的数据宽度。例如，测试 B版本的 IP core 自然需要 64 位宽的 stream，因为 PCIe 数据路径是 64 位宽。选择更宽的 stream 可能会产生更好的结果，因为 application logic 可以更快地访问数据。然而，差异可能可以忽略不计。

版本 XL 的IP cores 应使用 128 位 data streams （或 256 位，应该不会产生显着差异）进行测试。对于 XXL，这些 IP cores 应使用 256 位 data streams进行测试。

5.9 cache 同步导致减速

这仅适用于 embedded 系统，因为 x86派生的 processors （32 位和 64 位）使用 coherent cache。此问题也不适用于使用 Zynq processor 的 AXI bus （例如使用 Xillinux）的 Xillybus ，因为它连接到相干端口（ACP），但当 Zynq processor 使用 Xillybus 而不是 PCIe bus时，它确实适用。

在多个 embedded processors上，特别是在 ARM processor上使用 PCIe bus 时，访问 DMA buffers 时需要同步 cache 会大大减慢传输速度。这不是 Xillybus 问题，因为它适用于 processor上任何基于 DMA的 I/O ，例如 Ethernet、 USB 端口和其他可能的 PCIe 设备。

尽管使用了大型 buffers （如上文第 5.3 段所述），但在 system call 状态（ “time” 实用程序的“sys” 行输出）下不合理的 CPU 消耗发现了由于 cache 导致的减速。这是在调用 processor的 cache同步操作码时浪费了 CPU 周期的结果。

x86/x64 架构上不存在此问题，因为它们具有不需要同步的 coherent cache 。

5.10 参数调整

demo bundles 中可供下载的 PCIe 模块经过调整，可以处理基于 x86的主流 processor上的请求带宽。同样，在 IP Core Factory 和 “Autoset Internals” 中生成的 streams 通常在性能和 FPGA 资源利用率之间提供最佳平衡，并确保为每个 stream定义的带宽。

在极少数情况下，并且仅针对 B 和更高版本的 cores ，可能需要稍微进一步调整 PCIe 模块的参数以达到所要求的数据速率，特别是在从 host 到 FPGA的 streams 上。这在 The guide to defining a custom Xillybus IP core的 4.5 节中讨论。

但是请注意，即使在这种特殊情况下，修改的不是 Xillybus IP core的参数，而是 PCIe 模块的参数。尝试通过调整 IP core的参数来提高数据速率是一个常见的错误，但问题几乎总是出在上述问题之一。

6

故障排除

用于 Xillybus / XillyUSB 的 drivers 旨在生产有意义的 log messages。因此，建议在出现问题时在 “dmesg” 命令的输出中查找包含单词 “xillybus” 的消息。在 /var/log/syslog（或某些系统中的 /var/log/messages ）中可以找到相同的 log messsages

事实上，即使一切看起来都很好，也建议密切关注 system log 。

来自 PCIe / AXI driver 的消息列表及其解释可在以下位置找到

http://xillybus.com/doc/list-of-kernel-messages

但是，通过在消息文本本身上使用 Google 可能更容易找到特定消息。

A

Linux command line的简短生存指南

不习惯 command line 界面的人可能会觉得在 Linux 机器上做事有些困难。由于基本的命令界面已经 30 多年了，所以网上有很多关于如何使用每个命令的教程。这个简短的指南只是一个入门。

A.1 一些击键

这是最常用的击键的摘要。

- CTRL-C：停止当前运行的应用程序
- CTRL-D：完成此会话（关闭 terminal 窗口）
- CTRL-L：清屏
- TAB ：在 command prompt，尝试自动完成当前写入的项目。对长文件名很有用：键入名称的开头，然后键入 [TAB]。
- 向上和向下箭头：在命令 prompt处，建议来自命令历史记录的先前命令。用于重复刚刚完成的事情。也可以编辑以前的命令，因此它也可以与以前的命令几乎相同。
- space：当电脑用 terminal pager显示东西时， [space] 就是 “page down”。
- q： “Quit”。在逐页显示中， “q” 用于退出该模式。

A.2 获得帮助

没有人真正记得所有的标志和选项。有两种常用方法可以获得更多帮助。一是 “man” 命令，二是帮助标志。

例如，要了解有关 “ls” 命令的更多信息（列出当前目录中的文件）：

```
$ man ls
```

请注意，“$”符号是命令 prompt，计算机打印出来表示它已准备好执行命令。在大多数情况下， prompt 更长，并且包含一些关于用户和当前目录的信息。

手册页与 terminal pager一起显示。使用 [space]、箭头键、 Page Up 和 Page Down 导航， ’q’ 退出。

要获得更短的命令摘要，请运行带有 `--help` 标志的命令。一些命令响应 `-h` 或 `-help` （带有一个破折号）。其他人会在语法不正确时打印帮助信息。这是试验和错误。对于 ls 命令：

```
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
```

...

（它继续）

A.3 显示和编辑文件

如果文件预计很短（或者如果可以使用 terminal window的滚动条），可以通过以下方式在 console 上显示其内容：

```
$ cat filename
```

较长的文件需要 terminal pager：

```
$ less filename
```

至于编辑文本文件，有很多编辑器可供选择。最流行的（但不一定是最容易开始的）是 emacs （和 Xemacs）以及 vi。众所周知， vi 编辑器很难学习，但由于其简单性，它始终可用且始终有效。

推荐的简单 GUI 编辑器是 gedit或 xed，以可用者为准。它可以通过桌面菜单或从 command line启动：

```
$ gedit filename &
```

末尾的'&'表示该命令应该执行 "in the background"。或者简单地说，下一个 command prompt 在命令完成之前出现。 GUI 应用程序最好这样开始。

当然，这些示例中的 'filename' 也可以是一条路径。例如查看系统主 log file：

```
# less /var/log/syslog
```

要跳到文件末尾，请在 "less" 运行时按 shift-G。

请注意，日志文件只能由某些计算机上的 root 用户访问。

A.4 root 用户

所有 UNIX 机器，包括 Linux ，都有一个用户为 ID 0 ，名称为 "root"，也称为 superuser。这个用户的特别之处在于它允许一切。访问文件、资源和某些操作的常见限制是根据您是哪个用户实施的，不会强加给 root 用户。

这不仅仅是多用户计算机上的隐私问题。被允许的一切包括在 shell prompt上用简单的命令删除硬盘中的所有数据。它包括其他几种错误删除数据、使计算机一般无用或使系统容易受到攻击的方法。任何 UNIX 系统的基本假设是，拥有 root 访问权限的人都知道自己在做什么。计算机不会询问 root are-you-sure 问题。

作为 root 工作是系统维护（包括软件安装）所必需的。不把事情搞砸的关键是在按下 ENTER之前思考，并确保命令完全按照要求输入。完全按照安装说明进行操作通常是安全的。不要在不了解他们到底在做什么的情况下进行任何修改。如果可以以非 root 的用户身份重复相同的命令（可能涉及其他文件），请尝试以该用户身份执行此操作。

由于作为 root的危险，作为 root 运行命令的常用方法是使用 sudo 命令。例如查看主日志文件：

```
$ sudo less /var/log/syslog
```

这并不总是有效，因为系统需要设置为允许特定用户使用这种特殊模式。系统需要用户密码（不是 root 密码）。

第二种方法是键入 "su" 并启动一个会话，其中每个命令都以 root给出。这在 root需要完成几个任务时很方便，但也意味着有更大的机会忘记自己是 root，想都不想就写错了。保持 root 会话简短。

```
$ su
Password:
# less /var/log/syslog
```

这次需要 root 密码。

shell prompt 的变化表明身份从普通用户更改为 root。如有疑问，请键入 “whoami” 以获取您当前的 user name。

在某些系统上，sudo 适用于相关用户，但仍可能希望将会话作为 root调用。如果 “su” 不能使用（主要是因为 root 密码未知），简单的替代方法是

```
$ sudo su
#
```

A.5 选定的命令

最后，这里有几个常用的 UNIX 命令。

一些文件操作命令（最好使用 GUI 工具）：

- cp ——复制文件或文件。
- rm –删除一个或多个文件。
- mv ——移动文件或文件。
- rmdir ——删除目录。

以及一些建议一般了解的内容：

- ls – 列出当前目录中的所有文件（或另一个，如果指定）。 “ls -l” 列出了它们及其属性。
- lspci - 列出 bus上的所有 PCI （和 PCIe）设备。用于判断 Xillybus 是否已被检测为 PCIe 外围设备。也可以试试 lspci -v、 lspci -vv 和 lspci -n。
- lsusb – 列出 bus上的所有 USB 设备。用于判断 XillyUSB 是否已被检测为外围设备。也可以试试 lsusb -v 和 lsusb -vv。
- cd –更改目录
- pwd –显示当前目录

- cat ——将文件（或多个文件）发送到 standard output。或者如果没有给出参数，则使用 standard input 。这个命令的最初目的是连接文件，但它最终成为了文件的简单输入和输出的瑞士刀。
- man – 在某个命令上显示 manual page 。也可以试试 “man -a” （有时一个命令有多个 manual page ）。
- less —— terminal pager。逐页显示来自 standard input 的文件或数据。看上面。也用于显示命令的长输出。例如，

```
$ ls -l | less
```

- head –显示文件的开头
- tail ——显示文件的结尾。或者更好的是，使用 -f 标志：在它们到达时显示结尾 + 新行。适用于日志文件，例如（如 root）：

```
# tail -f /var/log/syslog
```

- diff ——比较两个文本文件。如果它什么也没说，文件是相同的。
- cmp ——比较两个二进制文件。如果它什么也没说，文件是相同的。
- hexdump ——以简洁的格式显示文件的内容。首选标志 -v 和 -C 。
- df –显示已安装的磁盘以及每个磁盘中剩余的空间。更好的是， “df -h”
- make – 尝试按照 Makefile中的规则构建（运行 compilation ）项目
- gcc —— GNU C compiler。
- ps –获取正在运行的进程列表。 “ps a”、 “ps au” 和 “ps aux” 将提供不同数量的信息。通常太多了。

还有一些高级命令：

- grep – 在文件中搜索 pattern 或 standard input。该模式是 regular expression，但如果它只是文本，那么它会搜索字符串。例如，在主日志文件中搜索 “xillybus”作为 case insensitive 字符串，并逐页显示输出：

```
# grep -i xillybus /var/log/syslog | less
```

- find ——查找文件。它具有复杂的参数语法，但它可以根据文件的名称、年龄、类型或您能想到的任何内容来查找文件。请参阅 man page。