Xillybus host application programming guide for Linux

Xillybus Ltd.

www.xillybus.com

Version 3.0

1

Introduction

Xillybus was designed to present the Linux host with a simple and well-known interface, having a natural and expected behavior. The host driver generates device files that behave like named pipes. They are opened, read from and written to just like any file, but behave much like pipes between processes or TCP/IP streams. To the program running on the host, the difference is that the other side of the stream is not another process (over the network or on the same computer), but a FIFO in the FPGA. Just like a TCP/IP stream, the Xillybus stream is designed to work well with high-rate data transfers as well single bytes arriving or sent occasionally.

Since the interface with Xillybus is all through device files that are accessed like just any file, typically any practical programming language can be used, with no need for a special module, extension or any other adaption. If a file can be opened with the chosen language, it can be used to access the FPGA through Xillybus.

One driver binary supports any Xillybus IP core configuration: The streams and their attributes are auto-detected by the driver as initializes the device, and device files are created accordingly. These device files are accessed as `/dev/xillybus_something` (or `/dev/xillyusb_something` with XillyUSB).

During operation, a handshake protocol between the FPGA and host makes an illusion of a continuous data stream. Behind the scenes, the driver's buffers are filled and processed. Techniques similar to those used for TCP/IP streaming are used to ensure an efficient utilization of the buffers, while maintaining responsiveness for small pieces of data.

Since Xillybus I/O is carried out just like any device file I/O in Linux, there is apparently no need for a programming guide, as common programming practices can be employed.

Even so, communication with FPGA often involves tasks that are not typical to file I/O. This guide suggests methods to implementing common FPGA-related projects, as well as how to achieve optimal performance. Experienced programmers may choose different methods with equal success.

Much of this guide is no more than an outline of how robust and efficient I/O is implemented in UNIX systems. Those familiar with such techniques may find several parts in this guide redundant, and indeed they are; Xillybus was designed not to invent any new API, but rather behave like experienced programmers would expect it to.

The examples in this guide are given in plain C, for clarity and because it has a set of functions that are known to be closely related to low-level system calls. The techniques described can be implemented in several other languages, including script languages such as Perl and Python, in particular when the requirements for performance and synchronization between host and FPGA actions are less strict.

Some I/O can even be done with shell scripts and one-liners.

2

Synchronous streams vs. asynchronous streams

2.1 Overview

Each Xillybus stream has a flag, which determines whether it behaves synchronously or asynchronously. This flag's value is fixed in the FPGA's logic.

When a stream is marked asynchronous, it's allowed to communicate data between the FPGA and the host's kernel driver without the user space software's involvement, as long as the respective device file is open.

Asynchronous streams have better performance, in particular when the data flow is continuous. Synchronous streams are easier to handle, and are the preferred choice when tight synchronization is needed between the actions of the user space application and what happens in the FPGA.

In custom IP cores that are generated in the IP Core Factory, the selection between making each stream synchronous or asynchronous is automatically based upon the information about the stream's intended use, as declared by the tool's user when "autoset internals" is enabled. If the autoset option is turned off, the user makes this choice explicitly.

Either way, the "readme" file, included in the bundle that is downloaded from the IP Core Factory, specifies the synchronous or asynchronous flag for each stream (among other attributes).

In all demo bundles, the streams related to xillybus_read_* and xillybus_write_* are asynchronous. xillybus_mem_8 is seekable and therefore synchronous. When XillyUSB is used, the same applies to the respective xillyusb_* files.

2.2 Motivation for asynchronous streams

Multitasking operating systems such as Linux and Microsoft Windows are based upon CPU time sharing: Processes get time slices of the CPU, with some scheduling algorithm deciding which process gets the CPU at any given moment.

Even though there's a possibility to set priorities for processes, there is no guarantee that a process will run continuously, or that the periods of preemption have a limited duration, not even on a multiprocessor computer. The underlying assumption of the operating system is that any process can accept any period of CPU starvation. Real-time oriented applications (e.g. sound applications and video players) have no definite solution to this problem. Instead, they rely on the typical de-facto behavior of the operating system, and make up for the preemption periods with I/O buffering.

Asynchronous streams tackle this issue by allowing a data to flow continuously while the application is either preempted or busy with other tasks. The exact significance of this for streams in either direction is discussed next.

2.3 Streams from FPGA to host

In the upstream direction (FPGA to host), if the stream is asynchronous, the IP core in the FPGA attempts to fill the host driver's buffers whenever possible. That is, when the file is open, data is available and there is free space in those buffers.

On the other hand, if the stream is synchronous, the IP core fetches data from the user application logic (usually from a FIFO) only when the user application software on the host has a pending request to read that data from the file descriptor. In other words, when the user application software is in the middle of a read() function call.

Synchronous streams should be avoided in high-bandwidth applications, mainly for these two reasons:

- The data flow is interrupted while the application is preempted or doing something else, so the physical channel remains unutilized during certain time periods. In most cases, this leads to a significant bandwidth performance reduction.
- An overflow may occur on the FIFO in the FPGA during these time gaps. For example, if its fill rate is 100 MB/sec, a typical FPGA FIFO with 2 kByte goes from empty to full in around 0.02 ms. Practically, this means that any preemption of the user space program can potentially cause the overflow of the FIFO in the FPGA.

Despite these drawbacks, synchronous streams are useful when the time at which the data was collected at the FPGA is important. In particular, memory-like interfaces require a synchronous interface.

Data that has been received by the Xillybus IP core on the FPGA from the application logic is available for reading immediately by the host's user space application, regardless of whether the stream is synchronous or asynchronous.

2.4 Streams from host to FPGA

In the downstream direction (host to FPGA), a stream being asynchronous means that the host application's write() function calls will return immediately most of the time. More precisely, the calls to functions writing to the device file will return immediately if the data can be stored entirely in the driver's buffers. The data is then transmitted to the FPGA at the rate requested by the user application logic at the FPGA, with no involvement of the host's application software.

There is a slight difference between XillyUSB and the other Xillybus IP cores, regarding how soon the data is sent to the FPGA, on behalf of an asynchronous streams to the FPGA.

For the IP cores that are based upon PCIe or AXI, data is sent to the FPGA only when one of these happen:

- The current DMA buffer is full (there are several buffers for each stream).
- A flush is requested explicitly on the device file by the application software (see paragraph 3.4)
- The file descriptor is being closed.
- A timer expires, forcing an automatic flush if nothing has been written to the stream for a specific amount of time (typically 10 ms).

With a XillyUSB stream, the data is sent virtually immediately. More precisely, the driver attempts to queue USB transfers of a fixed size (typically 64 kB), but a smaller transfer is queued if there is data for transmission, and there's no other transfer queued for the related stream. Therefore, for each stream, there is never more than one queued transfer with less than the fixed size, but there is always at least one transfer in progress as long as there is data for transmission. This results in an efficient use of USB transfers as well as quick response to short data segments.

All in all, asynchronous streams on all IP cores (XillyUSB and other Xillybus IP cores) behave roughly the same, with XillyUSB having a quicker response time on short segments of data (no 10 ms delay).

On the other hand, if the stream is synchronous, a call to the low-level function writing to the device file will not return until all data has reached the user application's logic in the FPGA. In a typical application, this means that when the function call to write() returns, it indicates that the data has arrived to the FIFO that is connected to the IP core in the FPGA.

IMPORTANT:

Higher-level I/O functions, such as fwrite(), involve a buffer layer created by the library functions. Hence fwrite() and similar functions may return before the data has arrived at the FPGA, even for synchronous streams.

Synchronous streams should be avoided in high-bandwidth applications, mainly for these two reasons:

- The data flow is interrupted while the application is preempted or doing something else, so the physical channel remains unutilized during certain time periods. In most cases, this leads to a significant bandwidth performance hit.
- An underflow may occur on the FIFO in the FPGA during these time gaps. For example, if its drain rate is 100 MB/sec, a typical FPGA FIFO with 2 kByte goes from full to empty in around 0.02 ms. Practically, this means that any preemption of the user space program can potentially cause the underflow of the FIFO in the FPGA.

Despite these drawbacks, synchronous streams are useful when it's important for the application to know that the data has arrived to the FPGA. This is the case when the stream is used to transmit commands that must be executed before some other operation takes place (e.g. configuration of the hardware).

2.5 Uncertainty vs. latency

A common mistake it to require a low latency on asynchronous streams for the sake of synchronization between data. For example, if the application is a modem, there is usually a natural need to synchronize between received and transmitted samples.

This often leads to a misconceived design, based upon the notion the uncertainty in the synchronization is necessarily smaller than the total latency. To keep the uncertainty low, the latency, and hence buffers, are made as small as possible, leading to an overall system with difficult real-time requirements.

With Xillybus, the synchronization is easily made perfect (at the level of a single sample), as explained in paragraph 6.2. The limitation on latency is therefore derived from the need to respond quickly to arriving data, if there is such a need.

For example, with a modem, the maximal latency has an impact on how quickly the application's data source responds to data that is sent to it. In a camera application, the host may program the camera to adjust the shutter speed to compensate for changing light conditions. Data that arrives with a larger latency slows down this control loop.

These are the real considerations that need to be taken, and still, they are usually significantly less stringent than those derived from the misunderstanding of mixing uncertainty with latency.

3

I/O programming practices

3.1 Overview

Xillybus works properly with any programming language which is able to access files, and any API for accessing files is suitable.

In this guide there's an emphasis on the low-level API set, based upon functions such as open(), read(), write() and close(). This set is chosen over other well-known sets (e.g. fopen(), fwrite(), fprintf() etc.) because the low-level API's functions have no extra layer of buffers. These buffers can have a positive effect on performance, but with them there's no control over the actual I/O operations.

This is less important when data is transmitted constantly and no direct relation is expected between software operations and the I/O with the hardware.

An extra buffer layer can also cause confusion, making it look like there's a software bug where there isn't. For example, a function call to fwrite() can merely store the data in a RAM buffer without performing any I/O operation until the file is closed. A developer not aware of this may be mislead to think that the fwrite() failed because nothing happened on the FPGA side, when in fact the data is waiting in the buffer.

This section describes the recommended UNIX programming practices, using the low-level C run-time library functions. This elaboration is given here for the sake of completeness, as there is nothing specific to Xillybus about any of these practices.

The code snippets are taken from the demo applications described in Getting started with Xillybus on a Linux host. The device file names in these examples are those of the Xillybus IP core for PCIe / AXI. For XillyUSB, the prefix is xillyusb_00_* instead of xillybus_*.

3.2 Guidelines for reading data

Assuming that the variables have been declared as follows:

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

The device file is opened with the low-level open (the file descriptor is in integer format):

```
fd = open("/dev/xillybus_ourdevice", O_RDONLY);

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

A “Device or resource busy” (errno = EBUSY) error will be issued if the device file is already opened for read by another process (non-exclusive file opening is available on request). If “No such device” (errno = ENODEV) occurs, it’s most likely an attempt to open a write-only stream.

With the file opened successfully and `buf` pointing at an allocated buffer in memory, data is read with:

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

`numbytes` is the maximal number of bytes to read.

The returned value, `rc`, contains the number of bytes actually read (or a negative value if the function call completed abnormally).

Note that read() always returns immediately if the amount of data that was requested in `numbytes` is available. Otherwise, it will return after about 10 ms if there is any data available. If no data at all is available, read() sleeps until it can return with data.

The driver checks the availability of data in the sense that the IP core has received that data from the application logic in the FPGA. The mechanism of DMA buffers is transparent to the caller of the function read(), and never delays the delivery of data to the read() function call because a DMA buffer isn’t full, as explained in section A.3.5 of the Appendix.

IMPORTANT:

There is no guarantee that all requested bytes were read from the file, even on a successful return of read(). It's the caller's responsibility to make another function call to read(), if the completed amount of data was unsatisfactory.

The function call to read() should be followed by checking its return value as shown below ("continue" and "break" statements assume a while-loop context):

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

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

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

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

The first if-statement checks if read() returned prematurely because of a signal. This is a result of the process receiving a signal from the operating system.

This is not an error really, but a condition that forces the driver to return control to the application immediately. The use of the EINTR error number is just a way to tell the function's caller that there was no data read. The program responds with a "continue" statement, resulting in a renewed attempt to call the function read() with the same parameters.

If there is some data in the buffer when the signal arrives, the driver will return the number of bytes already read in `rc`. The application will not know that a signal has arrived, and according to UNIX programming convention, it has no reason to care: If the signal requires action (e.g. SIGINT resulting from a CTRL-C on keyboard), the responsibility for this action is either on the operating system, or a registered signal handler.

Note that some signals shouldn't have any effect on the execution flow, so if signals aren't detected as shown above, the program may suddenly report an error for no

apparent reason.

Handling the EINTR scenario is also necessary to allow the process to be stopped (as with CTRL-Z) and resumed properly.

The second if-statement terminates the loop if a real error has occurred after reporting a user-readable error message.

The third if-statement detects if end of file has been reached, which is indicated by a return value of zero. When reading from a Xillybus device file, the only reason for this to happen is that the application logic has raised the stream's _eof pin (which is part of the IP core's interface on the FPGA).

3.3 Guidelines for writing data

Assuming that the variables have been declared as follows:

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

The device file is opened with the low-level open (the file descriptor is in integer format):

```
fd = open("/dev/xillybus_ourdevice", O_WRONLY);

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

A "Device or resource busy" (errno = EBUSY) error will be issued if the device file is already opened for write by another process (non-exclusive file opening is available on request). If "No such device" (errno = ENODEV) occurs, it's most likely an attempt to open a read-only stream.

With the file opened successfully and `buf` pointing at an allocated buffer in memory, data is written with:

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

`numbytes` is the maximal number of bytes to be written.

The returned value, `rc`, contains the number of bytes actually written (or a negative value if the function call completed abnormally).

IMPORTANT:

There is no guarantee that all requested bytes were written to the file, even on a successful return of write(). It's the caller's responsibility to make another function call to write(), if the completed amount of data was unsatisfactory.

The function call to write() should be followed by checking its return value as shown below ("continue" and "break" statements assume a while-loop context):

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

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

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

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

The first if-statement checks if write() returned prematurely because of a signal. This is a result of the process receiving a signal from the operating system.

This is not an error really, but a condition that forces the driver to return control to the application immediately. The use of the EINTR error number is just a way to tell the function's caller that there was no data written. The program responds with a "continue" statement, resulting in a renewed attempt to call the function write() with the same parameters.

If some data was written before the signal arrived, the driver will return the number of bytes already written in `rc`. The application will not know that a signal has arrived, and according to UNIX programming convention, it has no reason to care: If the signal requires action (e.g. SIGINT resulting from a CTRL-C on keyboard), the responsibility for this action is either on the operating system, or a registered signal handler.

Note that some signals shouldn't have any effect on the execution flow, so if signals aren't detected as shown above, the program may suddenly report an error for no apparent reason.

Handling the EINTR scenario is also necessary to allow the process to be stopped (as with CTRL-Z) and resumed properly.

The second if-statement terminates the loop if a real error has occurred after reporting a user-writable error message.

The third if-statement detects if the end of file has been reached, which is indicated by a return value of zero. When writing to a Xillybus device file, this should never happen.

3.4 Performing flush on asynchronous downstreams

As mentioned in paragraph 2.4, data written to an asynchronous stream on a PCIe / AXI IP core is not necessarily sent immediately to the FPGA, unless a DMA buffer is full (there are several DMA buffers). This behavior improves performance by making sure that the allocated buffer space is utilized. This also improves the efficiency of the packets sent on the PCIe / AXI bus.

As also mentioned already, XillyUSB IP cores send the data virtually right away, even when the stream is asynchronous, as there's an efficient arrangement for that with the USB interface. Performing flush has therefore a significance with XillyUSB IP cores only when it involves waiting for the transmission to complete.

Streams to the FPGA undergo a flush automatically when closing the file descriptor, however this is a best-effort mechanism that can't be relied upon. The function call to close() is delayed until all data has arrived at the FPGA in a manner similar to the way write() function calls are delayed on synchronous streams. The significant difference is that close() waits up to one second for the flush to complete. If the flush isn't completed by then, close() returns anyhow, and issues a warning message in the system log. Note however that in some rare scenarios, the last few words of remaining data may be lost without any warning while closing a file descriptor.

It's also possible to request a flush of an asynchronous stream explicitly, by calling the function write() with a buffer that has a length of zero, i.e.

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

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

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

  break; // Flush successful
}
```

Please note the following:

- The manual page for UNIX doesn't define what a write() function call should do when the count is zero, leaving the choice to each device driver. This method for flushing is specific to Xillybus.
- Unlike close(), a write() as shown above returns immediately, regardless of when the data is consumed on the FPGA.
- Because of this, this kind of write() is pointless with XillyUSB. It has nothing to do, and indeed does nothing: The data is sent virtually immediately anyhow, and the write() function call wouldn't wait in any case.
- Since no data is read from the buffer, the buffer argument in the write() function call can take any value, including NULL, as demonstrated above.
- Using higher-level API, with a buffer with zero length, may not have any effect at all. For example, calling the function fwrite() to write zero bytes may simply return with nothing done, since what this function usually does is adding the data to a buffer created by the C run-time library.
- fflush() is irrelevant: It performs a flush of the higher-level buffer, but doesn't send a flush command to the low-level driver.
- There is no need perform a flush on streams in the other direction (from FPGA to host), and there's no way to do so. This is because a flush of such streams is automatically performed when a host's attempt to read data is about to put the process to sleep (i.e. block).

3.5 select() and nonblocking I/O

Even though not recommended, the Xillybus driver for Linux supports nonblocking calls and the select() function. Note that the driver for Windows doesn't support anything similar, so using this functionality makes the application harder to port if necessary. The recommended way to handle multiple sources is with multiple threads (and preferably RAM FIFOs) as demonstrated in the fifo.c example program, discussed in paragraph 4.4.

Function calls to select(), pselect() and poll() can be used like with any UNIX file descriptor, for read and write alike.

The nonblocking calls and select() features are not enabled in Xillybus IP cores that have been set up for as "Windows only" in the IP Core Factory.

For the sake of completeness, we shall revisit the code outline for reading data in paragraph 3.2, using nonblocking reads. This code merely demonstrates the conventional method for any nonblocking read from a file in UNIX.

The file is opened with the O_NONBLOCK flag:

```
fd = open("/dev/xillybus_ourdevice", O_RDONLY | O_NONBLOCK);

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

There is no difference in how the file is read, the arguments or the meaning of the return value:

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

But there is now another check on the return values: If `rc` is negative and EAGAIN is given as the error code, this means there was nothing to read. More precisely, there is no data in the driver's buffers, and the FIFO in the FPGA is empty.

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

  if ((rc < 0) && (errno == EAGAIN)) {
    // do something else
    continue;
   }

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

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

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

Note that the code above doesn't make sense unless something meaningful is done when the function call returns with an EAGAIN. Otherwise it just wastes CPU time by spinning in the while loop, instead of sleeping when there is no data to read.

For nonblocking writing, make the respective changes in the example in paragraph 3.3.

3.6 Monitoring the amount of data in driver's buffers

This topic is discussed in Xillybus FPGA designer's guide, in the section named "Monitoring the amount of buffered data".

3.7 XillyUSB: The need to monitor the quality of the physical data link

Unlike PCIe, the physical data link that is used with USB 3.0 has been observed generating bit errors. This is uncommon, and indicates a problem with one of the involved components, most likely the host's USB port or the cable.

The USB protocol provides a variety of mechanisms for overcoming bit errors when such occur, however the random nature of these errors puts the link protocol in states

that are rarely reached. As a result, this may reveal bugs in the host's USB controller. Such bugs, to the extent that they exist, are normally hidden, and cause a variety of weird behaviors.

Hence if the physical data link suffers from frequent bit errors, there's a significant risk that the USB connection will become stuck, spontaneously disconnected, or in rare cases, even cause errors in the application data.

XillyUSB provides a means for monitoring the health of the physical data link, by virtue of a dedicated device file, /dev/xillyusb_NN_diagnostics. The showdiagnostics utility (explained on this web page) exposes the information collected on this matter.

It's highly recommended that applications based upon XillyUSB continuously monitor the first five counters that are displayed by the showdiagnostics utility (relating to bad packets, errors detected and Recovery requests), and ensure that they don't increase. If they do so, and in particular if they increase repeatedly, the application software should suggest corrective actions, possibly one of the following:

- Disconnect and reconnect the USB plug to another port. This may help, because some motherboards have different ports connected to different brands of USB host controllers (usually to support later versions of the USB 3.x protocol).
- Disconnect and reconnect the USB plug on the same port. This might help if the analog signal equalizer (which cancels attenuations and reflections caused the physical signal path) ends up in a suboptimal state.
- Attempt using a different USB cable.

It's quite likely that an application continues to work flawlessly even in the presence of bit errors. The suggestion for corrective actions is therefore best done while taking into account that the user probably doesn't experience any visible problem.

The showdiagnotics.pl utility is a Perl script which can be used as reference code. Alternatively, the diagnostic utility for Windows, which is given as C source code, can be referred to.

Note that none of these problems is specific to XillyUSB. Rather, these issues are as likely to affect any USB 3.0 device, however XillyUSB offers means to detect them. Also, it's worth reiterating that PCIe links are not known to suffer from any similar issues, most likely due to the better controlled physical connections and signal routings.

4

Continuous I/O at high rate

4.1 The basics

There are four practices that are nearly essential to achieve a high-rate continuous data flow between the host and the FPGA:

- Using asynchronous streams
- Making sure the driver's buffers are large enough to compensate for time gaps between the I/O operations of the user space application.
- Having the user space application read data from the device file as soon as there is data available, or write data to it as soon as there is space available in the buffers.
- Never closing and reopening the device files while the FPGA keeps inserting or draining data.

XillyUSB presents additional challenges with maintaining a continuous flow of data, as explained on this web page.

Monitoring how much data is held in the driver's buffers at any given time is discussed in Xillybus FPGA designer's guide, in the section named "Monitoring the amount of buffered data".

The first item in the list above, of using asynchronous streams, is discussed in section 2. The second and third are discussed in the remainder of this section.

To understand the the fourth item, recall that the advantage of asynchronous streams is that data runs between the FPGA and host without the intervention of the user space application. This flow is stopped when the file is closed.

Specifically for a stream from the host to the FPGA, closing the file forces a flush of all data in the buffers, and the file is closed only after that is finished (or after one second). As a result, there is a time gap with no data flow from the moment that the file is closed, to when the file is open again (and data is written to the file descriptor).

As for streams from the FPGA, closing the file leads to loss of any data in the pipe that goes from the application logic in the FPGA, to the user space application in the host (i.e. the FPGA's FIFO and driver's buffers). The only way to avoid this loss is draining all data from this pipe before closing the file. Once again, there's a time gap with no data flowing, between closing the file and opening it again.

A common mistake is to use the EOF capability to mark data chunks (e.g. complete video frames), and by doing so, forcing the host to close and reopen the device file at known boundaries. However this significanly increases the risk for an overflow at the FPGA's FIFO.

It's important to keep in mind that the operating system may take away the CPU from a user space application at any given moment (preemption), so time gaps of several, and sometimes tens of milliseconds can occur between subsequent function calls in a program.

4.2 Large driver's buffers

One of the greatest challenges in transferring data at a high rate between the FPGA and host is to maintain a continuous flow. In applications involving data acquisition and playback, an overflow or shortage of data renders the system nonfunctional. To avoid this, the driver allocates large RAM buffers on the host for its own use. These buffers compensate for the gaps in time, during which the application isn't available to handle data transfers.

Xillybus allows allocation of huge driver's buffers, but this memory must be allocated from the pool of the operating system's kernel RAM. On some systems (32-bit systems in particular) the addressing space of such memory is limited to 1 GB by the Linux operating system, even if the total RAM available is significantly larger. In systems with RAM of less than 1 GB, (embedded Linux in particular), all memory may be used for driver's buffers.

Much larger buffers can be allocated on 64-bit systems when using an enhanced host driver, as discussed on this page:

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

Except for with XillyUSB, the driver's buffers are allocated when the Xillybus driver

is loaded (typically early in the boot process) and is freed only when the driver is unloaded from the kernel (usually during system shutdown). When the buffers are huge, this usually means that a significant part of the kernel's RAM pool is occupied by the driver's buffers. It's a fairly reasonable setting, since the application using these buffers is likely to be the main purpose of the machine it's running on.

A potential problem with huge buffers is tha they occupy contiguous segments of physical RAM. This is contrary to a buffer allocated in a userspace program, which is contiguous in virtual address space, but can be spread all over the physical memory, or even not occupy any physical RAM at all.

The pool of available memory becomes fragmented as the operating systems runs. This is why the Xillybus driver allocates its buffers as soon as possible, and retains them even when not actively used. Attempting to unload the driver and reload it at a later stage may fail for the same reason.

XillyUSB has a different approach to memory allocation, which is more tolerant to physical memory fragmentation. This is one of the reasons that its driver allocates RAM for its buffers when a device file is opened, and releases it when the file is closed.

Precautions should however be taken to avoid a shortage of kernel RAM. Xillybus' IP Core Factory's automatic memory allocation ("autoset internals") algorithm is designed not to consume more than 50% of the relevant memory pool, e.g. 512 MB for a PC computer, based upon the assumption that a modern PC has more than 1 GB of RAM installed. It's probably safe to go as high as 75% as well, which can be done by setting the buffer sizes manually.

Overallocation of buffers may lead to system instability. In particular, the operating system is likely to kill processes apparently randomly, whenever it fails to allocate RAM from the kernel pool.

4.3 RAM buffers in user space

For applications that require buffers larger than 512 MB on a 32-bit machine, it's recommended to do some of the buffering in user space RAM. On 64-bit machines, this option is rarely relevant, except for when the desired buffer size is very large, and not a power of 2 (2^N). For example, supplying a buffer of 62 GB for a stream is not possible by virtue of Xillybus DMA buffers, but can be achieved with user space RAM.

It may seem counterintuitive that the problem of I/O continuity can be solved by allocating a huge buffer in the user space application. Indeed, this solution doesn't help

when the operating system starves the application of CPU time. But if the operating system's scheduler is fairly well designed and the priorities are set right, a user space application will get its CPU slice often enough, even on a computer that is under heavy load.

It's important to pay attention to the first fill of the buffer: Modern operating systems don't allocate any physical RAM when a user space application requests memory. Instead, they just set up the memory page tables to reflect the memory allocation. Actual physical memory is allocated only when the application attempts to use it. This is a brilliant method for saving resources, but can have a disastrous impact on a data acquisition application: For example, consider what happens when data begins to rush in from a data source. The application writes the data to the buffer just allocated, but each time a new memory page is accessed, the operating system needs to get a new physical memory page. If there happens to be free physical RAM, or if there is a quick way to release physical memory (e.g. disk buffers which are already in sync with the disk), this memory juggling can go by unnoticed. But in the absence of immediate sources of physical RAM, disk operations may have to take place (RAM swapping to disk or flushing disk buffers), which can halt the application for too long.

The really bad news is that the ability to take the initial load of data depends on the overall system's state. Hence a program that usually works may suddenly fail, because some other program just did something data intensive on the same computer.

The natural solution is memory locking: mlock() tells the operating system that a certain chunk of (virtual) memory must be held in physical RAM. This forces allocation of physical memory immediately, so if disk operations are needed to complete the function call, it may take some time to return.

The operating system is reluctant to lock large chunks of RAM, as this impacts its overall performance. In most cases, there's a need to raise some limit in the shell or set up configuration files.

4.4 Overview of the fifo.c demo application

Among the demo applications, which can be downloaded for Linux and Windows, there's one called "fifo.c". It's an example of how to implement a RAM FIFO using two threads, which has been tested on 32-bit and 64-bit platforms.

For more about the demo applications, see Getting started with Xillybus on a Linux host.

Note that unlike everywhere else in the documentation, the word "FIFO" in this section

refers to a RAM buffer on the host, and not the FIFO in the FPGA.

The purpose of this program is to test fast streams, where a RAM FIFO is necessary to maintain a huge RAM buffer. In other words, if you need a buffer smaller than say, 16 GB, odds are that you don't need this program.

It can also be used a basis for modification and adoption in custom applications. It's designed with no mutexes, so no thread ever goes to sleep just because another thread holds the lock. Sleeping (blocking) does occur, of course, when the FIFO's state requires that (e.g. a read is requested from an empty FIFO).

This implementation without mutexes requires careful use of the API functions, as they're not reentrant. This is however no problem with one thread for reading, and one thread for writing.

To run it for data acquisition from a device file into a disk file with a buffer of 128 MB, type something like:

```
$ ./fifo 134217728 /dev/xillybus_async > dumpfile
```

If no file name is given as the second argument, the program reads from standard input.

There's probably a need to lift the limit on locked memory, using 'limit -l' at shell prompt, with root privileges (possibly use "su - your-username" as root to drop your privileges back to a regular user, and retain the updated limit). For a constant change in the limit, refer to your Linux distribution's docs.

The program creates three threads:

- read_thread() reads from standard input (or the file given in the command line) and writes the data into the FIFO
- write_thread() reads from the FIFO and writes to standard output
- status_thread() prints a status line to standard error recurrently

The third thread has no functional significance, and can be eliminated. It's also possible to have one of the read/write functionalities running in the main thread. For example, in a data acquisition application, it may be natural to launch only read_thread() to move data from the file descriptor to the FIFO, but consume the data from the FIFO in the thread of the main application.

4.5 fifo.c modification notes

If you want to modify the program, here are a few things to keep in mind:

- The fifo_* functions are not reentrant. It's safe to use them when each thread uses a set of functions that no other thread uses (which is a natural use).
- The function fifo_init() can take time to return, and should be called before an asynchronous Xillybus device file is opened.
- The thread that reads and the thread that writes in the applications always attempt the maximal number of bytes allowed in their I/O requests. This can be problematic in some cases, e.g. when the I/O source is /dev/zero and the destination is /dev/null. Both will complete the entire request in one attempt, so the FIFO will go from completely empty to completely full and over again. In such cases, it's more sensible to limit the requested number of bytes in calls to I/O functions.

4.6 RAM FIFO functions

Except for modifying the fifo.c example, it's possible to adopt a group of functions from the source code.

A section of FIFO API functions is clearly distinct in the fifo.c file. These functions can be used in custom applications, by following the example and according to the functions' description below.

IMPORTANT:

Even though the fifo_* functions are intended for use in a multi-threaded environment, these functions are ***not reentrant*** *. This means that one thread should call functions related to reading from the FIFO, and another thread should do writes, so each thread calls its separate set of functions.*

Except for an initializer, destroyer and a thread join helper, the API has four functions for reading and writing, two for each direction. Neither of these functions actually access the data in the FIFO; they merely maintain the FIFO's state and supply the information necessary to perform reads, writes, memory copies etc.

The intended execution procedure is as follows: The thread that reads from the FIFO calls the function fifo_request_drain(), which returns information about how many bytes

can be read, and a pointer from which data can be read. If the FIFO is empty, the thread will sleep until data arrives.

The user application then makes whatever use it needs with the data pointed to. After finishing to consume some or all of the data (write to a file, copy data, run some algorithm etc.), it calls the function fifo_drained() to inform the FIFO API how many bytes were actually consumed. The API releases the relevant portion of memory in the FIFO. If the thread that writes was sleeping because the FIFO was full, it is woken up.

Note that the thread that reads doesn't ask for a specific number of bytes. Rather, fifo_request_drain() tells the application how many bytes can be consumed, and the application reports back how many it chose to consume in fifo_drained().

As for the opposite direction, a similar approach is taken: The thread that writes calls the function fifo_request_write(). This function returns the number of bytes that can be written to the FIFO, or sleeps if the FIFO is full. The user application writes as many bytes it needs (but not more than fifo_request_write() allowed it to) to the address it got from fifo_request_write() and then reports back what it did to fifo_wrote().

We'll now go through each of these functions in detail.

4.6.1 fifo_init()

fifo_init(struct xillyfifo *fifo, unsigned int size) – This function initializes the FIFO's information structure and allocates memory for the FIFO as well. It also attempts to lock the FIFO's virtual memory to physical RAM, making it ready for immediate fast writing and preventing it from being swapped to disk.

fifo_init() allocates memory for a buffer of `size` bytes. `size` can be any integer (i.e. doesn't have to be a power of 2, 2^N) but a multiple of what the system considers `int` is recommended.

Note that this function can take several seconds to return: The request for a large portion of physical RAM may force the operating system to swap other processes' RAM pages to disk, or force disk cache flushing. In both cases, fifo_init() may have to wait for a lot of data to be written to disk before returning.

The function returns zero on success, nonzero otherwise.

4.6.2 fifo_destroy()

fifo_destroy(struct xillyfifo *fifo) – Frees the FIFO's memory after unlocking it, and releases thread synchronization resources. This function **should** be called when the main program exits, because even though the thread synchronization resources are released automatically in current implementations of Linux, their API doesn't guarantee this.

This function is of void type (hence returns nothing).

4.6.3 fifo_request_drain()

fifo_request_drain(struct xillyfifo *fifo, struct xillyinfo *info) – Supplies a pointer to read data from the FIFO as info->addr, and informs how many bytes can be read, beginning from that pointer, in info->bytes.

The `info` structure must **not** be the same one that is used for function calls to fifo_request_write(). Each thread should maintain a local variable of its own for this structure.

IMPORTANT:

*The number of bytes returned does **not** indicate how much data is left for reading in the FIFO: It may also reflect the number of bytes left until the end of the FIFO's memory buffer. Hence a significantly lower number is possible when the pointer comes close to the end of the buffer.*

The function also sets fifo->position to indicate the FIFO's current read position as a value between 0 and size-1, where `size` is the value that was given to fifo_init(). A nonzero fifo->slept indicates that the FIFO was empty upon invocation.

The function returns the number of bytes allowed for read (same as info->taken). But if the function fifo_done() has been called, and the FIFO is empty, fifo_request_drain() returns zero.

4.6.4 fifo_drained()

fifo_drained(struct xillyfifo *fifo, unsigned int req_bytes) – This function changes the FIFO's state to reflect the consumption of req_bytes bytes. If fifo_request_write() was sleeping because the FIFO was full, it will be woken up.

IMPORTANT:

*There is **no sanity check** on req_bytes. It's the user application's responsibility to make sure that req_bytes is not larger than info->bytes returned by the last function call to fifo_request_drain().*

This function is of void type (hence returns nothing).

4.6.5 fifo_request_write()

fifo_request_write(struct xillyfifo *fifo, struct xillyinfo *info) – Supplies a pointer to write data to the FIFO as info->addr, and informs how many bytes can be written, beginning from that pointer, in info->bytes.

The `info` structure must **not** be the same one that is used for function calls to fifo_request_drain(). Each thread should maintain a local variable of its own for this structure.

IMPORTANT:

*The number of bytes returned does **not** indicate how much data is left for writing in the FIFO: It may also reflect the number of bytes left until the end of the FIFO's memory buffer. Hence a significantly lower number is possible when the pointer comes close to the end of the buffer.*

The function also sets fifo->position to indicate the FIFO's current write position as a value between 0 and size-1, where `size` is the value that was given to fifo_init(). A nonzero fifo->slept indicates that the FIFO was full upon invocation.

The function returns the number of bytes allowed for write (same as info->taken). But if the function fifo_done() has been called, fifo_request_write() returns zero, even if the FIFO is not full (there is no point writing data into a FIFO that will never be read).

4.6.6 fifo_wrote()

fifo_wrote(struct xillyfifo *fifo, unsigned int req_bytes) – This function changes the FIFO's state to reflect the insertion of req_bytes bytes. If fifo_request_drain() was sleeping because the FIFO was empty, it will be woken up.

IMPORTANT:

*There is **no sanity check** on req_bytes. It's the user application's responsibility to make sure that req_bytes is not larger than info->bytes returned by the last function call to fifo_request_write().*

This function is of void type (hence returns nothing).

4.6.7 fifo_done()

fifo_done(struct xillyfifo *fifo) – This function is optional for use, and helps the application to quit gracefully if either of the threads (reading or writing) has finished. It merely sets a flag in the FIFO's structure and wakes up both threads if they were sleeping. By doing so, the fifo_request_drain() will return zero rather than sleeping if the FIFO is empty, and fifo_request_write() will return zero regardless.

This way, the callers of these functions know that the FIFO has no more use, and may act as necesssary, which is most likely to stop the execution of the thread.

Call this function when the data source feeding the pipe has ended (e.g. EOF reached) or when the data comsumer is no longer receptive (e.g. a broken pipe).

This function is of void type (hence returns nothing).

4.6.8 The FIFO_BACKOFF define variable

Sometimes it's not desirable to let the FIFO get full to the last byte. Even though there is no apparent reason avoiding that, it may be desirable to maintain a small gap between where data is written to and where its read from.

For example, FIFO_BACKOFF can be set to 8, so the last byte written to the FIFO never shares a 64-bit word with the first valid byte for read. This is a rather far-fetched precaution, but comes at the low price of 8 bytes of memory.

There is no need for this feature when working with Xillybus or XillyUSB.

5

Cyclic frame buffers

5.1 Introduction

In some applications, in particular real-time processing of video images, it's often desired to maintain a number of buffers, so that each buffer has a fixed size. In a video processing application, each such buffer contains one frame. This allows skipping frames or replaying them more than once, as necessary.

In a frame grabber application, an overflow condition can be handled by skipping one or more frames until there is a vacant buffer. For example, in a live view application, such overflow condition may occur when the viewing window is moved or resized. Dropping frames like this prevents the disruption of the continuous data flow from the video source, while maintaining a small latency.

In a frame replay application (e.g. a screen showing live output), the output image is repeated when there is no newer frame to display. This resolves situations where the source (e.g. a disk) momentarily stalls, causing the displayed image to freeze for a short while. While not completely graceful, it's better than having the stream going out of sync. In many cases, the image repetition mechanism, although somewhat crude, works well for overcoming differences in frame rates, in particular when the output's frame rate is considerably higher than the input's frame rate (e.g. 30 fps to 60 fps).

This section discusses how the FIFO demo application, which was introduced in paragraph 4.4, can be modified to manage a set of buffers of this sort.

5.2 Adapting the FIFO example code

There are similarities between maintaining a cyclic set of frame buffers and a FIFO. In fact, if each byte in the FIFO represents a frame buffer, the readiness to read or

write a certain byte in the FIFO is equivalent to the readiness to read or write an entire frame buffer.

For example, suppose a frame grabber application, where four frame buffers are allocated for containing the received image data. Suppose further that a FIFO of four bytes is set up to help managing these four frame buffers as follows:

The thread receiving the data starts from the first frame buffer, and continues to the next ones in a cyclic manner. Before starting to write to a new frame buffer, this thread checks that the four-byte FIFO isn't full. After it has completed a frame buffer, it writes a byte into the FIFO and goes to the next one if the FIFO isn't full.

The thread consuming the image data cycles through the frame buffer in the same order. Before attempting to read from a new frame buffer, it checks that the four-byte FIFO isn't empty. When it has finished with a frame buffer and is ready to go to the next, it reads a byte from the FIFO.

By sticking to this convention, it's guaranteed that the thread receiving the data will never overrun a frame buffer that hasn't been consumed, and that the consuming thread will never attempt to read from a frame buffer that contains invalid data. As a matter of fact, the number of bytes in the FIFO represents the number of valid frame buffers in the set.

Note that the values of the bytes written and read make no difference, so there's no actual need to allocate these four bytes of memory and store data in them. Only the FIFO's handshake mechanism plays a role.

Hence, the FIFO API outlined in paragraph 4.6 can be adopted as is:

- Call the function fifo_init() with the size parameter as the number of frame buffers (recall that `size` can be any integer). fifo_init() will allocate and lock memory for the FIFO, which will never be used (since each bytes just symbolizes a frame buffer). This waste of memory is negligible, but the relevant portions in the code can be removed to avoid future confusion.
- Call the function fifo_request_drain() to get a frame buffer to read from. info->position will contain the index to the frame buffer to use (numbering starts at 0). If no frame buffer is ready, fifo_request_drain() will sleep until there is.
- After reading from the buffer, call the function fifo_drained() with bytes_req=1.
- The functions fifo_request_write() and fifo_wrote() are called in the same way by the thread writing to the frame buffers.

- FIFO_BACKOFF should be set to zero. There is no point for this feature with frame buffers.

5.3 Dropping and repeating frames

Let's take the case of a continuous source of image frames which must never reach the state of overflow, given that the data comsumer may not always collect the data fast enough.

The idea is to prevent blocking on the thread, which transports data from the data source to the frame buffers. To achieve this, the following sequence should be looped on for each incoming frame:

- Call the function fifo_request_write() to find out which frame buffer to write to
- Write to the frame buffer pointed at by info->position
- When done writing, call the function fifo_request_write() again. This function call will surely not sleep (block), because no buffer has been reported as written to, since the previous call.
- If fifo_request_write() just returned a value larger than 1, call the function fifo_wrote() (with req_bytes=1, of course). A subsequent function call to fifo_request_write() will surely not sleep (block), because there were more than one buffer to spare, and only one was consumed. In fact, the next function call to fifo_request_write() can be substituted by just picking the next frame buffer.
- On the other hand, if fifo_request_write() returns just 1, don't call the function fifo_wrote(). Instead, use the current buffer again on the next loop executing for accepting incoming data, or just drain a whole frame from the data source to no particular destination.

Since this usage prevents blocking, it's possible to delete the while() loop in the implementation of fifo_request_write(), as it is never invoked. Further code reduction is possible by removing the relevant semaphore, as well as its initialization and destruction code. Leaving them in the code has a minimal effect, so this optimization is a mostly a matter of keeping the code readable.

A similar approach can be taken to repeat frames on the thread writing from the FIFO: Call the function fifo_request_drain() again just before calling the function fifo_drained(), and repeat the current frame if it returns less than 2.

6

Specific programming techniques

6.1 Seekable streams

A synchronous Xillybus stream can be configured to be seekable. The stream's position is presented to the application logic in the FPGA in separate wires as an address, so interfacing memory arrays or registers in the FPGA is straightforward, as shown in the demo bundle and example code.

This feature is useful in particular for setting up control registers in the FPGA. The synchronous nature of the stream ensures that the register in the FPGA is set before the low-level I/O function returns.

The following code snippet demonstrates how to write `len` bytes of data to address `address` in the memory or register space in the FPGA, assuming that these two variables are previously set.

```
int rc, sent;

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

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

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

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

  sent += rc;
}
```

`fd` is also assumed to be the value returned from a function call to open(), where the file was opened for write or read-write, and `buf` pointing to the buffer containing data to be written.

This example is an extension of the example shown in paragraph 3.3.

The only special thing in this code is the function call to lseek(), which sets the address. Only the SEEK_SET option should be used as the third argument, when calling the lseek() function.

Subsequent function calls update the address in accordance with the I/O stream's position, so there is no limitation on making multiple sequential writes after calling the function lseek().

For streams which are accessed as 16-bit or 32-bit words in the FPGA, the address given to lseek() must be a multiple of 2 or 4, respectively. The address presented to the application logic in the FPGA is maintained at all times as the stream's I/O position (initially as given to lseek()) divided by 2 or 4, respectively. For wider words, the same logarithmic rule applies.

The tell() function *may* return a correct position in the stream (i.e. the current address), but it's not a reliable source for this information. If in doubt, call the function lseek()

again.

lseek() can be used in the same way for reading data. See memwrite.c and memread.c in the demo application bundle (and their descriptions in Getting started with Xillybus on a Linux host).

6.2 Synchronizing streams in both directions

In certain applications, there's a need to synchronize several streams, possibly in opposite directions. For example, a radio transmission system may be implemented on the host, receiving digital samples from an A/D converter, which is connected to an RF receiver. Likewise, it may be sending digital samples to a D/A converter, connected to an RF transmitter. In scenarios of this sort, it's often needed to produce the digital samples for transmission so that the time of transmission is known in relation to the received samples. It may also be significant to know the exact time of a received signal.

Luckily, can be implemented with simple FPGA logic. One such solution is to ignore the received digital samples until the first sample for transmission arrives to the FPGA:

The host starts with opening the stream for reading samples from the FPGA. This stream is idle at this stage, because the FPGA drops its reception samples. Then the host opens the stream for writing samples for transmission to the FPGA, and begins writing data to it. As the first sample arrives to the FPGA, it stops ignoring received samples, and starts sending them towards the host.

As a result, the first sample that will be read from the FPGA will match the first sample written to the FPGA. The application on the host can therefore match the timing of any sample for transmission with any sample received just by matching their position in the respective stream. A slight correction may be needed to compensate for latency in the FPGA and the delay of the A/D and D/A, but such a latency is constant and known.

The streams have to be kept continuous at all times. How to achieve this was discussed in section 4.

This solution is satisfactory if maintaining a relative time relationship between transmission and reception is enough. When the samples need to be synchronized with an external event or another time reference, the same principle of skipping samples can be adapted as necessary to achieve the desired result.

Monitoring how much data is held in the driver's buffers at any given time is discussed in Xillybus FPGA designer's guide, in the section named "Monitoring the amount of buffered data".

6.3 Packet communication

Some applications require dividing the data stream into packets with varying length. The suggested solution uses two separate streams, and doesn't require the sender of the data to know the length of the packet at the time it starts to submit the packet itself through the channel.

The trivial case of packets with a fixed and known length is solved simply by transmitting them one after the other on one single stream. The receiver at the other side merely reads that fixed number of words for each packet. This is the typical solution in a video frame grabber or video replay application.

For the case of varying length packets, let's look at an upstream application, where the FPGA sends packets of bytes to the host. Let's assume that the FPGA knows the length of the packet only when the last byte arrives.

The implementation on the FPGA's side (i.e. the sender's side) is as follows:

- The FPGA writes all bytes in the packets to the first Xillybus stream.
- The FPGA resets a byte counter when it writes the first byte in a packet, and increments it for each additional byte it writes.
- When the last byte in a packet is written, the FPGA sends the counter's value on the second Xillybus stream. It contains the packet's length (minus one).

An important attribute of this solution is that the FPGA doesn't need to store the entire packet before sending it. It merely passes on the data as it arrives.

The user application at the host runs a loop as follows:

- Read one word from the second stream, containing the number of bytes in the next packet.
- Allocate memory for a buffer of the requested size if necessary.
- Read the given number of bytes into the buffer dedicated to the packet from the first stream.

Note that the host fetches the number of bytes to read before accessing the data, but the FPGA wrote these to the streams in the reverse order. The use of separate Xillybus streams allows this reversal.

A similar arrangement applies when the packets are sent from the host to the FPGA. The principle of using two streams, one for data and one for byte count remains. The

FPGA's application logic now gains the possibility to read the number of bytes from one stream before fetching the data from the other.

This arrangement is also extensible to passing other metadata in the non-data stream, e.g. the packet's destination or routing in some network (which is sometimes not known, when the first bytes arrive).

6.4 Emulating hardware interrupts

In small microcontroller projects, it's common to use hardware interrupts to alert the software that something has happened, and that the software needs to take some action. When the software runs as a userspace process in Linux, hardware interrupts are out of the question, and even software interrupts, like any asynchronous event, are not so pleasant to handle.

The suggested solution for a Xillybus-based system is to allocate a special stream for carrying messages. In its simplest form, a hardware interrupt is emulated by sending one single byte on that dedicated stream.

On the host side, the userspace application attempts to read data from the stream. The result is that when no "interrupt" is signaled, the application sleeps (blocking) until a byte arrives and wakes it up. The application handles the event, and then attempts to read another byte from the dedicated stream, hence going to sleep again if necessary, and so on.

To achieve a proper interaction between the main application and the interrupt routine, this dedicated stream can be read by a separate software thread or process. With this arrangement, the main code flows regardless of the thread that reads from the dedicated message stream, and the latter sleeps and wakes up, depending on messages sent.

A variant on this method uses the transmitted byte's value to pass information about the nature of the emulated interrupt. Also, each message can be longer than a single byte, if that makes sense in the implementation.

This method may appear to be a waste of logic resources, but Xillybus was originally designed not to consume much logic for each stream added, in order to make solutions like this sensible.

6.5 Timeout

In certain applications, there's a wish to limit the time an I/O operation may remain in blocking state, in particular when there's a chance of some hardware failure leading to the data flow being stopped.

Xillybus itself has been tested extensively to verify that it's never the reason why data is stopped this way, but data sources and data comsumers can stop for various reasons.

The less preferred way to tackle this is using select() or pselect() functions. They are intended when waiting for multiple file descriptors is needed, but also have a timeout functionality. It's not recommended to use these functions as their non-trivial interface may be a source of bugs, in particular in those special cases which a timeout is there to catch.

A more natural method is using Linux' alarm feature: It's a per-process timeout mechanism, which sends a signal (software interrupt) to the process when it expires. Please recall that a signal forces a read() or write() function call that is sleeping to return control immediately (see paragraphs 3.2 and 3.3). These functions return with a negative value and errno set to EINTR. In the previous examples, such interrupts were just a disturbance, but they are nevertheless useful for implementing a timeout .

Any process can receive several signals which are unrelated to its functionality. Receiving a signal is not an indication of a timeout condition in itself. There are several ways to tell, but the safest way is not to depend on that question at all: If the I/O operation took more than a certain amount of time, it's a timeout. So the most straightforward strategy is to measure time, as in the example shown next, which is based upon the one calling the function read() from paragraph 3.2.

The typical list of include files for this example is a bit long:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
```

Specific to this example, the following declarations are needed:

```
struct timespec before, after;
double elapsed;
```

The while-loop for reading data now starts as follows:

```
while (1) {
  if (clock_gettime(CLOCK_MONOTONIC, &before)) {
    perror("Failed to get time");
    exit(1);
  }

  alarm(2);
  rc = read(fd, buf, numbytes);

  if (clock_gettime(CLOCK_MONOTONIC, &after)) {
    perror("Failed to get time");
    exit(1);
  }
```

The time is measured before and after calling the function read() with clock_gettime(). This is the preferred function for measuring time differences, since it has access to a monotonic time measurement (as opposed to the system clock, which is modified by system utilities). Note that this function may require that the `-lrt` flag is added to gcc's arguments, so it loads the necessary library.

The function call to alarm() requests a signal after two seconds (the argument is the number of seconds). There is only one alarm timer for each process, so care must be taken not to override another use of the same timer, e.g. sleep() in some Linux implementations.

This code follows:

```
  elapsed = (after.tv_sec - before.tv_sec);
  elapsed += (after.tv_nsec - before.tv_nsec) / 1000000000.0;

   if (elapsed >= 2.0) {
     fprintf(stderr, "Timed out\n");
     exit(1);
   }
```

The time difference is calculated and stored in `elapsed`. It's a double-precision floating point variable to avoid word length portability issues in this simple example. But this can be done with an integer as well.

The condition is simple: If two seconds or more have elapsed between the time measurements, it's a timeout. The reason why read() returned isn't examined. It may be a signal or that data arrived eventually, but too late. In either case, it's an error.

Note that the function call to alarm() was made after the first time measurement took place, so a timeout is guaranteed to make the time differences at least two seconds long.

The while-loop continues just like before:

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

  if (rc < 0) {
    perror("read() failed");
    exit(1);
  }

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

As seen above, signals are still ignored. If the timer woke the process up, the time difference should reveal the timeout condition and exit.

Note that this method of implementing timeout is based upon a UNIX signal, which becomes a complicated issue in a multi-threaded environment. If multiple threads are deployed, it's easiest to make one of them the watchdog for the others.

Also note that in the example above, a timeout causes the process to terminate, which is easier to implement with a signal handler that performs this operation. The method shown above is more suitable when the corrective response is performed within the running process.

For higher precision of the timeout interval, consider using setitimer() instead.

6.6 Coprocessing / Hardware acceleration

Coprocessing (also known as *hardware acceleration*) is a technique that allows applications to take advantage of the logic fabric's flexibility to perform certain operations faster, cheaper, with a lower energy consumption or otherwise more efficient than a given processor. Whatever the motivation is, an efficient data transmission flow is

crucial to make coprocessing an eligible solution.

It's important to realize that the data flow in a coprocessing-based application is fundamentally different from the common programming data flow. To illustrate this difference, let's take, for example, a computer program that needs to calculate the square root of a number in floating point representation.

The programmer's straightforward way is to pass the number as an argument to sqrt(), call it, and wait until the function returns.

Suppose that it's desired to calculate the square root in the FPGA's logic fabric instead. A common *mistake* is to replace sqrt() with a special function that sends the value for calculation to the FPGA, waits for it to complete, and then returns with the result. Even though this is indeed a simple drop-in replacement for sqrt(), it's most likely going to be *slower* and otherwise less efficient than the original sqrt(): The time it takes for the data to travel across the bus in both directions, plus the time it takes for the FPGA to make the calculation, is probably considerably longer than the processor cycles needed by sqrt(). Having said that, calculating the square root on the FPGA can be much faster, if the data flow is designed correctly.

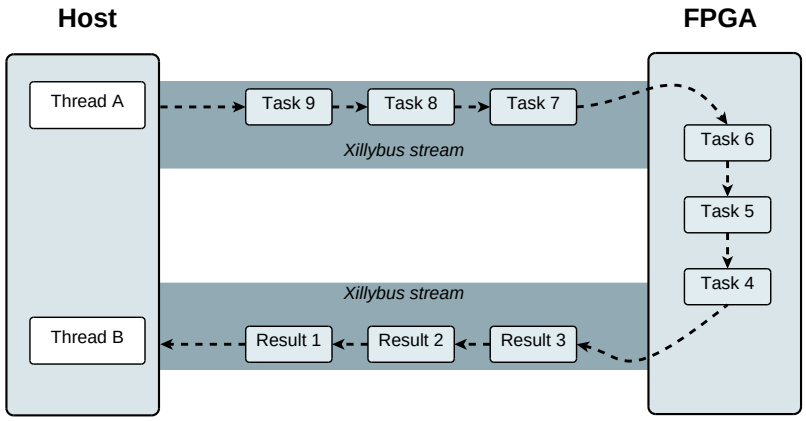
In order to overcome the latencies imposed by the bus and the FPGA's logic, there's a need to reorganize the software. In particular, the tasks in a program with a single thread need to be split into two or more threads (or processes). If multiple threads are not possible or desirable, other programming techniques can be utilized to mimic the behavior of multi-threading, but the programming paradigm is nevertheless multi-threaded.

Returning to the example of sqrt(), the call to this functions is divided into two threads: The first thread sends the data for square root calculations to the hardware (or some other form of data structure representing the request for operation). The second thread receives the results from the hardware and continues the processing from that point in the algorithm.

This doesn't seem to make much sense when looking at a single piece of data, but the motivation for coprocessing implies that there are many data items to handle. So the first thread sends a *flow* of data for calculation, and the second thread receives a *flow* of results.

This technique of *pipelining* minimizes the effect of the hardware's latency, since neither of the threads is effectively waiting the time of this latency. Instead, the latency influences the amount of processing items that are between the two threads – but the *throughput* depends only on the processing capabilities of the two threads and the FPGA logic.

The following conceptual drawing summarizes the idea.



The accelerated calculation of sqrt() is a relatively simple example, but it covers much of the challenge in utilizing coprocessing. Almost always, large parts of the computer program needs to be rewritten so that everything is driven by the pipeline's data flow.

Another issue to be aware of, is that since Xillybus works with read() and write(), it's possibly beneficial to group several data items for calculation before writing them to the stream towards the FPGA. Likewise, attempting to read more than one result item in each read() call may improve performance. The rationale behind this is that read() and write() are system calls with a certain overhead. If the data elements are small and transmitted at a high rate, these system call's overhead can be substantial. The case with sqrt() is a good example for this: A double float is typically 8 bytes long. A I/O system call of this length is quite inefficient, so concatenating several double float elements for a single system call will make a difference.

It's also worth to mention, that not all applications involve data chunks of constant lenghts. For example, using coprocessing for calculating hashes (e.g. SHA1) of arbitrary strings is likely to involve data elements for processing with different lengths. Section 6.3 suggest a solution for this.

A

Internals: How streams are implemented

A.1 Introduction

Even though using Xillybus doesn't require any understanding of its implementation details, some designers prefer knowing what happens under the hood, whether for curiosity or for verifying the eligibility of a certain solution.

This section outlines the main techniques implemented for creating continuous streams based upon DMA buffers. It applies to Xillybus for PCIe / AXI, but not to XillyUSB, which uses another mechanism.

The goal of how Xillybus is designed, is to make the underlying mechanisms transparent to the user, and to a large extent there is no reason to be aware of them. Please keep this in mind when going down to the technical details below, as they are very likely to be unnecessary for the sake of using Xillybus as an IP core. This part is more about how it works, and less about things the user needs to know.

There are two main sections below, one for the upstream flow, and one for the downstream. As similar techniques are employed in both directions, much of one section is a repetition of the other.

For the sake of simplicity, the descriptions focus on asynchronous streams, except for where it says otherwise. The end-of-file signal, as well as the option for non-blocking I/O, are not discussed here.

A.2 "Classic" DMA vs. Xillybus

Traditionally, data transport between hardware and software takes the form of a number of buffers with a fixed size. The data is organized into buffers with a fixed length,

which may or may not be filled completely. Each time a buffer is ready, some sort of signal is sent to the other side. For example, if the hardware has finished writing to a buffer, it may send an interrupt to the processor to inform the software that data is ready for processing. The software consumes the data, and informs the hardware that the buffer can be written to again, typically by writing to some memory-mapped register. Typically, both the sides access the buffers in a round-robin manner.

Xillybus presents a continuous stream transport to the user interface, both on the FPGA and the software side. Under the hood, Xillybus uses the traditional round-robin paradigm with a set of DMA buffers.

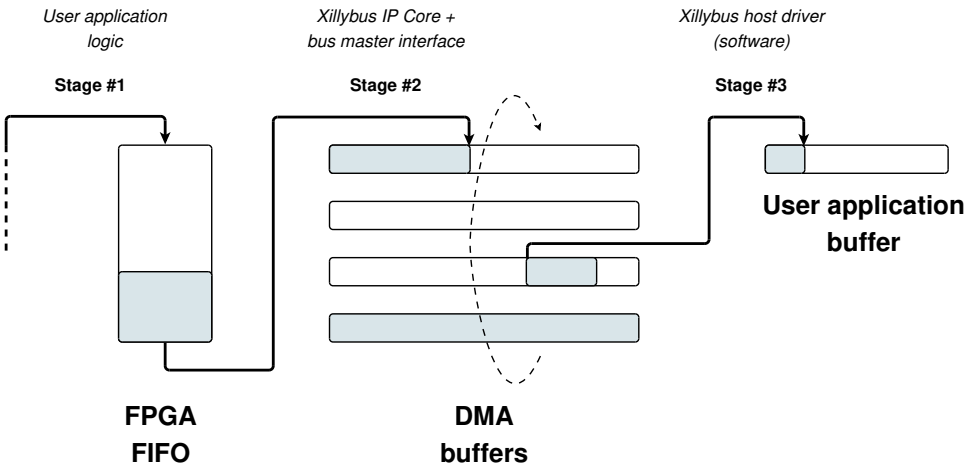
However the techniques described below are employed to create an illusion of a continuous stream, so that the user can ignore the existence of the underlying data transport. In particular, even if the application consists of sending data in fixed-sized chunks, there is no need to match the DMA buffers' size to the application data, as explained below.

A.3 FPGA to host (upstream)

A.3.1 Overview

The figure below depicts the flow in the FPGA to host (upstream) direction. The shaded areas represent data that is yet to be consumed, in the respective storage elements.

In this example, four DMA buffers are shown, even though the number of these can be configured in the IP Core Factory.



The data flows towards the host in three stages, as detailed next.

A.3.2 Stage #1: Application logic to intermediate FIFO

The user application logic in the FPGA pushes data into the FIFO that connects between the user application logic and the Xillybus IP core. There is no requirement on when or how much data is pushed, except for respecting the FIFO's "full" signal to avoid overflow.

A.3.3 Stage #2: Intermediate FIFO to DMA buffer

In this stage, the Xillybus IP core copies the data from the FIFO to a DMA buffer in the host's RAM. To accomplish this, the core uses some bus master interface (PCIe, AXI4 etc) to write data directly to the host's memory, without the intervention of the host's processor.

A pool of DMA buffers is allocated in the host's RAM. The lifecycle of each DMA buffer is like in many similar settings: In the beginning, all DMA buffers are empty and conceptually belong to the hardware. The hardware writes data to the buffers in a round-robin manner: When it has finished writing to a certain buffer, it informs the host that the buffer is ready for use (the buffer is handed over to the host, so to speak), after which it continues to write on the following buffer. The host may then consume the data in the buffer handed over to it, after which it informs the hardware that the buffer can be written to again (the host returns the buffer to the hardware).

The data flow of stage #2's is controlled by the FIFO's "empty" signal and by the availability of space in the pool of DMA buffers. When the Xillybus IP core senses a low "empty" signal from the FIFO, and there is space left in some DMA buffer, it fetches data from the FIFO and writes it into a DMA buffer. When the FIFO becomes empty again, or there is no space in any DMA buffer, the IP core's internal state machine stops fetching data momentarily, and then continues from where it left off in the DMA buffer.

While the data flow is stalled, the IP core might be busy with other activities, for example copying data on some other stream's behalf (i.e. draining another intermediate FIFO). As a result, there might be a random delay between the time that the FIFO changes the "empty" signal to low, and the resumption of fetching data from it. This delay varies, but overall, the IP core guarantees that a FIFO of 512 words will not reach a state of overflow (as long as the average rate is within limit).

Each DMA buffer may be filled completely before handing it over to the host, or may

be submitted to the host partially filled. The conditions for handing over a partially filled buffer are detailed later (section A.3.5), as they require some understanding of the software's behavior.

The case of synchronous streams is quite similar, except that the Xillybus IP core waits for an explicit request for a certain amount of data before fetching data from the intermediate FIFO.

A.3.4 Stage #3: DMA buffer to user software application

This stage is implemented on Xillybus' driver on the host by responding to read() system calls (or the counterpart IRPs on Microsoft Windows). According to the well-established API, the read() request includes a buffer that is supplied by the user application, as well as the size of the buffer, which is also the maximal number of bytes to read. The function call may return after reading the maximal number of bytes (complete fulfillment) or less.

The driver starts by checking the DMA buffers that are handed over to it, to determine whether there is enough data for consumption in the DMA buffers to allow for a complete fulfillment of the read() request. If so, it copies the data to the user's buffer, possibly returning DMA buffers to the hardware, and returns from the system call.

Otherwise, the standard API for a read() function call allows the driver to either return with less than the number of requested bytes, or wait (sleep) for any period of time. The driver is designed not to return too often with little data (which may cause a lot of read() function calls with little data each, hence wasting CPU cycles), but also avoid unnecessary latency. The dilemma is what to do if there is less data in the DMA buffers than required by the read() function call: To return with a partial fulfillment or wait (and how much to wait).

The chosen strategy is to wait for up to 10 ms for more data, and then return with whatever was available (or wait indefinitely if no data is available, as the standard API requires). This results in a fairly responsive return time, but limits the overhead to 100 read() function calls per second, if the read() function caller requests more than the data available all the time.

This is not to say that read() function calls necessarily have a latency of 10 ms: If the user space application knows in advance how many bytes should be ready, it may request no more than that number. By doing so, it ensures a latency which is in the order of magnitude of microseconds.

There is however a tricky part: The host knows about the DMA buffers that have been

handed over to it, but there may be a partially filled DMA buffer, which the host isn't aware of. So it might be, that there actually is enough data to fulfill a read() function call completely, if the partially filled DMA buffer is taken into consideration.

In order to handle this case properly, the driver checks whether the missing number of bytes would fit into a partially filled buffer. If this is indeed the case, it informs the hardware how much data will be enough. The driver then starts the 10 ms wait. This gives the hardware a chance to immediately send a partially filled buffer, if it indeed allows completing the read() function call entirely.

If and when the partially filled buffer reaches the necessary amount (possibly right away), the hardware hands it over to the host, which then completes the read() function call immediately.

When the 10 ms period is over, the driver returns with as much data it has available. If there is no data at all, the driver sends a request to the hardware to pass any partially filled buffer it has. The purpose is to return as soon as there is any data, since the 10 ms period is already over.

In all situations, when a DMA buffer has been consumed completely, the driver returns it to the hardware (i.e. informs the hardware that it can be written to again).

A few words on synchronous streams: The flow is the same in principle, except that data is never available in the DMA buffers when the read() function call is invoked. This is because the hardware is allowed to copy data from the FPGA's FIFO only when it's instructed to. Accordingly, the read() function call for synchronous streams involves informing the hardware on the amount of data it should copy. The waiting mechanism remains the same: First 10 ms, and then require any partially filled buffer.

A.3.5 Conditions for handing over partially filled buffers

The cases for handing over partially filled buffers can be deduced from the above, and are listed here for convenience.

The general rule is that a partial buffer is handed over to the host if the hardware has been informed that such early submission will result in an immediate return of the read() function call, which happens in either of three conditions:

- The host is currently handling a read() function call, which will be fulfilled completely when the current partially filled buffer is handed over.
- A read() function call stands at zero bytes, and has reached the time limit (i.e. 10 ms).

- On synchronous streams only: When the hardware has completed fetching the amount requested by the host.

Note that when FIFO becomes empty, it's *not*, by itself, a reason for a DMA buffer submission.

A.3.6 Examples

Let's consider the following simple case of an 8-bit asynchronous stream. Suppose that a stream starts with not containing any data, after which the FIFO is filled with a single element (that is, one byte). The application program on the host then calls the read() function, requesting one byte. This is a possible chain of events:

- The Xillybus IP core detects the low "empty" signal, and hence fetches a single byte from the FIFO, after which it becomes empty again.
- The byte is written, with DMA, to the first position in the DMA buffer. The host isn't notified, as the buffer isn't full.
- A read() function call is invoked on the host, requesting one byte.
- The driver has no DMA buffer to take data from: The only DMA buffer containing data (one byte) is only known to the hardware.
- The driver detects that the amount of data it needs is less than a DMA buffer's size, and therefore tells the hardware to hand over a partially filled buffer, if it has at least one byte.
- The driver starts a 10 ms sleep, waiting for something to happen.
- The hardware responds immediately with handing over the partially filled buffer to the host.
- The driver wakes up immediately, copies the requested one byte into the buffer that was supplied with the read() function call, and returns.

This simple example demonstrates how a read() function call returns virtually immediately, even though the data's size was significantly smaller than the DMA buffer.

Let's look at the example again, with one small difference: The read() function call requests two bytes, even though only one is written to the FIFO. The sequence is as follows.

- The Xillybus IP core detects the low “empty” signal, and hence fetches a single byte from the FIFO, after which it becomes empty again.
- The byte is written, with DMA, to the first position in the DMA buffer. The host isn’t notified, as the buffer isn’t full.
- A read() function call is invoked on the host, requested *two* bytes.
- The driver has no DMA buffer to take data from: The only DMA buffer containing data (one byte) is only known to the hardware.
- The driver detects that the amount of data it needs is less than a DMA buffer’s size, and therefore tells the hardware to hand over a partially filled buffer, if it has at least *two bytes*.
- The driver starts a 10 ms sleep, waiting for something to happen.
- The hardware does nothing, as it has only one byte in the DMA buffer, but two were requested.
- The driver wakes after 10 ms, having nothing. It sends a request to the hardware to hand over a partially filled buffer as soon as possible, unless it’s empty.
- The hardware responds immediately with handing over the partially filled buffer to the host.
- The driver wakes immediately, copies the one byte requested into the function caller’s buffer, and returns.

This second example shows the consequence of asking for two bytes when there was actually only one: The function call returns only after 10 ms, with one byte. Note however that this delay is unnoticed in most practical scenarios.

A.3.7 Practical conclusions

- Even if the application-level data always consists of chunks of N bytes, there is no reason to adapt the DMA buffer size in any way. The user application software just needs to make sure to make the read() function calls request data amounts exactly as needed, and the partial buffer mechanism will make sure that the function call returns when the data has been pushed into the FPGA’s FIFO, with a very low latency.

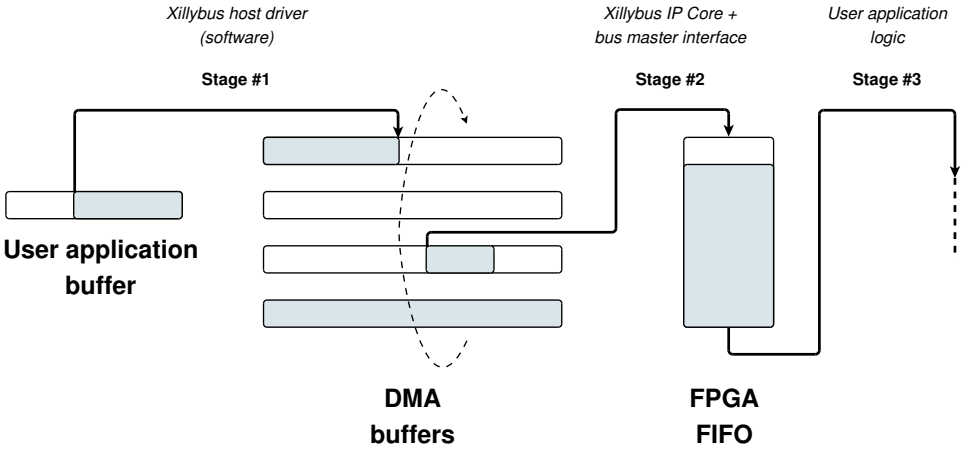
- Even for continuous streams of data, latency can be reduced by making read() function calls with small buffers, at the cost of additional operating system's overhead. Regardless of the DMA buffers' size, the latency depends only on the data rate and the number of bytes that are requested in the read() function calls. Reducing the DMA buffer size won't help, since the read() function call will continue waiting up to 10 ms if it can't fulfill the read() function call completely.
- If 10 ms is an acceptable latency, there is no point in optimizing, as the read() function call is guaranteed to return after this time period, unless there is no data at all to return with.

A.4 Host to FPGA (downstream)

A.4.1 Overview

The figure below depicts the data flow in the host to FPGA (downstream) direction. The shaded areas represent data that is yet to be consumed, in the respective storage elements.

In this example, four DMA buffers are shown, even though the number of these can be configured in the IP Core Factory.



As before, the data flows from the host to the FPGA in three stages, as detailed next.

A.4.2 Stage #1: User software application to DMA buffer

This stage is implemented on Xillybus' driver (on the host) by responding to write() system calls (or the counterpart IRPs on Microsoft Windows). According to the well-established API, the write() function call request includes a buffer that is supplied by the user application, as well as the size of the buffer, which is also the maximal number of bytes to write. The function call may return after writing the maximal number of bytes (complete fulfillment) or less.

A pool of DMA buffers is allocated in the host's RAM memory. The lifecycle of each DMA buffer is like in many similar settings: In the beginning, all DMA buffers are empty and conceptually belong to the host. The host writes data to the buffers in a round-robin manner: When it has finished writing to a certain buffer, it informs the hardware that the buffer is ready for use (the buffer is handed over to the hardware, so to speak), after which it continues to write on the following buffer. The hardware may then consume the data in the buffer, after which it informs the host that the buffer can be written to again (the hardware returns the buffer to the host).

Xillybus' driver responds to write() function calls by attempting to copy as much data as possible into the DMA buffers. When a DMA buffer is filled completely, it's handed over to the hardware, i.e. the host informs the hardware that the buffer can be consumed, and guarantees not to write to it again before the hardware returns the buffer to the host.

If the driver managed to write at least one byte before running out of DMA buffer space, the write() function call returns with the number of bytes written. Otherwise is waits indefinitely (by sleeping, i.e. "blocking"), until a DMA buffer is made available for writing, and then it writes as much data as possible into the DMA buffer and returns.

Note that if a DMA buffer is partially filled, it's *not* handed over to the hardware at the end of the write() function call, so there may be data in one DMA buffer, which the hardware isn't aware of. A "flush" operation hands over a partially filled buffer, and it takes place in any of the following four cases:

- An explicit flush, caused by making a write() function call with zero bytes to write. This write() function call returns immediately (i.e. it doesn't wait for the data to be consumed by the FPGA).
- An automatic flush is initiated 10 ms after the last write() function call.
- When the file is closed, a flush occurs. In this scenario, the close() function call waits for up to one second for the data to be fully consumed by the FPGA before returning.

- On synchronous streams, every function call to write() ends with a flush(), which waits indefinitely until the data is fully consumed by the FPGA.

Note that a write() function call with a zero-length buffer forces an explicit flush, making sure that all data that has been written is available to the FPGA. However it doesn't give the application software an indication on when the data is consumed by the FPGA. If such synchronization is required, a synchronous stream should be used.

A.4.3 Stage #2: DMA buffer to Intermediate FIFO

In this stage, the Xillybus IP core copies the data from the DMA buffers in the host's RAM to the FIFO in the FPGA. To accomplish this, the core uses some bus master interface (PCIe, AXI4 etc) to read data directly from the host's memory, without the intervention of the host's processor.

The data flow of stage #2's is controlled by the FIFO's "full" signal and by the availability of data in the pool of DMA buffers belonging to the FPGA. When the Xillybus IP core senses a low "full" signal from the FIFO, and there is data ready in some DMA buffer, it fetches data from the DMA buffer and writes it into the FIFO. When the FIFO becomes full again, or the DMA buffers are empty, the IP core's internal state machine stops fetching data momentarily, and then continues from where it left off in the DMA buffer pool.

While the data flow is stalled, the IP core might be busy with other activities, for example copying data on some other stream's behalf (i.e. filling another intermediate FIFO). As a result, there might be a random delay between the time that the FIFO changes the "full" signal to low, and the resumption of data copying. This delay varies, but overall, the IP core guarantees that a FIFO of 512 words is deep enough.

The hardware is of course aware of partially filled DMA buffers, and keeps track of how much data each one contains.

A.4.4 Stage #3: Intermediate FIFO to application logic

The user application logic in the FPGA fetches data from the FIFO that connects between the user application logic and the Xillybus IP core. There is no requirement on when or how much data is fetched, except for respecting the FIFO's "empty" signal to avoid underflow.

A.4.5 An example

Let's consider the following simple case of an 8-bit asynchronous stream. Suppose that a stream starts with not containing any data, after which the host's application writes a single byte to the device file.

The sequence of events is as follows:

- The driver's write() function call is invoked with a request to write one byte.
- As the stream contains no data, clearly there's space in the DMA buffers. Hence the driver copies the byte into the first DMA buffer and returns.
- Nothing happens during 10 ms.
- The autoflush mechanism is triggered after 10 ms, causing the driver to hand over the DMA buffer to the hardware with the information that it contains one byte.
- The Xillybus IP core reads the byte from the DMA buffer and writes it into the intermediate FIFO.
- The application logic may read the byte from the FIFO at will.

A.4.6 Practical conclusions

- Even if the application-level data always consists of chunks of N bytes, there is no reason to adapt the DMA buffer size in any way. The user application software just needs to request a flush of the data, with a write() function call that requests zero bytes, at the end of each chunk. A latency, which is in the order of magnitude of microseconds, is achieved this way.
- Even for continuous streams of data, latency can be reduced by making write() function calls with small buffers, followed by a flush (write() function call with zero bytes), at the cost of additional operating system's overhead. Regardless of the DMA buffers' size, the latency depends only on the data rate and the amount of data between the flush requests.
- It can make sense to reduce the DMA buffer size if it's know in advance that a flush always occurs after a given chunk of data, and hence no DMA buffer is ever filled beyond a certain level. However the only advantage of doing so is saving an amount of RAM at the host, which is unlikely to be significant.

- If 10 ms is an acceptable latency, there is no point in optimizing, as the auto-flushing mechanism kicks in after 10 ms of no activity.