Xillybus host application programming guide for Windows

Xillybus Ltd.

www.xillybus.com

Version 2.3

1 Introduction

Xillybus was designed to present the Windows host with a simple and well-known interface, having a natural and expected behavior. The host driver generates system object which behave like named pipes, and declare themselves as such. 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 objects that are accessed like just any file, 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 it's loaded into the host's operating system, and device files are created accordingly. These device files are accessed as `\\.\xillybus_something`.

Also at driver load, DMA buffers are allocated in the host's memory space, and the FPGA is informed about their addresses. The number of DMA buffers and their size are separate parameters for each stream. These parameters are hardcoded in the FPGA IP core for a given configuration, and are retrieved by the host during the discovery process.

A handshake protocol between the FPGA and host makes an illusion of a continuous data stream. Behind the scenes, DMA buffers are filled, handed over and acknowledged. Techniques similar to those used for TCP/IP streaming are used to ensure an efficient utilization of the DMA buffers, while maintaining responsiveness for small pieces of data.

2

Synchronous 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 level software 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 host program's actions 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 synchronous or asynchronous flag for each stream (among other attributes).

In all demo bundles, the xillybus_read_* and xillybus_write_* streams are asynchronous. xillybus_mem_8 is seekable and therefore synchronous.

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 preemption periods 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 orientated applications (e.g. sound applications and video players) have no definite solution to this problem. Instead, they rely on a certain de-facto behavior of the operating system, and make up for the preemption periods with I/O buffering.

Asynchronous streams address this issue by allowing a "background" flow of data 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 FPGA to host streams

In the upstream direction (FPGA to host), an asynchronous stream fills the host's DMA buffers whenever possible. That is, when the file is open, data is available and there is free space in the DMA buffers.

On the other hand, if the stream is synchronous, the IP core's logic in the FPGA will not fetch data from the user application logic unless the user application has an outstanding request to read that data from the file descriptor.

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.
- The FIFO between the application logic and the IP core's logic in the FPGA may overflow during these time gaps. For example, if its fill rate is 100 MB/sec, a typical 2 kByte FPGA FIFO goes from empty to full in around 0.02 ms. Practically, this means that any preemption of the user space program will make the FPGA's FIFO overflow.

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 FPGA IP core is available for reading immediately by the host's user space application, regardless of whether the stream is synchronous or asynchronous. There's a mechanism handling partially filled DMA buffers, making this issue transparent to the user space application, thus avoiding unnecessary blocking.

2.4 Host to FPGA streams

In the downstream direction (host to FPGA), the stream being asynchronous means that the host application's 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 DMA 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 application.

For asynchronous streams to the FPGA, the 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)
- The file descriptor is explicitly flushed (see paragraph 3.4)
- The file descriptor is 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).

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, the return of this low-level function indicates that the data has arrived to the FIFO in the FPGA, which is connected by the user to the IP core.

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 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 segments. In most cases, this leads to a significant bandwidth performance hit.

- The FIFO between application logic and IP core's logic in the FPGA may underflow during these time gaps. For example, if its drain rate is 100 MB/sec, a typical 2 kByte FPGA FIFO goes from full to empty in around 0.02 ms. Practically, this means that any preemption of the user space program will make the FPGA's FIFO underflow.

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 Latency vs. uncertainty

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 buffering, is made as small as possible, leading to an overall system with tough realtime 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 close loops, if there is such a need.

For a modem, the maximal latency has an impact on how quickly the underlying data source responds to data 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 arriving with 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 mixing timing 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 file access API is suitable.

In this guide there's an emphasis on the low-level classic C run-time API set, based upon functions such as _open(), _read(), _write() and _close(). These underscore-prefixed functions behave exactly like their non-underscored counterparts, (i.e. _read() and read() is the same function) but some Microsoft compilers issue a warning when the non-underscored functions are used.

The Microsoft API (CreateFile(), ReadFile() etc.) can be used as well (see paragraph 3.5), but has been mostly ignored here because

- the extra features given by the Windows API are not necessary (even though the error reporting supplied by Windows' API is more complete).
- the classic C API is well known, simpler, and more easily adopted by programmers.
- the classic C API is less likely to be modified across different versions of Windows.
- the classic C API is portable.

The low-level 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 they also detach the timing and amounts of actual I/O operations from the function calls issued by the program.

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

An extra buffer layer can also cause confusion, making it look like there's a software bug where there isn't. For example, a call to fwrite() can merely store the data in a RAM buffer without performing any I/O operation until the file is either closed or fflush() is called. 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 in 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 Windows host.

The typical header files for these examples are

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

3.2 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("\\\\.\\xillybus_ourdevice", O_RDONLY | _O_BINARY);

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

The `\\\\.\\` prefix in the file name turns into `\\.\` after the backslashes are unescaped.

Note the `_O_BINARY` flag, which tells Windows to treat the stream as non-text data. In the absence of this flag, Windows will convert newline characters and treat CTRL-Z (0x1a) as EOF (end of file).

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);
```

where `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 call completed abnormally).

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 call _read() again if the completed amount was unsatisfactory.

The 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 an interrupt. This is a result of the process receiving a signal from the operating system.

This is not an error really, but a condition forcing the I/O driver to return control to the application immediately. The use of the EINTR error number is just a way to tell the caller that there was no data read. The program responds with a “continue” statement, resulting in a renewed attempt to call _read() with the same parameters.

If there is some data in the buffer when the interrupt arrives, the driver will return the number of bytes already read in `rc`. The application will not know an interrupt 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), either the operating system or a registered signal handler will respond as necessary.

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

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

Despite all said about signal handling above, these signals are UNIX oriented, and it’s not clear if they will arrive at all on a Windows computer. Either way, the related if-statement is useless but harmless at worst.

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 zero read byte count. When reading from a Xillybus device file, this can only happen as a result of the application logic raising the stream's _eof pin on the IP core's interface.

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("\\\\.\\xillybus_ourdevice", O_WRONLY | _O_BINARY);

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

The `\\\\.\\` prefix in the file name turns into `\\.\` after the backslashes are unescaped.

Note the `_O_BINARY` flag, which tells Windows to treat the stream as non-text data. In the absence of this flag, Windows will convert newline characters and treat CTRL-Z (0x1a) as EOF (end of file).

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 write-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);
```

where `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 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 call _write() again if the completed amount was unsatisfactory.

The 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 an interrupt. This is a result of the process receiving a signal from the operating system.

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

If some data was written before the interrupt arrived, the driver will return the number of bytes already written in `rc`. The application will not know an interrupt 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), either the operating system or a registered signal handler will respond as necessary.

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

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

Despite all said about signal handling above, these signals are UNIX oriented, and it's not clear if they will arrive at all on a Windows computer. Either way, the related if-statement is useless but harmless at worst.

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 the a zero write byte count. When writing to a Xillybus device file, this should never happen

3.4 Flushing asynchronous downstreams

As mentioned in paragraph 2.4, data written to an asynchronous stream 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 PCI Express bus.

Streams to the FPGA are automatically flushed when the file descriptor is closed. The call to _close() returns immediately, but the driver attempts to flush any remaining data in the background for up to a second. If the flush isn't completed by then, a warning message is issued in the event log.

It's also possible to flush an asynchronous stream explicitly, by calling _write() with a zero-length buffer, 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:

- Unlike close(), a zero-count write() returns immediately, regardless of when the data is consumed on the FPGA.
- Since no data is read from the buffer, the buffer argument in the _write() call can take any value, including NULL, as demonstrated above.
- There is no clear system definition for what _write() with a zero count should do, leaving the choice to each device driver. This method for flushing is Xillybus-specific, and there is also a chance that certain Windows implementations will ignore a zero count _write(), even though tests made so far indicate that this method works.
- Using higher-level API with a zero buffer may not have any effect at all. For example, calling 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 flushes the higher-level buffer, but doesn't send a flush command to the low-level driver.
- There is no need to flush streams in the other direction (from FPGA to host) since these streams are automatically flushed when a host's attempt to read data is about to block.

3.5 Using Windows' native API

Although not recommended for use, an example of reading from a stream with Windows' native API is given for the sake of completeness. The full code is available as winstreamread.c in the demo application bundle (see Getting started with Xillybus on a Windows host).

First, let's define a helper function for printing out errors. It's the Windows counterpart for perror(), accepting a string for a description of the attempted action and the error code.

The function prints the given string, the error code and the human-readable error description as translated by Windows.

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

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


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

  LocalFree(lpMsgBuf);
}
```

The following variable declarations are assumed:

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

A file is opened with a call to CreateFile(), which despite its name doesn't necessarily create a file, but opens it.

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

CreateFile() always opens the file in “binary mode”. As a matter of fact, it doesn't support the LF conversion known as “text mode”.

The `\\\\.\\` prefix in the file name turns into `\\.\` after the backslashes are unescaped.

The success of opening the file is checked, and errorprint() defined above is called in case of a failure.

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

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

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

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

`rc`, which is written to by ReadFile(), contains the number of bytes actually read. If it's zero, the end of file has been reached. When reading from a Xillybus device file, this can only happen as a result of the application logic raising the stream's _eof pin on the IP core's interface.

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

IMPORTANT:

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

4

Continuous high rate I/O

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 DMA buffers are large enough to compensate for time gaps between the user space application's I/O operations
- 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 buffering space available (depending on the direction).
- Never closing and reopening the device files while the FPGA keeps pumping or draining data.

The first practice 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 user space application's intervention by virtue of DMA. This flow is stopped when the file is closed.

On a stream to the FPGA, the data is flushed and then the file closed, so the FIFO on the FPGA will be drained until the file is reopened and written to again.

As for streams from the FPGA, closing the file leads to loss of any data in the pipe going from the application logic in the FPGA to the user space application in the host (i.e. the FPGA's FIFO and DMA buffers on the host). The only way to avoid this loss is draining this pipe of data, and thus losing the buffering capabilities it has.

A common mistake is to use the EOF capability to mark data chunks (e.g. complete video frames) by forcing the host to close and reopen the device file at known boundaries, but this significanly increases the risk for data overflows at the FPGA.

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

4.2 Large DMA 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. The common way to avoid this is by allocating large RAM buffers for DMA transfers on the host. These buffers compensate for the gaps in time, during which the application isn't available to handle data transfers.

Xillybus allows allocation of huge DMA buffers, but this memory must be allocated from the pool of the operating system's kernel RAM. Typically, the addressing space of such memory is limited to 1 GB by the Windows operating system, even if the total RAM available is significantly larger. In systems with a total of less than 1 GB RAM, all memory may be used for DMA buffers.

The DMA 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 DMA 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 that since the DMA buffers are seen directly by hardware, they must occupy continuous segments of physical RAM. This is contrary to a buffer allocated in a userspace program, which is continuous in virtual address space, but can be spread all over the physical address space, 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 the not actively used. Attempting to unload the driver and reload it at a later stage may fail for the same reason.

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, i.e. 512 MB, 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 User space RAM buffer

For applications that require buffers larger than 512 MB, it's recommended to do some of the buffering in user space RAM.

It may seem counterintuitive that the problem of I/O continuity can be solved by allocating a huge buffer in the userspace application. Indeed, this solution doesn't help when the OS starves the application of CPU time. But if the 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 loaded computer.

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 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: VirtualLock() 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 call, it may take some time to return.

The operating system is reluctant to lock large chunks of RAM, as this impacts the overall OS' performance. Calling SetProcessWorkingSetSize() to raise the RAM locking limit is often necessary for VirtualLock() to succeed.

4.4 Why not just Windows pipes?

The Windows standard API supports (at least) two functions for creating pipes: CreatePipe() and CreateNamedPipe(). These two functions allow the caller to determine the size of the buffer, so on the face of it, they could do the job. Unfortunately, Windows considers the requested buffer size as a suggestion, as stated in the web pages for CreatePipe() and CreateNamedPipe().

4.5 The fifo.c demo application overview

Among the demo applications, which can be downloaded for Linux and Windows, there's one called "fifo.c" (for more about the demo applications, see Getting started with Xillybus on a Windows host). It's an example of how to implement a RAM FIFO using two threads, and has been compiled and tested as 32-bit and 64-bit executables.

The purpose of this program is to test fast streams, where a RAM FIFO is necessary to maintain a buffer larger than the 512 MB, that can be allocated as DMA buffers.

It can also form a basis for modification and adoption in custom target applications. It's designed with no mutexes, so no thread ever goes to sleep just because another thread holds the lock. Threads will sleep, of course, when the FIFO's state requires blocking (e.g. a read is requested from an empty FIFO). This mutex-less design requires careful use of the API functions, as they're not reentrant. This is however no issue in a simple writer-reader thread setting.

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

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

or in Windows:

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

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

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 main application thread.

4.6 fifo.c hacking 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 in separate threads when each thread has its exclusive set of functions (which is a natural use).
- fifo_init() can take time to return, and should be called before an asynchronous Xillybus file stream is opened.
- The read and write threads 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 sink is /dev/null. Both will complete the entire request in one go, 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 byte count in calls to I/O functions.

4.7 RAM FIFO functions

Except for hacking 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, 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 workflow: The thread reading from the FIFO calls 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 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 writing user application thread was blocking because the FIFO was full, it is woken up.

Note that the user application 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 writing thread calls fifo_request_write(), which returns the number of bytes that can be written to the FIFO, or blocks if the FIFO is full. The user application writes to the address it got from fifo_request_write() as many bytes it needs to write (but not more than fifo_request_write() allowed it to) and then reports back what it did to fifo_wrote().

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

4.7.1 fifo_init()

fifo_init(struct xillyfifo *fifo, unsigned int size) – This function initializes the FIFO 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 two) 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 many megabytes of data written to disk before returning.

Returns zero on success, nonzero otherwise.

4.7.2 fifo_destroy()

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 Windows, their API doesn't guarantee this.

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

4.7.3 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 used for calls to fifo_request_write(). A thread local variable is the straightforward choice.

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, so a significantly lower byte count is possible as the pointer approaches its wrap to the beginning 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, `size` as given in fifo_init(). A nonzero fifo->slept indicates that the FIFO was empty upon invocation.

Returns the number of bytes allowed for write (same as info->taken), or zero if fifo_done() has been called and the FIFO is empty.

4.7.4 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 blocking because the FIFO was full, it will we 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 call to fifo_request_drain().*

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

4.7.5 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 used for calls to fifo_request_drain(). A thread local variable is the straightforward choice.

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, so a significantly lower byte count is possible as the pointer approaches its wrap to the beginning 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, `size` as given in fifo_init(). A nonzero fifo->slept indicates that the FIFO was full upon invocation.

Returns the number of bytes allowed for write (same as info->taken), or zero if fifo_done() has been called, even if the FIFO is not full: There is no point writing data into a FIFO that will never be read.

4.7.6 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 blocking because the FIFO was empty, it will we 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 call to fifo_request_write().*

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

4.7.7 fifo_done()

This function is optional for use, and helps the application to quit gracefully if either of the read or write threads 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 blocking if the FIFO is empty, and fifo_request_write() will return zero regardless.

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

4.7.8 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, buggy I/O drivers may clutter the bytes sharing the 32-bit boundary of the last byte written to. Just to have this clear, the Xillybus drivers do **not** have such bug.

To avoid this rare but possible problem, 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.

5

Cyclic frame buffers

5.1 Introduction

In applications such as video camera image frame grabbers or raw video playback, it's desirable to manage a number of frame buffers with a fixed size. The advantage of this setting is that frames can be skipped or replayed more than once if the data flow becomes jammed on the other side.

In a frame grabbing application, some input images can be dropped if the data sink momentarily stops receiving data. In a live view application, this can be the case when the viewing window is moved or resized. Dropping the overflowing images prevents the disruption of the continuous data flow from the video source, while maintaining a small latency from source to sink.

In a frame replay application (e.g. driving a live output screen), any output image is repeated until a fresh one arrives. 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 bulky, works well for overcoming frame rate differences, in particular when the output frame rate is considerably higher than the input frame rate (e.g. 30 fps to 60 fps).

5.2 Adapting the FIFO example code

There are similarities between maintaining a circular 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 grabbing 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 software 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 software 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 4.7 can be adopted as is:

- Call 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 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 block until there is.

- After reading from the buffer, call fifo_drained() with bytes_req=1.

- 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 to backoff with frame buffers.

- If the request_* routines return more than 1, skipping buffers may be desirable. This isn't a sufficient solution for the other end being temporarily jammed, as explained next.

5.3 Frame dropping and repetition

Let's take the case of a continuous image frame source which must never overflow, but with an output sink which may not collect the data all the time.

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 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 fifo_request_write() again. This call will surely not 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 fifo_wrote() (with req_bytes=1, of course). A subsequent call to fifo_request_write() will surely not block, because there were more than one buffer to spare, and only one was consumed. In fact, the next 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 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 event, as well as its initialization and destruction code. Leaving them in the code has a minimal effect, so this optimization is better done at a late stage.

A similar approach can be taken to repeat frames on the thread writing from the FIFO: Call fifo_request_drain() again just before calling 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 (in FPGA logic) to be seekable. The stream's position is presented to the 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 FPGA 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 integers 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 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 call to _lseek(), which sets the address. Only the SEEK_SET call should be used.

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

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.

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 _lseek() again.

Seekable reading works in the same manner. See memwrite.c and memread.c in the demo application bundle (and their descriptions in Getting started with Xillybus on a

Windows 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 samples from an A/D converter connected to an RF receiver and sending samples to an D/A converter, connected to an RF transmitter. In cases like this, it's often needed to produce the samples for transmission so their actual air transmission takes place at a defined time. Likewise, it may be significant to know the timing of a received signal.

Luckily, this is quite simple to implement with simple FPGA logic. One such solution is to drop reception 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 dropping receptions 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 host application can therefore match the timing of any sample for transmission with any sample received just by matching their position in the respective streams. A slight correction may be needed to compensate for latencies in the FPGA and analog conversion, but these latencies are known given the electronics.

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 timing relationship between transmission and reception is enough. When the timing needs to be synchronized with an external event, the FPGA can refrain from reading data from the host until that event has occurred. The principle of dropping received samples may apply, depending on the application.

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 grabbing or replaying application.

For the case of varying length packets, let's look at an upstream application, where the FPGA sends packets to the host. Let's also assume that the FPGA receives the packets as a byte stream, with a signal on the first and last byte. In other words, the FPGA knows the length of the packet only when the last byte arrives.

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

- The FPGA writes every byte it receives to the first Xillybus stream, from the one marked as the first one to the one marked as last.
- The FPGA resets a byte counter when a byte marked as first arrives, and counts the other bytes arriving.
- When a byte marked as last arrives, the FPGA sends the counter's value on the second Xillybus stream.

An important quality 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.

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.

This arrangement is also extensible to passing other metadata in the non-data stream, e.g. the packet's routing (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 blocks (sleeps) 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 main application versus interrupt routine setting, this dedicated stream can be read by a separate software thread or process. With this arrangement, the main code flows regardless of the thread reading 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 Coprocessing / Hardware acceleration

Coprocessing (also known as *hardware accelerated*) applications 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 the coprocessing an eligible solution.

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

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 fabric 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.

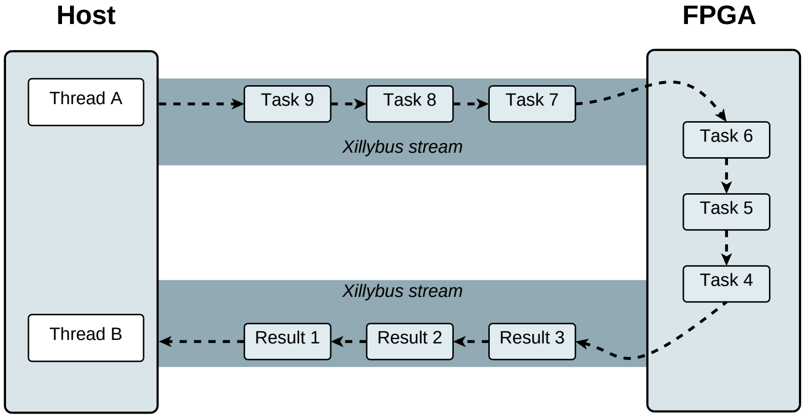
In order to overcome the latencies imposed by the bus and the processing logic itself, there's a need to reorganize the software. In particular, the tasks in a single-threaded program need to be split into two or more threads (or processes). If multiple threads are not possible or desirable, certain 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 *pipelined* technique minimizes the hardware's latencies, since neither of the threads is effectively waiting the latency time. Instead, the latency influences the minimal gap of processing items between the both 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 acceleration of sqrt() is a relatively simple example, but it covers much of the challenge in utilizing coprocessing techniques. Almost always, large parts of the computer program needs to be rewritten so that the everything is driven by the pipelined data flow.

Another issue to be aware of, is that since Xillybus is based upon read() and write() calls, 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 sqrt() acceleration is a good example for this: A double float is typically 8 bytes long. I/O system calls of this length are quite inefficient, so concatenating several double floats for a single I/O call will make a difference.

It's also worth to mention, that not all applications involve data chunks of constant lenghts. For example, using FPGA coprocessing for obtaining 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.

7 Hibernation

“Hibernation”, “standby” and “sleep” are common names for a computer’s power saving state. Upon hibernation, the computer saves the contents of its RAM to a hard disk, and effectively powers down. When commanded to power up again, it quickly resumes to apparently the same state as it was upon entering hibernation.

Hibernation with an FPGA-based peripheral requires extra caution, because the device driver doesn’t know what the FPGA is about to go through.

If it’s powered from an external source, it will stay powered on while the rest of the computer goes down. But it’s not clear if the PCIe bus itself will remain powered on, or if it will be reset at some stage.

If it takes its power source from the computer’s supply, it will most likely lose power. In that case, it may reconfigure from a flash when power returns, and continue being a PCIe card on the bus, or it can disappear completely from the bus if it depends on manual configuration.

Note that Windows can fail to wake up from hibernation if a PCIe interface suddenly goes away while the computer was down. So FPGAs that depend on the computer’s power source should definitely configure themselves automatically when powered on.

The Xillybus device driver copes with this uncertainty by acting as if the driver is about to be unloaded completely, when the operating system informs it that the computer is about to hibernate: It generates an error on all outstanding I/O requests, forces all open files to close immediately, frees all resources, removes the device files and brings the Xillybus IP core in the FPGA to a quiescent state.

When the computer resumes from hibernation, the driver initializes as in bootup.

Despite the proven reliability of this process, there are two possible problems:

- The driver may not be able to allocate enough RAM for DMA buffers, in particular if the overall DMA buffer size exceeds several megabytes. Large chunks of continuous memory in physical mapping are by far easier to allocate when the driver is loaded at computer boot than after the computer has been running for a while. The fragmented memory map is retained, so the chunks of continuous physical memory that were freed before hibernation may not exist on resumption.
- Even though user space applications should be ready to handle errors from any I/O function call, an unexpected error resulting from hibernation may expose a rare hidden bug.

The overall conclusion is that the best way to handle hibernation is to avoid it on a computer with an FPGA-based peripheral attached to the PCIe bus. This is easily done by setting the computer's power saving attributes, in particular turning off automatic hibernation resulting from what the operating system may consider idling. This will also prevent the computer from going to sleep due to lack of human interaction, while it was in fact in the middle of an important unattended task.

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. The goal of this design is to make the underlying buffers transparent to the user, and to a large extent they are. Please keep this in mind when going down to the technical details below, as they are very likely to be unnecessary for using Xillybus. 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 where comments on synchronous streams are given explicitly. The end-of-file signal as well as the option for non-blocking I/O are not covered either.

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, but 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.

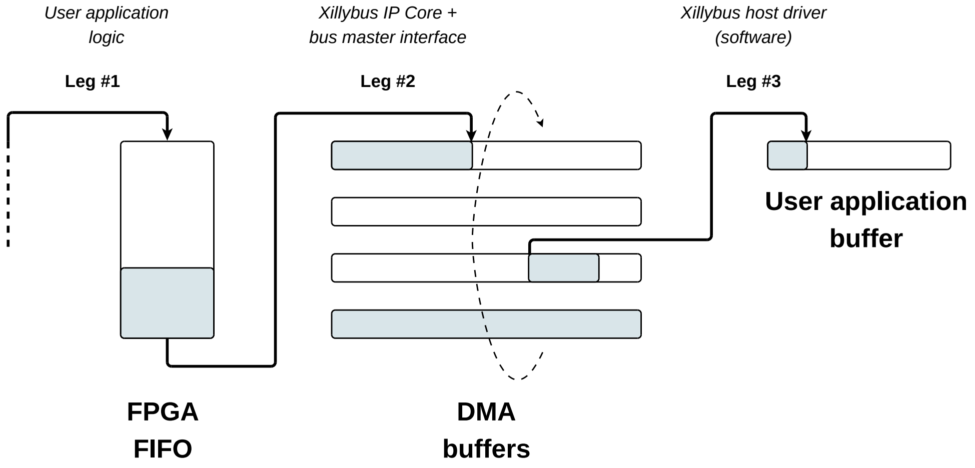
However the techniques described below are employed to create an illusion of a continuous stream, so that the user can ignore the existence of the underlying data transport. In particular, even if the application consists of sending data in fixed-sized chunks, there is no need to match the DMA buffers'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

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



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

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

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

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

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

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 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"), 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").

Leg #2's data flow 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 deasserted FIFO's "empty" signal, and there's 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 deassertion of the "empty" signal by the FIFO and the resumption of fetching data from it. This delay varies, but the overall design guarantees that a FIFO of 512 words will not 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 Leg #3: DMA buffer to user software application

This leg is implemented on Xillybus' driver on the host by responding to read() system calls (or the counterpart IRPs on Windows). According to the well-established API, the read() 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 read. The 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, determining whether there is enough unconsumed data in the DMA buffers for a complete fulfillment, and if so, it copies the data to the user buffer, possibly returning DMA buffers to the hardware, and returns.

Otherwise, the API for a read() 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() 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 buffer(s) than required by the read() 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 possibly forever if no data is available, per API). This results in a fairly responsive return time, but limits the overhead to 100 read() calls per second, if the read() caller requests more than the data available all the time.

This is not to say that there is necessarily a 10 ms latency on read() calls: If the user space application knows in advance how many bytes should be ready, it may request no more than that number, and ensure a latency measured in 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() call completely, if the partially filled DMA buffer is counted in.

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 many data elements would be enough. This gives the hardware a chance to send a partially filled buffer, if it indeed allows completing the read() call entirely. The driver then starts the 10 ms wait.

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() 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 so 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() call is invoked, since the hardware isn't allowed to copy data from the FPGA's FIFO unless instructed to. Accordingly, the read() 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() call, which happens in either of three conditions:

- The host is currently handling a read() call, which will be fulfilled completely when the current partially filled buffer is handed over.
- A read() call stands at zero bytes, and has reached the time limit (typically 10 ms).
- On synchronous streams only: When the hardware has completed fetching the amount requested by the host.

Note that the FIFO becoming empty is *not* a reason for a DMA buffer submission by itself.

A.3.6 Examples

Let's consider the following simple case of an 8-bit asynchronous stream. Suppose that a stream starts from fresh, after which the FIFO is filled with a single element (that is, one byte). The application program on the host then calls read() 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 (nearly) full.
- A read() 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 immediately, copies the one byte requested into the read() call buffer, and returns.

This simple example demonstrates how a read() 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() 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 (nearly) full.
- A read() 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 caller's buffer, and returns.

This second example shows the consequence of asking for two bytes when there was actually only one: The call returns only after 10 ms, with the same one byte. Note however that the read() call is still fairly responsive.

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() calls request data amounts exactly as needed, and the partial buffer mechanism will make sure that the call returns when the data has been pushed into the FPGA's FIFO, with a latency measured in microseconds.
- Even for continuous streams of data, latency can be reduced by making read() 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 byte count size given on the read() calls. Reducing the DMA buffer size won't help, since the read() call will continue waiting up to 10 ms if it can't fulfill the read() call completely.

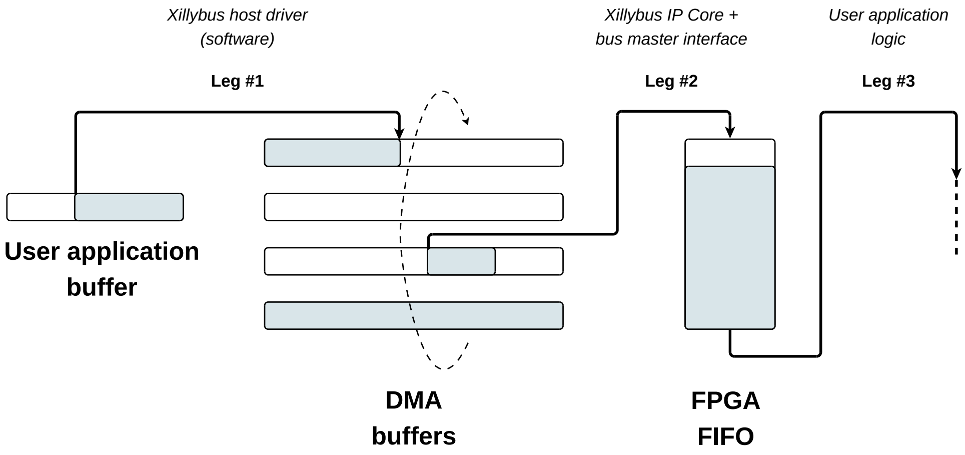
- If 10 ms is an acceptable latency, there is no point in optimizing, as the read() call is guaranteed to return after this time period, unless there is no data at all to return with.

A.4 Host to FPGA (downstream)

A.4.1 Overview

The figure below depicts the flow in the host to FPGA (downstream) direction. The shaded ares represent unconsumed data in the respective storage elements.

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



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

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

This leg is implemented on Xillybus' driver on the host by responding to write() system calls (or the counterpart IRPs on Windows). According to the well-established API, the write() 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 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”), 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 host “returns the buffer to the host”).

Xillybus’ driver responds to write() calls by attempting to copy as much data as possible into the DMA buffers. When any 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() call returns with the number of bytes written. Otherwise is waits (”blocks” by sleeping), possibly indefinitely, until a DMA buffer is made available for writing, at which 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() 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() call with the required byte count set to zero. The write() 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() call.
- When the file is closed, a flush occurs, and the close() call waits for up to one second for the data to be consumed by the FPGA before returning.
- On synchronous streams, every call to write() ends with a flush(), which waits indefinitely until the data is consumed by the FPGA.

Note that a zero-count write() call 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 applied.

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

In this leg, the Xillybus IP core copies the data from the DMA buffers in the host's RAM space 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 host's processor's intervention.

Leg #2's data flow 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 deasserted FIFO's "full" signal, and there's 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 exhausted, 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 deassertion of the "full" signal by the FIFO and the resumption of data copying. This delay varies, but the overall design 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 is filled with.

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

The user application logic in the FPGA fetches data elements from the FIFO connecting between the user application logic and the Xillybus IP core. There is no requirement on when or how much data is fetched, except 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 from fresh, after which the host application writes a single byte to the device file.

- The driver's write() call is invoked with a request to write one byte.
- As the stream is fresh, clearly there's space in the DMA buffers. Hence the driver copies the byte into the first DMA buffer and returns.

- Nothing happens for 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 flush the data with a zero-count write at the end of each chunk to obtain a latency measured in microseconds.
- Even for continuous streams of data, latency can be reduced by making write() calls with small buffers, followed by a zero-count write() flush, 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 byte count size given on the write() calls.
- 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 some 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 autoflushing mechanism kicks in after 10 ms of idling.