# Getting started with Xillybus on a Linux host

*Xillybus Ltd.*

www.xillybus.com

*Version 4.0*

# 1

# Introduction

This guide goes through the steps for installing the driver for the purpose of running Xillybus / XillyUSB on a Linux host. How to try out the basic functionality of the IP core is also shown.

For the sake of simplicity, it is assumed that the host is a fully capable computer, which is able to perform compilation. The procedure for an embedded platform is similar, with a few straightforward differences (in particular, cross compilation may be necessary).

This guide also assumes that a bitstream which is based upon Xillybus' demo bundle has already been loaded into the FPGA, and that the FPGA has been recognized as a peripheral by the host (through PCI Express, the AXI bus, or USB 3.x, as relevant).

The steps for reaching this stage are outlined in one of these documents (depending on the chosen 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)

The host driver generates device files which behave like named pipes: These device files are opened, read from and written to just like any file. However, these files behave in a similar way to pipes between processes. This behavior is also similar to TCP/IP streams. To the program running on the host, the difference is that the other side of the stream is not another process (on the same computer or on a different computer on the network) but instead, the other side is a FIFO inside the FPGA. Just

like a TCP/IP stream, the Xillybus stream is designed to work efficiently with high-rate data transfers, but the stream also performs well when small amounts of data are transmitted occasionally.

One single driver on the host is used with all Xillybus IP cores that communicate with the host through PCIe. A different driver is intended for the AXI interface. There is also a different driver for XillyUSB.

There is no need to change the driver when a different IP core is used in the FPGA: The streams and their attributes are automatically detected by the driver as the driver is loaded into the host's operating system. The device files are created accordingly, with file names of the format /dev/xillybus_*something*. Likewise, the driver for XillyUSB creates device files with the format /dev/xillyusb_00_*something*. In these file names, the 00 part is the index of the device. This part is replaced with 01, 02 etc. when more than one XillyUSB device is connected to the computer at the same time.

More in-depth information on topics related to the host can be found in Xillybus host application programming guide for Linux.

# 2

# Installing the host driver

## 2.1 Stages for installing Xillybus' driver

Installing the Linux kernel driver consists of the following steps:

- Checking if an installation is needed at all. If not, skip the other steps below (possibly not skipping the last step, i.e. copying the udev file)
- Checking for prerequisites (i.e. checking that the compiler and kernel headers are installed)
- Uncompressing the downloaded file which contains the driver as a kernel module.
- Running a compilation of the kernel module
- Installing the kernel module
- Installing the udev file, so that Xillybus device files become accessible to any user (and not only root).

These steps are carried out using command-line ("Terminal"). The short Linux survival guide in Appendix A may be helpful for those who are less experienced with this interface.

## 2.2 Do you really need to install anything?

### 2.2.1 General

The majority of Linux kernels and Linux distributions support Xillybus (for PCIe or AXI) without a need to do anything. This is explained further below.

That said, it's worth looking at section 2.7 regarding the installation of a udev file, even if Xillybus is already supported.

The driver for XillyUSB is part of the Linux kernel from version 5.14 (released in August 2021).

### 2.2.2 Linux distributions with pre-installed driver

Most Linux distributions have the PCIe / AXI Xillybus driver already installed ("out of the box"). For example:

- Ubuntu 14.04 and later
- Any fairly recent Fedora distribution
- Xillinux (for the Zynq and Cyclone V SoC platforms only)

In order to quickly check if the driver is installed, type the following at shell prompt:

```
$ modinfo xillybus_core
```

If the driver is installed, information about it is printed. Otherwise it says "modinfo: ERROR: Module xillybus_core not found".

Likewise, to check for XillyUSB's driver, the command is:

```
$ modinfo xillyusb
```

XillyUSB works without any need for installation with Ubuntu 22.04 and later, Fedora 35 and later, as well as distributions that are derived from these two.

Note that if Linux runs inside a virtual machine, it will not detect Xillybus on the PCIe bus. The operating system with the driver must run on bare metal. XillyUSB may work inside a virtual machine.

Section 2.7 shows how to permanently change the permissions of the Xillybus device files. This change makes these files accessible to any user (and not only to the root user). This modification is often desired when using Xillybus on a desktop computer.

### 2.2.3 Linux kernels that include Xillybus' driver

Xillybus' driver (for PCIe and AXI) is included in the official Linux kernel starting from version 3.12. In kernels with versions between 3.12 and 3.17, the driver was included as "staging driver", which is a preliminary stage before Linux' community fully accepts a new driver. Xillybus' driver was admitted as non-staging in version 3.18. Despite several changes that are related to coding style, there are almost no functional differences between the earliest driver (in version 3.12 of the kernel) and the driver that is available today.

When a staging driver is loaded, the kernel issues a warning in the system log. This warning says that the driver's quality is unknown. Regarding Xillybus, this warning can be ignored safely.

As mentioned above, the driver for XillyUSB is included in the Linux kernel starting from version 5.14.

Regarding kernels that are a part of a Linux distribution: Even when Xillybus' drivers are part of the kernel's source code, these drivers are included in the compilation only if the kernel is configured to include these drivers. Xillybus' drivers are included as kernel modules in most mainstream Linux distributions, but each distribution has its own criteria for choosing what to include in the kernel. Accordingly, Xillybus may not be included in the kernel that arrives along with a distribution.

This guide focuses on installing the drivers by virtue of a separate compilation of the kernel modules. This is usually the easiest way. However, those who carry out a compilation of their kernel anyhow, may instead prefer to configure the kernel to include Xillybus' drivers. This method is discussed in section 2.9.

## 2.3 Checking for prerequisites

A Linux system may lack the basic tools for a kernel module compilation. The simplest way to know if these tools exist is to attempt running them. For example, type "make coffee" on command prompt. This is the correct response:

```
$ make coffee

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

Even though this is an error, we can see that the "make" utility exists. But if GNU make is missing and needs to be installed, the output will be something like this:

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

The C compiler is needed as well. Type “gcc” in order to check if the compiler is installed:

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

This response indicates that “gcc” is installed. There was an error message again, but not “command not found”.

On top of these two tools, the kernel headers need to be installed as well. This is a bit more difficult to check up. The common way to know if these files are missing is when the kernel compilation fails with an error saying that a header file is missing.

A kernel module compilation is a common task, so there is a lot of information on the Internet that is specific to each Linux distribution regarding how to prepare the system for the compilation.

On Fedora, RHEL, CentOS and other derivatives of Red Hat, a command of this sort is likely to get the computer prepared:

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

For Ubuntu and other distributions that are based upon Debian:

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

**IMPORTANT:**

*These installation commands must be issued as root. Those who are not familiar with the concept of being the root user are urged to learn about it first. See the Appendix’ section A.4.*

## 2.4 Uncompressing the downloaded file

After downloading the driver from Xillybus’ site, change directory to where the downloaded file is. At command prompt, type (excluding the $ sign):

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

For the XillyUSB driver:

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

There should be no response, just a new command prompt.

## 2.5 Running the compilation of the kernel module

Change directory to where the source code of the kernel module is. For the Xillybus driver:

```
$ cd xillybus/module
```

And for the XillyUSB driver:

```
$ cd xillyusb/driver
```

Type “make” to carry out the compilation of the modules. The transcript should be something like this:

```
$ 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'
```

The details may vary slightly, but no errors or warnings should appear. For XillyUSB, only a single module, xillyusb.ko, is generated.

Note that the compilation of the kernel modules is specific to the kernel that is running during the compilation.

If another kernel is intended for use, type “make TARGET=kernel-version”, where “kernel-version” is the name of the required version of the kernel. This is the name that appears in /lib/modules/. Alternatively, edit the following line in the file with the name “Makefile”:

```
KDIR := /lib/modules/$(TARGET)/build
```

Change the value of KDIR to the path of the required kernel headers.

## 2.6 Installing the kernel module

Without changing directory, change the user to root (e.g. with “sudo su”). Then type the following command:

```
# make install
```

This command can take a few seconds to complete, but shouldn't generate any errors.

If this fails, copy the *.ko files that were generated by the compilation to an existing subdirectory for kernel modules. Then run depmod. The following example shows how to do this for the PCIe driver, if the relevant version of the kernel is 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
```

The installation does not load the module into the kernel immediately. This is done on the next boot of the system if a Xillybus peripheral is discovered. How to load the module manually is shown in section 2.8.

For XillyUSB, a reboot is not necessary: The module is loaded automatically the next time the USB device is connected to the computer.

## 2.7 Copying the udev rule file

By default, Xillybus device files are accessible only by their owner, which is root. It makes a lot of sense to make these files accessible to any user, so that working as root can be avoided. The udev mechanism changes the file permissions when the device files are generated, by obeying specific rules.

How to enable this feature: Remain in the same directory, and remain being the root user. Copy the udev rule file to where such files are stored in your system (most likely /etc/udev/rules.d/).

For example:

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

The content of this file is simply:

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

This means that all files that are generated by the Xillybus device driver should be given permission mode 0666. In other words, reading and writing is allowed to everyone.

For XillyUSB, the file is 10-xillyusb.rules, containing

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

Note that the udev file can be changed to achieve different results. For example, it's possible to change the device files' owner instead, so only a specific user has access to these files.

## 2.8 Loading and unloading the module

In order to load the module (and start working with Xillybus), type as root:

```
# modprobe xillybus_pcie
```

Or, for XillyUSB:

```
# modprobe xillyusb
```

This will make the Xillybus device files appear (assuming that a Xillybus device is detected on the bus).

Note that this should not be necessary if a Xillybus PCIe / AXI peripheral was detected when the system carried out its boot process and the driver was already installed as described above. Neither is this necessary if a XillyUSB device is connected to the computer when the driver is already installed.

To see a list of modules in the kernel, type "lsmod". To remove the module from the kernel, type (for the PCIe driver)

```
# rmmod xillybus_pcie xillybus_core
```

This will make the device files vanish.

If something seems to have gone wrong, please check up the /var/log/syslog file for messages containing the word “xillybus” or “xillyusb”, as applicable. Valuable clues are often found in this log file. The same log information is also accessible with the “dmesg” command.

If no /var/log/syslog log file exists, it’s probably /var/log/messages instead. Possibly try the command “journalctl -k”.

## 2.9 Xillybus drivers in the official Linux kernel

As mentioned before, the driver for Xillybus is included in the Linux kernel, starting from version v3.12.0. It’s therefore possible to carry out a compilation of the entire kernel, so that this kernel supports Xillybus. This is an alternative to installing the kernel modules separately, as shown above.

From a functional point of view, the method that involves kernel compilation yields the same result as the steps described in sections 2.3 to 2.6.

In order to include Xillybus’ driver in a kernel that is intended for compilation, it is necessary to enable a few kernel configuration options. There are two ways to include the driver: Either as kernel modules or as part of the kernel image.

For example, this is the part that enables Xillybus’ driver for the PCIe interface in the kernel’s configuration file (.config):

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

“m” means that the driver is included as a kernel module. “y” means to include the driver in the kernel image.

Likewise, for XillyUSB (with kernel v5.14 and later):

```
CONFIG_XILLYUSB=m
```

The common way to make changes to .config is to use the kernel’s configuration tools: “make config”, “make xconfig” or “make gconfig”.

xconfig and gconfig are GUI tools that are easier to use, because they allow searching for the string “xillybus” in order to find Xillybus’ drivers. The driver is enabled by clicking on checkboxes. The textual representation of the .config file helps to verify that the correct options have been set.

On kernels that have a version below 3.18, it may be required to enable staging drivers before attempting to enable Xillybus. This results in the following line in the .config file.

```
CONFIG_STAGING=y
```

After enabling Xillybus' driver in the .config file, run the kernel compilation as usual.

Starting from kernel 5.14, an option with the name CONFIG_XILLYBUS_CLASS is automatically enabled when a driver for Xillybus or XillyUSB is enabled. This is a result of the configuration system's dependency rules. Changing this option manually is therefore unnecessary (and often impossible).

# 3

# The “Hello, world” test

## 3.1 The goal

Xillybus is a tool that is intended as a building block in a logic design. The best way to learn about Xillybus’ capabilities is hence to integrate it with your own user application logic. The demo bundle’s purpose is to be a starting point for working with Xillybus.

Therefore, the simplest possible application is implemented in the demo bundle: A loopback between two device files. This is achieved by connecting both sides of a FIFO to the Xillybus IP Core in the FPGA. As a result, when the host writes data to one device file, the FPGA returns the same data to the host through another device file.

The next few sections below explain how to test this simple functionality. This test is a simple method to verify that Xillybus operates correctly: The IP Core in the FPGA works as expected, the host detects the PCIe peripheral correctly, and the driver is installed properly. On top of that, this test is also an opportunity to learn how Xillybus works by making small modifications to the logic design in the FPGA.

As a first step, it’s recommended to make simple experiments with the demo bundle in order to understand how the logic in the FPGA and the device files work together. This alone often clarifies how to use Xillybus for your own application’s needs.

Aside from the loopback that is mentioned above, the demo bundle also implements a RAM and an additional loopback. This additional loopback is discussed briefly below. Regarding the RAM, it demonstrates how to access a memory array or registers. More information about this in section 4.4.

## 3.2 Preparations

A few preperations are required in order to perform the “Hello world” test:

- Xillybus’ driver is installed on the computer, as described in section 2.
- The FPGA must be loaded with the bitstream that is created from a demo bundle (without modifications). How to achieve this is explained in Getting started with the FPGA demo bundle for Xilinx or Getting started with the FPGA demo bundle for Intel FPGA.

  Those who use Xillinux (with Zynq or Cyclone V SoC), see Getting started with Xillinux for Zynq-7000 or Getting started with Xillinux for Cyclone V SoC (SoCKit): The demo bundle is already included in this system by default.
- Relevant for PCIe only: The FPGA was detected on the PCIe bus when the computer performed boot. This can be verified using the “lspci” command.
- Relevant for USB only: The FPGA is connected to the computer through a USB port, and the computer has detected the FPGA as a USB device. This can be verified using the “lsusb” command.
- You should be comfortable with using Linux command-line. Appendix A may help with this.

If these preparations have been done correctly, Xillybus’ device files should be available. For example, a file named /dev/xillybus_read_8 should exist.

## 3.3 The trivial loopback test

The easiest way to perform this test is using a Linux command-line utility that is named “cat”:

Open two terminal windows. On some computers, this can be done by double-clicking an icon with the name “Terminal”. If there is no such icon, search for it in the desktop’s menus.

On the first terminal window, type the following command at command prompt (don’t type the dollar sign, it’s the prompt):

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

This makes the “cat” program print out everything it reads from the xillybus_read_8 device file. Nothing is expected to happen at this stage.

*Those using XillyUSB will find the device file as xillyusb_00_read_8 instead. The “xillyusb” prefix is obvious, and the “00” index is intended to allow multiple USB devices to be connected to the same host. The naming convention for PCIe and for AXI is used in this guide.*

On the second terminal window, type

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

Note the > character. It tells “cat” to send everything that is typed on the console to xillybus_write_8 (redirection).

Now type some text on the second terminal, and press ENTER. The same text will appear on the first terminal. Nothing is sent to xillybus_write_8 until ENTER is pressed. This is a common convention on Linux computers.

Both of these two “cat” commands be stopped with CTRL-C.

If an error message is encountered while attempting these two “cat” commands, first verify that the device files have been created (i.e. that /dev/xillybus_read_8 and /dev/xillybus_write_8 exist). Also verify that there are no typos.

If the error is “permission denied”, this can be fixed as shown in section 2.7. However, note that a udev file becomes effective only when the kernel modules are loaded into the kernel. Refer to section 2.8 on how to reload the kernel modules. Alternatively, perform a reboot on the computer.

Another possibility to overcome the “permission denied” error is to work with the Xillybus device files as the root user. This is less recommended on desktop computers (but this is commonly done on embedded platforms). More about this in Appendix section A.4.

For other errors, follow the guidelines in section 2.8 on finding more information in /var/log/syslog or by using the “dmesg” command (or similar methods for obtaining the kernel log).

It’s also possible to perform trivial file operations. For example, without stopping the “cat” command in the first terminal, type the following in the second terminal:

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

Note that the FIFOs inside the FPGA are not at risk for overflow nor underflow: The core respects the ’full’ and ’empty’ signals inside the FPGA. When necessary, the

Xillybus driver forces the computer program to wait until the FIFO is ready for I/O. This is called blocking, which means forcing the user space program to sleep.

There is another pair of device files that have a loopback between them: /dev/xillybus_read_32 and /dev/xillybus_write_32. These device files work with a 32-bit word, and this is also true for the FIFO inside the FPGA. The “hello world” test with these device files will therefore result in similar behavior, with one difference: All I/O is carried out in groups of 4 bytes. Therefore, when the input hasn’t reached a boundary of 4 bytes, the last bytes from the input will remain untransmitted.

# 4

# Example host applications

## 4.1 General

There are four or five simple C programs that demonstrate how to access Xillybus' device files. These programs can be found in the the compressed file that contains the host driver for Xillybus / XillyUSB (available for download on the website). Refer to the "demoapps" directory, which consists of the following files:

- Makefile – This file contains the rules that are used by the "make" utility for the purpose of the programs' compilation.
- streamread.c – Reads from a file, sends data to standard output.
- streamwrite.c – Reads data from standard input, sends to file.
- memread.c – Reads data after performing seek. Demonstrates how to access a memory interface in the FPGA.
- memwrite.c – Writes data after performing seek. Demonstrates how to access a memory interface in the FPGA.

The purpose of these programs is to show correct coding style. They can also be used as a basis for writing your own programs. However, neither of these programs is intended for use in a real-life application, in particular because these programs don't perform well with high data rates. See chapter 5 for guidelines on achieving high bandwidth performance.

These programs are very simple, and merely demonstrate the standard methods for accessing files on a Linux computer. These methods are discussed in detail in the

Xillybus host application programming guide for Linux. For these reasons, there are no detailed explanations about these programs here.

Note that these programs use the low-level API, e.g. open(), read(), and write(). The more well-known API (fopen(), fread(), fwrite() etc.) is avoided, because it relies on data buffers that are maintained by the C runtime library. These data buffers may cause confusion, in particular because the communication with the FPGA is often delayed by the runtime library.

Those who download the driver for PCIe will find a fifth program in the "demoapps" directory: fifo.c. This program demonstrates the implementation of a userspace RAM FIFO. This program is rarely useful, because the device file's RAM buffers can be configured to be sufficient for almost all scenarios. fifo.c is hence useful only for very high data rates, and when the RAM buffer needs to very large (i.e. several gigabytes).

This program is not included along with XillyUSB's driver, because the data rates that may require fifo.c are not possible with XillyUSB.

## 4.2 Editing and compilation

If you're experienced with compilation of programs in Linux, you may skip to the next section: The compilation of Xillybus' example programs is done with "make" in the usual way.

First and foremost, change directory to where the C files are:

```
$ cd demoapps
```

To run a compilation of all five programs, just type "make" at shell prompt. The following transcript is expected:

```
$ 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
```

The five rows that start with "gcc" are the commands that are requested by "make" in order to use the compiler. These commands can be used for compilation of the programs separately. However, there is no reason to do so. Just use "make".

On some systems, the fifth compilation (of fifo.c) may fail if the POSIX threads library isn't installed (e.g. in some installations of Cygwin). This error can be ignored if you don't have an intention to use fifo.c.

The "make" utility runs compilation only on what is necessary. If only one file is changed, "make" will request the compilation of only that file. So the normal way to work is to edit the file you want to edit, and then use "make" for a recompilation. No unnecessary compilation will take place.

Use "make clean" in order to remove the executables that were generated by a previous compilation.

As mentioned above, the Makefile contains the rules for the compilation. The syntax of this file is not simple, but fortunately it is often possible to make changes to this file by using just common sense.

A Makefile relates to the files that are in the same directory as the Makefile itself. It is therefore possible to make a copy of the entire directory, and work on the files that are inside this replica. The two copies of the directory will not interfere with each other.

It is also possible to add a C file and easily change the Makefile, so that "make" also runs a compilation of this new file. For example, suppose that memwrite.c is copied to a new file, which is named mygames.c. This can be done with the GUI interface or with command line:

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

The next step is to edit the Makefile. There are many text editors and numerous ways to run each of them. On most systems, it is possible to start a GUI editor from the shell prompt by typing "gedit" or "xed". However, it's easier to find a GUI text editor in the computer desktop's menus. There are also many text editors that work inside the terminal window, for example vim, emacs, nano and pico.

Which editor to use is a matter of taste and personal experience. For example, this command can be used to start editing the Makefile:

```
$ xed Makefile &
```

The '&' at the end of the command tells the shell to not wait until the program finishes: The next shell prompt appears immediately. This is suitable for starting GUI applications, among others.

The row that should be changed in Makefile is:

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

This row is changed to:

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

The compilation of mygames.c will take place on the next time you type “make”.

## 4.3 Running the programs

The simple loopback example that is shown in section 3.3 can be done with two of the example programs.

Let’s assume that “demoapps” is already the current directory and that a compilation has already been done with “make’.

Type this in the first terminal:

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

This is the program that reads from the device file.

Note that the command begins with “./”: It is necessary to explicitly point at the directory of the executable. In this example, the expression “./” is used to request the current directory.

And then, in the second terminal window:

```
$ ./streamwrite /dev/xillybus_write_8
```

This works more or less like with the example with “cat”. The difference is that “streamwrite” doesn’t wait for ENTER before sending the data to the device file. Instead, this program attempts to operate separately on each character. In order to achieve this, the program uses a function called config_console(). This function is used only for the purpose of an immediate response to typing on the keyboard. This has nothing to do with Xillybus.

The examples above relate to Xillybus for PCIe / AXI. With XillyUSB, the names of the device files have a slightly different prefix. For example, xillyusb_00_read_8 instead of xillybus_read_8.

**IMPORTANT:**

*The I/O operations that are performed by streamread and streamwrite are inefficient: In order to make these programs simpler, the size of the I/O buffer is only 128 bytes. When a high data rate is required, larger buffers should be used. See section 5.3.*

## 4.4 Memory interface

The memread and memwrite programs are more interesting, because they demonstrate how to access memory on the FPGA. This is achieved by making function calls to lseek() on the device file. There is a section in Xillybus host application programming guide for Linux that explains this API in relation to Xillybus' device files.

Note that in the demo bundle, only xillybus_mem_8 allows seeking. This device file is also the only one that can be opened for both read and for write.

Before writing to the memory, the existing situation can be observed by using the hexdump utility:

```
$ 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
```

This output is the first 32 bytes in the memory array: hexdump opened /dev/xillybus_mem_8 and read 32 bytes from this device file. When a file that allows lseek() is opened, the initial position is always zero. Hence the output consists of the data in the memory array, from position 0 to position 31.

It's possible that your output will be different: This output reflects the FPGA's RAM, which may contain other values. In particular, these values may be different from zero as a result of previous experiments with the RAM.

A few words about hexdump's flags: The format of the output that is shown above is the result of "-C" and "-v". "-n 32" means to show first 32 bytes only. The memory array is just 32 bytes long, so it's pointless to read more than so.

memwrite can be used to change a value in the array. For example, the value at address 3 is changed to 170 (0xaa in hex format) with this command:

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

In order to verify that the command worked, it's possible to repeat the hexdump command from above:

```
$ 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  |...ł............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020
```

So evidently, the command worked.

In memwrite.c, the important part is where it says "lseek(fd, address, SEEK_SET)". This function call changes the position of the device file. Consequently, this changes the address of the array's element that is accessed inside the FPGA. The subsequent read operation or write operation starts from this position. Each such access increments the position according to the number of bytes that were transferred.

# 5

# Guidelines for high bandwidth performance

The users of Xillybus' IP cores often perform data bandwidth tests in order to ensure that the advertised data transfer rates are indeed met. Achieving these goals requires avoiding bottlenecks that may slow down the data flow considerably.

This section is a collection of guidelines, which is based upon the most common mistakes. Following these guidelines should result in bandwidth measurements that are equal to or slightly better than published.

It is of course important to follow these guidelines in the implementation of the project that is based on Xillybus, so that this project utilizes the IP core's full capabilities.

Often the problem is that the host doesn't process the data quickly enough: Measuring the data rate incorrectly is the most common reason for complaints about not being able to attain the published number. The recommended method is using the Linux' "dd" command, as shown in section 5.3 below.

The information in this section is relatively advanced for a "Getting Started" guide. This discussion also makes references to advanced topics that are explained in other documents. These guidelines are nevertheless given in this guide because many users carry out performance tests at the early stages of getting acquainted with the IP core.

## 5.1 Don't loopback

In the demo bundle (inside the FPGA) there is a loopback between the two pairs of streams. This makes the "Hello, world" test possible (see section 3), but this is bad for testing performance.

The problem is that the Xillybus IP core fills the FIFO inside the FPGA very quickly with

data transfer bursts. Because this FIFO becomes full, the data flow stops momentarily.

The loopback is implemented with this FIFO, so both sides of this FIFO are connected to the IP core. In response to the existence of data in the FIFO, the IP core fetches this data from the FIFO and sends it back to the host. This too happens very quickly, so the FIFO becomes empty. Once again, the data flow stops momentarily.

As a result of these momentary pauses in the data flow, the measured data transfer rate is lower than expected. This happens because the FIFO is too shallow, and because the IP core is responsible for both filling and emptying the FIFO.

In a real-life scenario there is no loopback. Rather, there is application logic on the FIFO's other side. Let's consider the usage scenario that attains the maximal data transfer rate: In this scenario, the application logic consumes the data from the FIFO as quickly as the IP core fills this FIFO. The FIFO is therefore never full.

Likewise for the opposite direction: The application logic fills the FIFO as quickly as the IP core consumes data. The FIFO is therefore never empty.

From a functional point of view, there's no problem that the FIFO occasionally becomes full or empty. This merely causes the data flow to stall momentarily. Everything works correctly, just not at the maximal speed.

The demo bundle is easily modified for the purpose of a performance test: For example, in order to test /dev/xillybus_read_32, disconnect user_r_read_32_empty from the FIFO inside the FPGA. Instead, connect this signal to a constant zero. As a result, the IP core will think that the FIFO is never empty. Hence the data transfers are performed the maximal speed.

This means that the IP core will occasionally read from an empty FIFO. As a result, the data that arrives to the host will not always be valid (due to underflow). But for a speed test, this doesn't matter. If the content of the data is important, a possible solution is that application logic fills the FIFO as quickly as possible (for example, with the output of a counter).

Likewise for testing /dev/xillybus_write_32: Disconnect user_w_write_32_full from the FIFO, and connect this signal to a constant zero. The IP core will think that the FIFO is never full, so the data transfer is performed at maximal speed. The data that is sent to the FIFO will be partially lost due to overflow.

Note that disconnecting the loopback allows testing each direction separately. However, this is also the correct way to test both directions simultaneously.

## 5.2 Don't involve the disk or other storage

Disks, solid-state drives and other kinds of computer storage are often the reason why bandwidth expectations aren't met. It is a common mistake to overestimate the storage medium's speed.

The operating system's cache mechanism adds to the confusion: When data is written to the disk, the physical storage medium is not always involved. Rather, the data is written to RAM instead. Only later is this data written to the disk itself. It's also possible that a read operation from the disk doesn't involve the physical medium. This happens when the same data has already been read recently.

The cache can be very large on modern computers. Several Gigabytes of data can therefore flow before the disk's real speed limitation becomes visible. This often leads users into thinking that something is wrong with Xillybus' data transport: There is no other explanation to this sudden change in the data transfer rate.

With solid-state drives (flash), there is an additional source of confusion, in particular during long and continuous write operations: In the low-level implementation of a flash drive, unused segments (blocks) of memory must be erased as a preparation for writing to the flash. This is because writing data to flash memory is allowed only to a blocks that is erased.

As a starting point, a flash drive usually has a lot of of blocks that are already erased. This makes the write operation fast: There is a lot of space to write the data to. However, when there are no more erased blocks, the flash drive is forced to erase blocks and possibly perform defragmentation of the data. This can lead to a significant slowdown that has no apparent explanation.

For these reasons, testing Xillybus' bandwidth should never involve any storage medium. Even if the storage medium appears to be fast enough during a short test, this can be misleading.

It's a common mistake to estimate performance by measuring the time it takes to copy data from a Xillybus device file into a large file on the disk. Even though this operation is correct functionally, measuring performance this way can turn out completely wrong.

If the storage is intended as a part of an application (e.g. data acquisition), it's recommended test this storage medium thoroughly: An extensive, long-term test on the storage medium should be made to verify that it meets its expectations. A short benchmark test can be extremely misleading.

## 5.3 Read and write large portions

Each function call to read() and to write() results in a system call to the operating system. A lot of CPU cycles are therefore required for carrying out these function calls. It's hence important that the size of the buffer is large enough, so that fewer system calls are carried out. This is true for bandwidth tests as well as a high-performance application.

Usually, 128 kB is a good size for the buffer of each function call. This means that each such function call is limited to a maximum of 128 kB. However, these function calls are allowed to transfer less data.

It's important to note that the example programs that were mentioned in section 4.3 (streamread and streamwrite) are not suitable for measuring performance: The buffer size in these programs is 128 bytes (not kB). This simplifies the examples, but makes the programs too slow for a performance test.

The following shell commands can be used for a speed check (replace the /dev/xillybus_* names as required):

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

These commands run until they are stopped with CTRL-C. Add "count=" in order to carry out the tests for a fixed amount of data.

## 5.4 Pay attention to the CPU consumption

In applications with a high data rate, the computer program is often the bottleneck, and not necessarily the data transport.

It's a common mistake is to overestimate the CPU's capabilities. Unlike common belief, when the data rate is above 100-200 MB/s, even the fastest CPUs struggle to do anything meaningful with the data. The performance can be improved with multi-threading, but it may come as a surprise that this should be necessary.

Sometimes an inadequate size of the buffers (as mentioned above) can lead to excessive CPU consumption as well.

It's therefore important to keep an eye on the CPU consumption. A utility program like "top" can be used for this purpose. However, the output of this program (as well as similar alternatives) can be misleading on computers with multiple processor cores (i.e. practically all computers nowadays). For example, if there are four processor

cores, what does 25% CPU mean? Is it a low CPU consumption, or is it 100% on a specific thread? If “top” is used, that depends on the version of the program.

Another thing to note, is how system calls’ processing time is measured and displayed: If the operating system’s overhead slows down the data flow, how is this measured?

A simple way to examine this is using the “time” utility. For example,

```
$ 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
```

The output of “time” at the bottom indicates that the time it took for “dd” to complete was 1.080 seconds. Out of this time, the processor carried out the user space program during 5 ms, and it was busy during 1.074 seconds with system calls. So in this specific example, it’s obvious that the processor was busy performing system calls almost all the time. This is not a surprise, because “dd” is not doing anything here.

## 5.5 Don’t make reads and writes mutually dependent

When communication in both directions is required, it’s a common mistake to write a computer program with only one thread. This program usually has one loop, which does the reading as well as the writing: For each iteration, data is written towards the FPGA, and then data is read in the opposite direction.

Sometimes there is no problem with a program like this, for example if the two streams are functionally independent. However, the intention behind a program like this is often that the FPGA should perform coprocessing. This programming style is based upon the misconception that the program should send a portion of data for processing, and then read back the results. Hence the iteration constitutes the processing of each portion of data.

Not only is this method inefficient, but the program often gets stuck. Section 6.6 of Xillybus host application programming guide for Linux elaborates more on this topic, and suggests a more adequate programming technique.

## 5.6 Know the limits of the host's RAM

This is relevant mostly to embedded systems and/or when using a revision XL / XXL IP core: There is a limited data bandwidth between the motherboard (or embedded processor) and the DDR RAM. This limitation is rarely noticed in usual usage of the computer. But for very demanding applications with Xillybus, this limit can be the bottleneck.

Keep in mind that each transfer of data from the FPGA to a user space program requires two operations on the RAM: The first operation is when the FPGA writes the data into a DMA buffer. The second operation is when the driver copies this data into a buffer that is accessible by the user space program. For similar reasons, two operations on the RAM are required when the data is transferred in the opposite direction as well.

The separation between DMA buffers and user space buffers is required by the operating system. All I/O that uses read() and write() (or similar function calls) must be carried out in this way.

For example, a test of an XL IP core is expected to result in 3.5 GB/s in each direction, i.e. 7 GB/s in total. However, the RAM is accessed double as much. Hence the RAM's bandwidth requirement is 14 GB/s. Not all motherboards have this capability. Also keep in mind that the host uses the RAM for other tasks at the same time.

With revision XXL, even a simple test in one direction might exceed the RAM's bandwidth capability, for the same reason.

## 5.7 DMA buffers that are large enough

This is rarely an issue, but still worth mentioning: If too little RAM is allocated on the host for DMA buffers, this may slow down the data transport. The reason is that the host is forced to divide the data stream into small segments. This causes a waste of CPU cycles.

All demo bundles have enough DMA memory for performance testing. This is also true for IP cores that are generated at the IP Core Factory correctly: "Autoset Internals" is enabled and "Expected BW" reflects the required data bandwidth. "Buffering" should be selected to be 10 ms, even though any option is most likely fine.

Generally speaking, this is enough for a bandwidth test: At least four DMA buffers that have a total amount of RAM that corresponds to the data transfer during 10 ms. The required data transfer rate must be taken into account, of course.

## 5.8 Use the correct width for the data word

Quite obviously, the application logic can transfer only one word of data to the IP core for each clock cycle inside the FPGA. Hence there is a limit on the data transfer rate because of the data word's width and bus_clk's frequency.

On top of that, there is a limitation that is related to IP cores with the default revision (revision A IP cores): When the word width is 8 bits or 16 bits, the PCIe's capabilities are not used as efficiently as when the word width is 32 bits. Applications and tests that require high performance should therefore use 32 bits only. This does not apply to revision B IP cores and later revisions.

The word width can be up to 256 bits starting with revision B. The word should be at least as wide as the PCIe block's width. Hence for a data bandwidth test, these data word widths are required:

- Default revision (Revision A): 32 bits.
- Revision B: At least 64 bits.
- Revision XL: At least 128 bits.
- Revision XXL: 256 bits.

If the data word is wider than required above (when possible), slightly better results are usually achieved. The reason is an improvement of the data transfer between the application logic and the IP core.

## 5.9 Slowdown due to cache synchronization

This issue does not apply to computers that are based upon CPUs that belong to the x86 family (32 and 64 bits). Those who use Xillybus with the AXI bus of a Zynq processor (e.g. with Xillinux) can also ignore this topic.

However, several embedded processors require an explicit synchronization of the cache when DMA buffers are used. This slows down the data transfer with the CPU's peripherals considerably.

This problem is not specific to Xillybus: Similar behavior is observed with all I/O that is based upon DMA, e.g. Ethernet, USB and other peripherals.

A slowdown because of the cache can be revealed by looking at the CPU consumption. If the CPU spends an unreasonable amount of time in the system call state ("sys"

row output of the “time” utility), this may indicate that the cache is the problem. This happens because the CPU is spending a lot of time performing cache synchronization.

However, it’s important to first rule out the possibility of small buffers (as mentioned in sections 5.3 and 5.7 above).

This problem never happens with the x86 family because these CPUs have coherent cache. Hence no cache synchronization is required. The same applies for Xillinux, because the IP core is connected to the CPU through the ACP port.

But when a Zynq processor uses Xillybus with the PCIe bus, this problem occurs. Several other embedded processors are also affected, in particular ARM processors.

## 5.10 Tuning of parameters

The parameters of the PCIe block in the demo bundles are chosen in order to support the advertised data transfer rate. The performance is tested on a typical computer with a CPU that belongs to the x86 family.

Also, the IP cores that are generated in the IP Core Factory usually don’t need any fine-tuning: When “Autoset Internals” is enabled, the streams are likely to have the optimal balance between performance and utilization of the FPGA’s resources. The requested data transfer rate is hence ensured for each stream.

It is therefore almost always pointless to attempt fine-tuning the parameters of the PCIe block or the IP core. With the default revision of IP cores (revision A) such tuning is always pointless. If such tuning improves performance, it’s very likely that the problem is a flaw in the application logic or in the user application software. In this situation, there is much more to gain by correcting this flaw.

However, in rare scenarios that require exceptional performance, it might be necessary to tune the PCIe block’s parameters slightly in order to attain the requested data rates. This is relevant in particular for streams from the host to the FPGA. Section 4.5 of The guide to defining a custom Xillybus IP core discusses how to perform this tuning.

Note that even when this fine-tuning is beneficial, it’s not the Xillybus IP core’s parameters that are modified. Only the PCIe block is adjusted. It’s a common mistake to attempt improving the data transfer rate by tuning the IP core’s parameters. Rather, the problem is almost always one of the issues that have been mentioned above in this chapter.

# 6

# Troubleshooting

The drivers for Xillybus / XillyUSB were designed to produce meaningful log messages. These are a few alternatives for obtaining them:

- The output of the “dmesg” command.
- The output of the “journalctl -k” command.
- In the log file. This depends on the operating system, for example /var/log/syslog or /var/log/messages.

It’s recommended to search for messages that contain the word “xillybus” or “xillyusb” when something appears to be wrong. It’s also advisable to occasionally examine the system log even when everything appears to work fine.

A list of messages from the PCIe / AXI driver and their explanation can be found at:

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

It may however be easier to find a specific message by using Google on the message’s text.

# A

# A short survival guide to Linux command line

People who are not used to the command line interface may find it difficult to get things done on a Linux computer. The basic command-line interface has been the same for more than 30 years. Hence there are plenty of online tutorials about how to use each and every command. This short guide is only a starter.

## A.1 Some keystrokes

This is a summary of the most commonly used keystrokes.

- CTRL-C: Stop the currently running program
- CTRL-D: Finish this session (close the terminal window)
- CTRL-L: Clear screen
- TAB : At command prompt, attempt an autocomplete on what is already written. This is useful with e.g. long file names: Type the beginning of the name, and then [TAB].
- Up and down arrows: At command prompt, suggests previous commands from the history. This is useful for repeating something just done. Editing of previous commands is possible as well, so it's also good for doing almost the same as a previous command.
- space: When computer displays something with a terminal pager, [space] means "page down".
- q: "Quit". In page-by-page display, "q" is used to quit this mode.

## A.2 Getting help

Nobody really remembers all the flags and options. There are two common ways to get some more help. One way is the “man” command, and the second way is the help flag.

For example, in order to know more about the “ls” command (list files in current directory):

```
$ man ls
```

Note that the ’$’ sign is the command prompt, which the computer prints out to say it’s ready for a command. Usually the prompt is longer, and it includes some information about the user and the current directory.

The manual page is shown with a terminal pager. Use [space], arrow keys, Page Up and Page Down to navigate, and ’q’ to quit.

For a short summary about how to run the command, use the `--help` flag. Some commands respond to `-h` or `-help` (with a single dash). Other commands print the help information when the syntax is incorrect. It’s a matter of trial and error. For the ls command:

```
$ 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
```

...

(and it goes on)

## A.3 Showing and editing files

If the file is expected to be short (or if it’s OK to use the terminal window’s scrollbar), displaying its content on the console can be done with:

```
$ cat filename
```

Longer files require a terminal pager:

```
$ less filename
```

As for editing text files, there are a lot of editors to choose from. The most popular (but not necessarily the easiest to start off with) are emacs (and XEmacs) as well as vi. The vi editor is difficult to learn, but it's always available and always works.

The recommended simple GUI editor is gedit, or xed, whichever is available. It can be started through the desktop menus or from the command line:

```
$ gedit filename &
```

The '&' at the end means that the command should be executed "in the background". Or simply put, the next command prompt appears before the command has completed. GUI applications are best started like this.

The 'filename' in these examples can be a path as well, of course. For example, to view the system's main log file:

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

To jump to the end of the file, press shift-G, while "less" is running.

Note that the log files are accessible only by the root user on some computers.

## A.4 The root user

All Linux computers have a user with the name "root" which has ID 0. This user is also known as the superuser. The special thing about this user is that it's allowed to do everything. Every other user has limitations on accessing files and resources. Not all operations are allowed for every user. These limitations are not imposed on the root user.

This is not just an issue of privacy on a multi-user computer. Being allowed to do everything includes deleting all data on the hard disk with simple command at shell prompt. It includes several other ways to mistakenly delete data, make the computer useless in general, or make the system vulnerable to attacks. The basic assumption in a Linux system is that whoever is the root user, knows what he or she is doing. The computer doesn't ask root are-you-sure questions.

Working as root is necessary for system maintenance, including software installation. The trick to not messing things up is thinking before pressing ENTER, and being

sure the command was typed exactly as required. Following installation instructions exactly is usually safe. Don't make any modifications without understanding what they're doing exactly. If the same command can be repeated as a user that isn't root (possibly involving other files), try it out to see what happens as that user.

Because of the dangers of being root, the common way to run a command as root is with the sudo command. For example, view the main log file:

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

This doesn't always work, because the system needs to be configured to allow the user to use "sudo". The system requires the user's password (not the root password).

The second method is to type "su" and start a session in which every command is given as root. This is convenient when several tasks need to be done as root, but it also means there is a bigger chance of forgetting being root, and write the wrong thing without thinking. Keep root sessions short.

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

This time, the root password is required.

The change in the shell prompt indicates the change of identity from a regular user to root. In case of doubt, type "whoami" to obtain your current user name.

On some systems, sudo works for the relevant user, but it may still be desired to invoke a session as root. If "su" can't be used (mainly because the root password isn't known) the simple alternative is:

```
$ sudo su
#
```

## A.5 Selected commands

And finally, here are a few commonly used Linux commands.

A few file manipulation commands (it's possibly better to use GUI tools for this):

- cp – Copy file or files.
- rm – Remove file or files.

- mv – Move file or files.
- rmdir – Remove directory.

And some which are recommended to know in general:

- ls – List all files in the current directory (or another directory, when specified). “ls -l” lists the files with their attributes.
- lspci – List all PCI (and PCIe) devices on the bus. Useful for checking if Xillybus has been detected as a PCIe peripheral. Also try lspci -v, lspci -vv and lspci -n.
- lsusb – List all USB devices on the bus. Useful for checking if XillyUSB has been detected as a peripheral. Also try lsusb -v and lsusb -vv.
- cd – Change directory
- pwd – Show current directory
- cat – Send the file (or files) to standard output. Or use standard input, if no argument is given. The original purpose of this command was to concatenate files, but it ended up as the Swiss knife for plain input and output from and to files.
- man – Show the manual page of a command. Also try “man -a” (sometimes there’s more than one manual page for a command).
- less – terminal pager. Shows a file or data from standard input page by page. See above. Also used to display a long output of a command. For example:

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

- head – Show the beginning of a file
- tail – Show the end of a file. Or even better, with the -f flag: show the end + new lines as they arrive. Good for log files, for example (as root):

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

- diff – compare two text files. If it says nothing, files are identical.
- cmp – compare two binary files. If it says nothing, files are identical.

- hexdump – Show the content of a file in a neat format. Flags -v and -C are preferred.
- df – Show the mounted disks and how much space there is left in each. Even better, “df -h”
- make – Attempt to build (run a compilation of) a project, following the rules in the Makefile
- gcc – The GNU C compiler.
- ps – Get a list of running processes. “ps a”, “ps au” and “ps aux” will supply different amounts of information.

And a couple of advanced commands:

- grep – Search for a textual pattern in a file or standard input. The pattern is a regular expression, but if it’s just text, then it searches for the string. For example, search for the word “xillybus”, as a case insensitive string, in the main log file, and show the output page after page:

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

- find – Find a file. It has a complicated argument syntax, but it can find files depending on their name, age, type or anything you can think of. See the man page.