Getting started with Xillybus on a Windows 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 Windows host. How to try out the basic functionality of the IP core is also shown.

This guide 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 or USB 3.x).

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

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 used with 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 `\\.\xillybus_`*something*. Likewise, the driver for XillyUSB creates device files with the format `\\.\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 Windows.

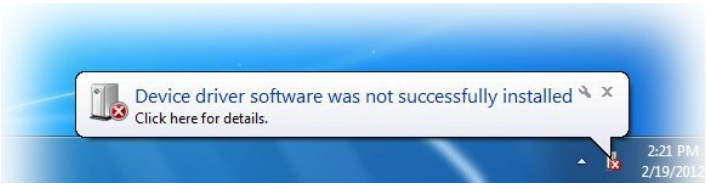
2

Installing the host driver

2.1 Installation procedure

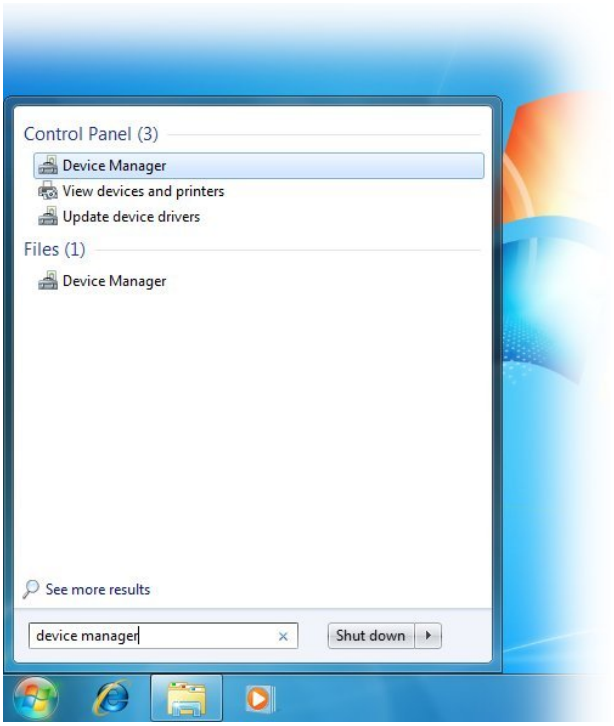
There is nothing special about installing the Windows driver for Xillybus or for XillyUSB. The procedure described below is the common method for installing a device driver from a specific location on the disk.

On the first time that Windows detects the PCIe Xillybus IP core during the boot process, it's likely that a warning bubble that looks like this will appear:

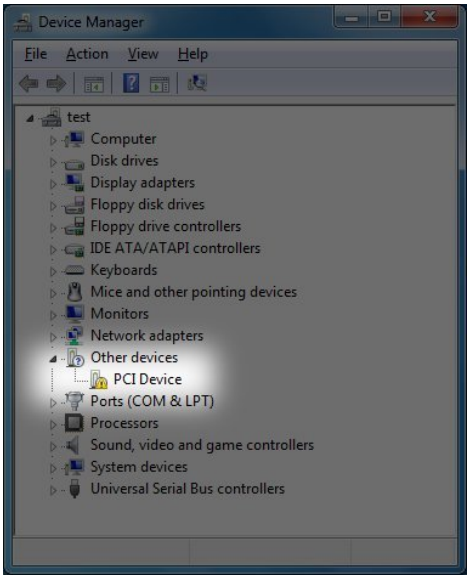


This notification says that new hardware has been found, but no relevant driver has been installed. This situation is normal, and is a sign Windows has detected something it doesn't yet recognize. The same outcome is expected when a XillyUSB device is connected to the computer for the first time.

In response to this event, start with running the Device Manager. This is easiest done by clicking on the "Windows start" button and then type "device manager" as shown in the image below. After this, click on the menu item at the top.



The opened Device Manager will look something like this (important part highlighted):



This screenshot relates to the PCIe scenario. For XillyUSB, a new item appears in the

"Universal Serial Bus controllers" group.

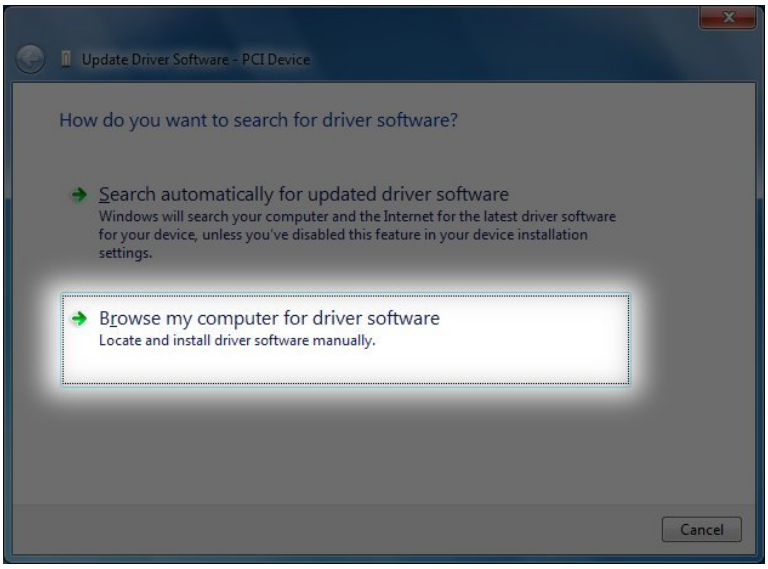
If a new entry doesn't appear in the Device Manager, there are several possible causes for this:

- The FPGA is loaded with the wrong bitstream, or not loaded with any bitstream at all.
- If the FPGA board receives its power supply from the PC, the bitstream is loaded into the FPGA when the PC computer is powered on. In this usage scenario, there's a possibility that the BIOS didn't detect the PCIe interface during boot, because the FPGA loaded the bitstream too slowly: The bitstream must already be inside the FPGA when the BIOS initializes the computer.

  Performing a Windows restart (without turning off the computer) is the safe way to fix this. However, performing Action $>$ Scan for New Hardware may work too.
- Incorrect configuration of the board (jumpers, DIP switches etc.), incorrect pin assignment, wrong frequency of the reference clock etc.

  Note that a problem of this sort is because the PCIe block on the FPGA is not detected. Such a problem has nothing to do with Xillybus, which uses the this PCIe block (which is supplied by AMD or Intel) for the interface with the PCIe bus.
- The Xillybus / XillyUSB driver is already installed. In this scenario, the Device Manager should look like the example shown at the end of the installation procedure.

Right-click the "PCI Device" item and pick "Update Driver Software...". The following window will be opened:



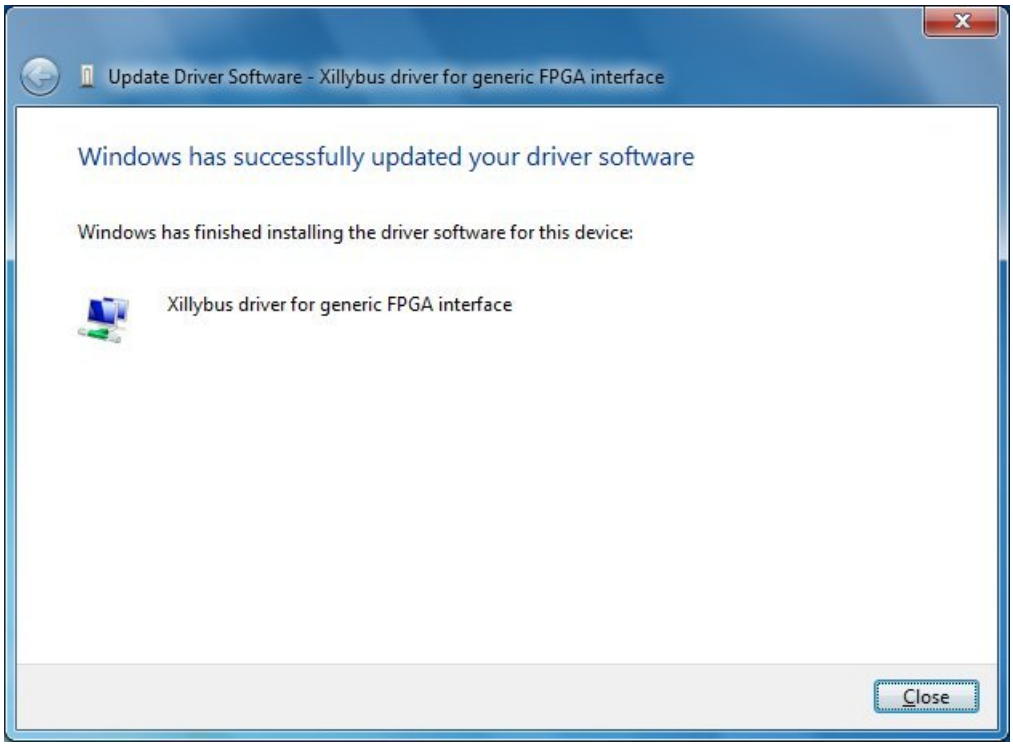
Choose “Browse my computer for driver software”. This is the next window:

Using the “Browse...” button, navigate to where the driver is stored (after the driver was uncompressed).

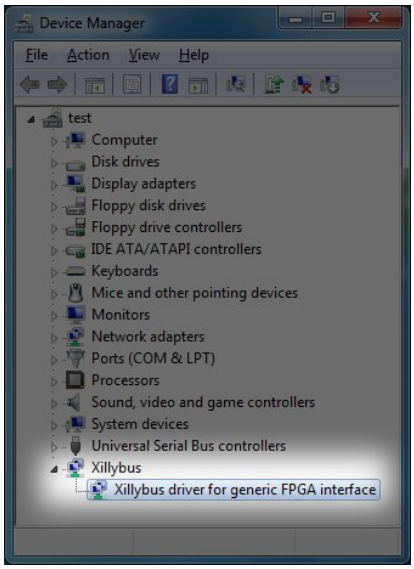
The next step is to confirm the installation:

Click on “Install”. The process of installing the driver takes 10-20 seconds on Windows 7. Newer versions of Windows usually require much less time.

The following window announces the successful completion of the installation:



The Device Manager will now show the newly installed device:



At this point, the driver is installed and has been automatically loaded into the system. This driver will be loaded every time the system carries out boot with the Xillybus IP core on the PCIe bus. XillyUSB's driver is loaded every time a relevant device is connected to the host.

It's recommended to set up the Event Viewer to display Xillybus' log messages, as explained next.

The screenshots shown above relate to Xillybus for PCIe. However, the process is the same for XillyUSB, with minor differences. In particular, then new group in the Device Manager is called "XillyUSB", and not "Xillybus".

2.2 The device files

The application computer program communicates with the Xillybus IP core by using the standard file I/O API. But instead of accessing regular files, device files are used in order to work with Xillybus.

The purpose of Xillybus' driver is hence to create these device files inside the operating system as a mechanism to communicate with the FPGA. These device files are called "Windows objects" in Microsoft's terminology.

Accessing Windows objects directly from a simple computer program can appear to be an undependable method. However, those who are familiar with Microsoft Windows' internals know that the software's interface with hardware is often done exactly like

this. It's indeed uncommon that these device files are exposed directly to application software. Rather, the hardware's manufacturer often supplies a DLL that allows the program to access the hardware through an API. Behind the scenes, this DLL uses device files in order to accomplish the required functionality.

Because Xillybus' interface is so simple, such a DLL is unnecessary. Hence the user application software accesses the device files directly. But unfortunately, Windows doesn't provide a simple method for obtaining a list of Xillybus' device files. This is because accessing Windows objects is considered an advanced technique. It's hence necessary to download a utility program in order to obtain this information from the operating system. But as explained below, the list of device files can be obtained from other sources as well.

The WinObj utility (available for download at Microsoft's site) allows navigating in Window's object tree. The Xillybus / XillyUSB device files can be found as symbolic links in the "subdirectory" with the name GLOBAL??. Other well-known Windows objects can be found at the same place, for example C: and COM1:.

There is also a command-line tool, which is named accesschk. This tool can be downloaded from Microsoft's website. The command for obtaining the names of the Xillybus / XillyUSB device files is:

```
> accesschk -o \\GLOBAL\?\?
```

Note that many other global device files are listed with this operation.

Even though it's possible to obtain a list of device files with these two tools, there is no need to do that: The names of the device files are known in advance.

Xillybus' device files have a prefix of the form `\\.\xillybus_` when the PCIe interface is used. For XillyUSB, the prefix is `\\.\xillyusb_nn_`.

This is the list of the device files that are generated by PCIe variant of the demo bundle:

- `\\.\xillybus_read_8`
- `\\.\xillybus_write_8`
- `\\.\xillybus_read_32`
- `\\.\xillybus_write_32`
- `\\.\xillybus_mem_8`

For a single XillyUSB device that is connected to the host, these are the device files:

- `\\.\xillyusb_00_read_8`
- `\\.\xillyusb_00_write_8`
- `\\.\xillyusb_00_read_32`
- `\\.\xillyusb_00_write_32`
- `\\.\xillyusb_00_mem_8`

As for a custom IP core that has been generated at the IP Core Factory: The list of device files can be found in the README file, which is included in the downloaded zip file.

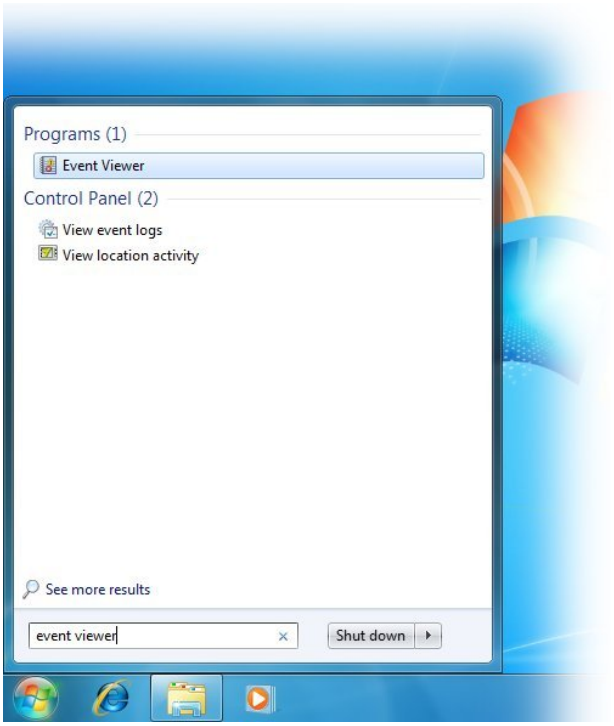
Note that in many programming languages (e.g. C/C++) an escape character is required before the backslashes in the file names. So it may be required to write the name of the device file as e.g. `\\\\.\\xillybus_read_8`.

2.3 Obtaining diagnostic information

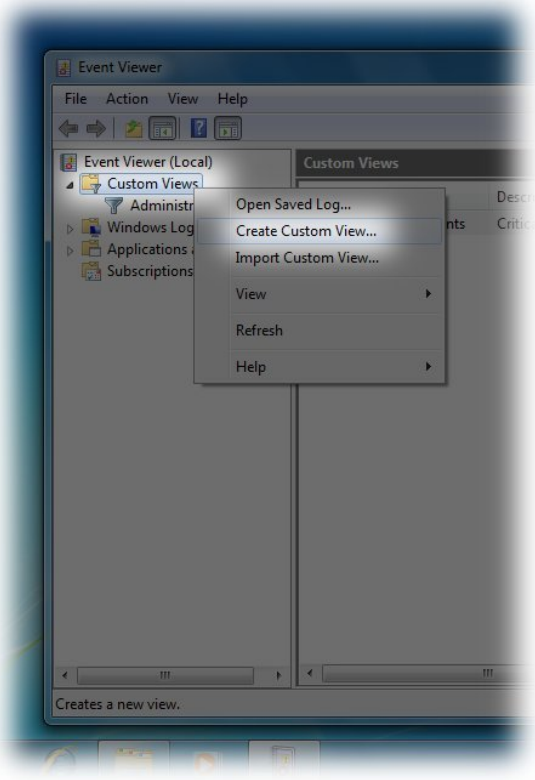
The driver for Xillybus / XillyUSB sends diagnostic messages to the operating system's main event logger. These messages include information about what went wrong when the driver fails to initialize (e.g. there wasn't enough memory for DMA buffers). Other messages that can be helpful in solving problems are also sent.

The procedure that is outlined next shows how to create a custom view in the Event Viewer. The purpose of this task is to display messages that are related to Xillybus or XillyUSB.

First, open the Event Viewer. This is easiest done by clicking on the "Windows start" button and then type "event viewer" as shown below. Click on the menu item at the top:

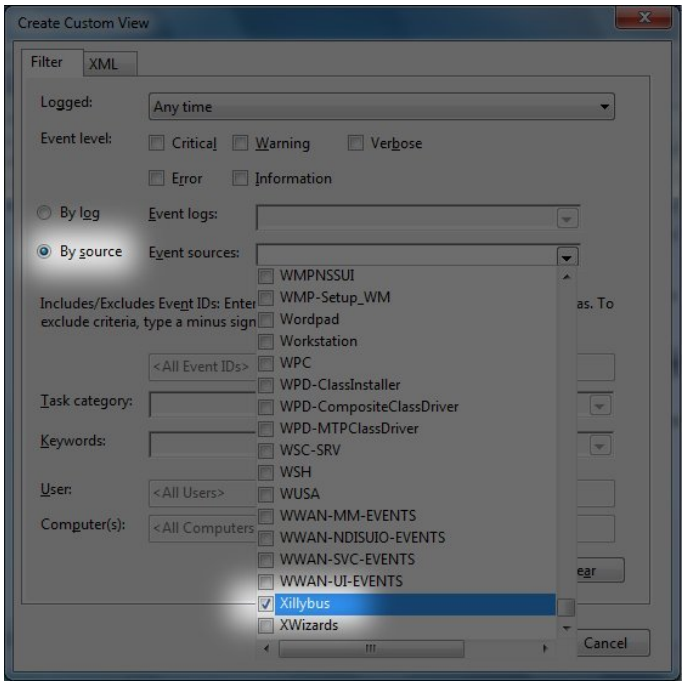


The Event Viewer will open. Right-click “Custom Views” and choose “Create Custom View...” from the menu.



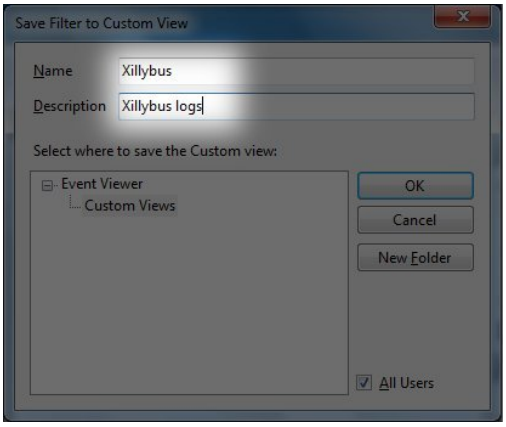
A window with the title “Create Custom View” will be opened. The purpose of this window is to define the filters that select which messages are displayed. Pick “By source”. Choose Xillybus in the drop-down menu. Keep the defaults for the other options.

In order to obtain the messages from XillyUSB instead, select XillyUSB in the menu.

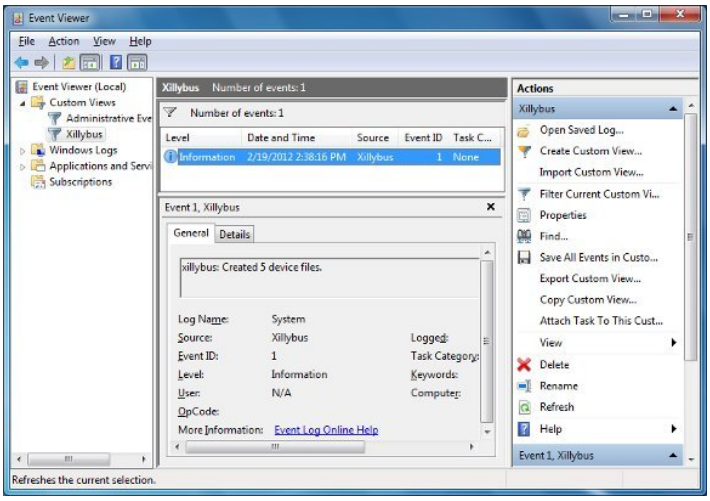


If the Xillybus / XillyUSB entry in the drop-down menu is not found, check if the driver is properly installed.

After clicking “OK”, another window will open for the purpose of assigning this custom view a name and a description. This is a matter of personal choice:



After clicking “OK”, the Event Viewer looks something like this:



The image above shows one message, which informs that Xillybus has started properly and that 5 device files have been created. This is what should be expected immediately after a successful installation of the driver when the FPGA is loaded with the demo bundle.

The messages are not deleted upon a reboot of the computer. Hence this custom view shows the history since the driver was installed (unless the history is deleted deliberately).

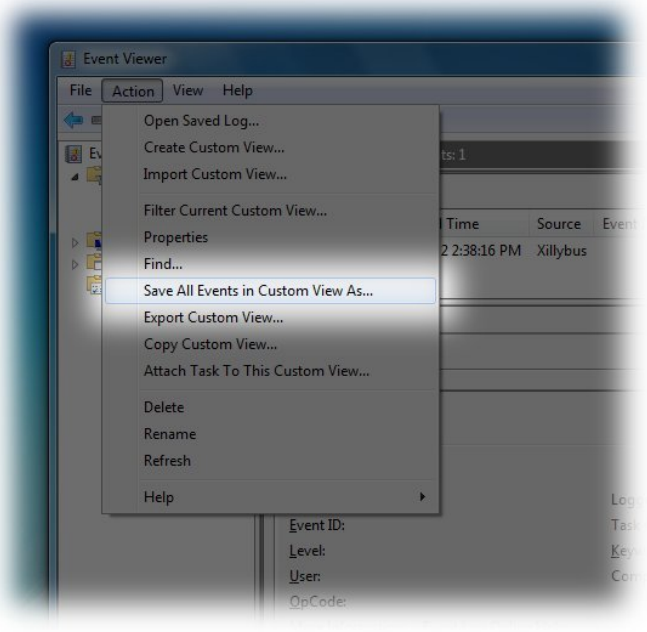
A list of messages from the PCIe driver and their explanation can be found at:

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

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

It's also possible to export the messages to a file. When asking for support, it's a good idea to send a file that contains the messages from the driver. These messages often contain valuable information.

In order to create such a file, choose the "Action" menu item and then choose "Save All Events in a Custom View As...":



A file selection window will be opened. For the purpose of asking for help, choose CSV as the output format.

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 3.4.

3.2 Preparations

The “Hello, world” test consists of running simple command-line programs, using Command Prompt windows.

As a first step, download the Xillybus package for Windows. It is a zip file that is available on the same web page that offers the driver. This zip file contains source code of these programs as well as executable binaries that are ready for running.

The easiest way is to use the executable binaries in order to carry out the “Hello world” test. However, it’s also possible to perform a compilation of these programs, as detailed in section 4.2.

Another possibility is to create a working environment that resembles Linux, as explained in section 4.3. If this possibility is chosen, follow the instructions for the “Hello world” test in the Getting started with Xillybus on a Linux host guide.

3.3 The trivial loopback test

A simple example of a loopback test with two Command Prompt windows is shown next.

Open a regular Command Prompt window and change directory to the “precompiled-demoapps” subdirectory of the Xillybus package for Windows. In order to use the results of your own compilation (as shown in section 4.2), change directory to XP32_DEBUG instead.

Type the following in this Command Prompt window:

```
> streamread \\.\xillybus_read_8
```

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

Note that the backslashes are not duplicated. If this had been done with Cygwin (and not in a plain Command Prompt window), it would be `\\\\.\\xillybus_read_8` instead.

Now open another Command Prompt window. Change to the same directory as the first Command Prompt, and type:

```
> streamwrite \\.\xillybus_write_8
```

This command sends anything that is typed in the second window to the device file (i.e. `\\.\xillybus_write_8`).

Type some text on the second Command Prompt window, and press ENTER. The same text will appear in the first Command Prompt window. Note that nothing is sent until ENTER is pressed. This is in accordance with the expected behavior of standard input.

If an error message was received while attempting these two commands, the following actions are suggested:

- Check for typos.
- Verify that the driver is installed and that the FPGA is detected as a Xillybus device: Open the Device Manager and compare with the last image in section 2.1.
- Check for errors in the Event Viewer, as explained in section 2.3.
- Ensure that the device files have been created, as explained in section 2.2 (search for `xillybus_read_8` and `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.

Also note streamwrite works on each row separately, and doesn't send anything to the FPGA before ENTER has been pressed. This is unlike the program with the same name for Linux.

There is another pair of device files that have a loopback between them: xillybus_read_32 and 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.

3.4 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 Windows 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.

In this section, a tool named "hexdump" is used to display the content of the FPGA's RAM. This tool can be found in the "unixutils" subdirectory of the Xillybus package for Windows. Alternatively, section 4.3 suggests other options for obtaining this tool.

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

```
> hexdump -C -v -n 32 \\.\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 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 \\.\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 \\.\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.

4

Example host applications

4.1 General

There are six C programs that demonstrate how to access Xillybus' device files. These programs can be found in the Xillybus package for Windows, which is a zip file that is available for download on the same web page that offers the driver.

Inside this file, there are precompiled executables in the "precompiled-demoapps" subdirectory.

The source code can be found in the "demoapps" subdirectory. These C programs are intended for Microsoft's Visual C++ compiler which can be downloaded free as part of Microsoft's SDK. These programs can also be used with Visual Studio.

It's also possible to run the sample programs inside Cygwin (refer to section 4.3). MinGW can also be used for this purpose. If one of these two tools is chosen, the source code for Linux should be used, and the instructions in Getting started with Xillybus on a Linux host should be followed. Also refer to section 4.4 regarding the substitution of /dev/ prefixes with `\\\\.\\`.

The "demoapps" subdirectory consists of the following files:

- Makefile – This file contains the rules that are used by the "nmake" 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.
- fifo.c – 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).
- winstreamread.c – Reads from a file, sends data to standard output. This program does the same as streamread.c, but winstreamread.c uses and demonstrates Microsoft's file I/O API instead of the standard API.

All programs (except for winstreamread.c) are written in classic Linux style, even though they are intended for compilation with Microsoft's compiler.

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. These methods are discussed in detail in the Xillybus host application programming guide for Windows. 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.

4.2 Compilation

As already mentioned, there is no need for a compilation for the purpose of trying out the example programs: The Xillybus package for Windows contains files that are ready to run on a Windows computer. But obviously, the compilation of these programs is necessary to make changes.

Those who are used to working with Microsoft Visual Studio will probably prefer using this compiler, and know how to use this tool. The example programs are simple Command Prompt applications.

However, the guidelines below are based upon Microsoft's software development kit (SDK) 7.1. This is an old and simple development kit, which is available for download at no cost. The instructions below are based on this software, mainly because a small number of steps are required to accomplish the task.

Download and install Windows SDK 7.1. Open Program Files in the "Start menu" and select Microsoft Windows SDK v7.1 >Windows SDK 7.1 Command Prompt. This opens a Command Prompt window, which has several environment variables configured for the purpose of compilation. The text in this window is displayed with a yellow font.

Change directory to where the C files are:

```
> cd \path\to\demoapps
```

To run compilation on all programs, type "nmake". The following transcript is expected:

```
> nmake

Microsoft (R) Program Maintenance Utility Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

 if not exist "XP32_DEBUG/" mkdir XP32_DEBUG
 cl -D_CRT_SECURE_NO_WARNINGS -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo  [ ... ]
 link /INCREMENTAL:NO /NOLOGO -subsystem:console,5.01 -out:XP32_DEBUG\ [ ... ]
 cl -D_CRT_SECURE_NO_WARNINGS -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo  [ ... ]
 link /INCREMENTAL:NO /NOLOGO -subsystem:console,5.01 -out:XP32_DEBUG\ [ ... ]
 cl -D_CRT_SECURE_NO_WARNINGS -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo  [ ... ]
 link /INCREMENTAL:NO /NOLOGO -subsystem:console,5.01 -out:XP32_DEBUG\ [ ... ]
 cl -D_CRT_SECURE_NO_WARNINGS -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo  [ ... ]
 link /INCREMENTAL:NO /NOLOGO -subsystem:console,5.01 -out:XP32_DEBUG\ [ ... ]
 cl -D_CRT_SECURE_NO_WARNINGS -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo  [ ... ]
 link /INCREMENTAL:NO /NOLOGO -subsystem:console,5.01 -out:XP32_DEBUG\ [ ... ]
 cl -D_CRT_SECURE_NO_WARNINGS -c -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -nologo  [ ... ]
 link /INCREMENTAL:NO /NOLOGO -subsystem:console,5.01 -out:XP32_DEBUG\ [ ... ]
```

The six rows that start with "cl" are the commands that are requested by "nmake" 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 "nmake". The same goes for the "link" commands, which perform linking on the object files and the libraries, and hence create executables.

The executables (and the object files) can be found in the XP32_DEBUG subdirectory. This subdirectory is created during the compilation process if necessary. As the directory's name implies, these files are intended for 32-bit Windows XP. However, the executables run on later versions of Windows, including 64 bit versions.

The "nmake" utility runs compilation only on what is necessary. If only one file is changed, "nmake" 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 "nmake" for a recompilation. No unnecessary compilation will take place.

Use "nmake 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 "nmake" also runs a compilation of this new file.

4.3 Using tools from Linux with Windows

People who work with Linux tend to use standard command line tools for carrying out simple tasks. These tools are less known among people who primarily use Windows. The main reason is that the command-line tools for Windows are unfortunately not as useful as the tools that exist on every Linux computer.

As already mentioned, it is possible to carry out the "Hello world" test as detailed in Getting started with Xillybus on a Linux host instead of the instructions in this guide. In order to do that in Windows, it is necessary to make a few tools available on the computer. There are a few alternatives for accomplishing that:

- Download and install two packages from the Gnuwin32 project: Coreutils and Util-Linux-NG. These two packages cover the needs of the "Hello world" test (and also supply programs that aren't required for this task). Note that even if these packages are installed with Gnuwin32's setup tool, there will not be change in the Command Prompt's execution path.
- Use the tools for Windows that are supplied by Xillybus: These can be found

in in the “unixutils” subdirectory of the Xillybus package for Windows. The tools that are obtained this way were selected from the Gnuwin32 packages in order to suffice for the “Hello world” test.

- Install Cygwin. Choosing this method means installing a whole system that offers a command-line interface that resembles Linux. An installation of this sort may include the GNU C compiler and other tools for software development. This is the recommended choice for those who are used to working with Linux' command-line.

4.4 Differences from Linux

When carrying out the “Hello world” test as described in Getting started with Xillybus on a Linux host on a Windows computer, there are a few differences to be aware of.

The most important difference is that the path to the device files is `\\.\`, and not `/dev/`. So for example, when the guide for Linux mentions `/dev/xillybus_read_8`, the correct file name for Windows is `\\.\xillybus_read_8`.

Because the name of the device file contains backslashes, there is a need for escape characters in some situations: The backslash itself is often treated as an escape character. Hence there's a need for two backslashes for each backslash in the file name. In other words, `\\.\` needs to be written as `\\\\.\\`. For example, when the guide for Linux mentions `/dev/xillybus_read_8`, the file name `\\\\.\\xillybus_read_8` should be used in some situations.

But it is not always so: When a program is executed from the Command Prompt, there is no need for an escape characters. The Command Prompt treats a backslash like any other character.

In most programming languages, there is a need for extra backslashes. Inside scripts there may be a need for extra backslashes. This depends on how the arguments are handled inside the script.

4.5 Cygwin' warning message

Extra backslashes are required with Cygwin's command-line interface. However, when the `\\\\.\\` prefix is used for the first time, Cygwin will probably display a warning as follows:

```
$ cat \\\\.\\xillybus_read_8
```

```
cygwin warning:
 MS-DOS style path detected: \\.\xillybus_read_8
 Preferred POSIX equivalent is: //./xillybus_read_8
 CYGWIN environment variable option "nodosfilewarning" turns off this warning.
 Consult the user's guide for more details about POSIX paths:
 \url{http://cygwin.com/cygwin-ug-net/using.html#using-pathnames}
```

This warning should be ignored.

Xillybus has been extensively tested with Cygwin, and the method that is shown above for accessing device files is correct. For normal file names, it's indeed a better idea to use forward slashes. But Cygwin does not translate `//./` into `\\.\`. Hence the use of backslashes is mandatory.

In order to avoid this warning, possibly follow the suggestion in the warning message regarding the environment variable.

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. This tool is available as an executable in the Xillybus package for Windows.

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 `\\.\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 `\\.\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 sections 4.1 and 3.3 (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 within Cygwin for a quick speed check (replace the `\\\\.\\xillybus\_*` names as required):

```
dd if=/dev/zero of=\\\\.\\xillybus_sink bs=128k
dd if=\\\\.\\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.

Refer to section 4.3 for more on using Cygwin for tests.

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. The Task Manager can be used for this purpose, for example. However, the information that this program displays 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? Different tools for system monitoring display this information in different ways.

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 6.5 of Xillybus host application programming guide for Windows 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 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 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 in the system's event log. It is therefore recommended to search for messages that are relevant when something appears to be wrong. This is done by applying a filter on the event log, as described in section 2.3.

It's also advisable to occasionally look at the event log even when everything appears to work fine.

A list of messages from the PCIe 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.