Xillybus driver for generic FPGA interface

Author: Eli Billauer, Xillybus Ltd. (http://xillybus.com)
Email: eli.billauer@gmail.com or as advertised on Xillybus' site.

Introduction

Background

An FPGA (Field Programmable Gate Array) is a piece of logic hardware, which can be programmed to become virtually anything that is usually found as a dedicated chipset: For instance, a display adapter, network interface card, or even a processor with its peripherals. FPGAs are the LEGO of hardware: Based upon certain building blocks, you make your own toys the way you like them. It's usually pointless to reimplement something that is already available on the market as a chipset, so FPGAs are mostly used when some special functionality is needed, and the production volume is relatively low (hence not justifying the development of an ASIC).

The challenge with FPGAs is that everything is implemented at a very low level, even lower than assembly language. In order to allow FPGA designers to focus on their specific project, and not reinvent the wheel over and over again, pre-designed building blocks, IP cores, are often used. These are the FPGA parallels of library functions. IP cores may implement certain mathematical functions, a functional unit (e.g. a USB interface), an entire processor (e.g. ARM) or anything that might come handy. Think of them as a building block, with electrical wires dangling on the sides for connection to other blocks.

One of the daunting tasks in FPGA design is communicating with a fullblown operating system (actually, with the processor running it): Implementing the low-level bus protocol and the somewhat higher-level interface with the host (registers, interrupts, DMA etc.) is a project in itself. When the FPGA's function is a well-known one (e.g. a video adapter card, or a NIC), it can make sense to design the FPGA's interface logic specifically for the project. A special driver is then written to present the FPGA as a well-known interface to the kernel and/or user space. In that case, there is no reason to treat the FPGA differently than any device on the bus.

It's however common that the desired data communication doesn't fit any well- known peripheral function. Also, the effort of designing an elegant abstraction for the data exchange is often considered too big. In those cases, a quicker and possibly less elegant solution is sought: The driver is effectively written as a user space program, leaving the kernel space part with just elementary data transport. This still requires designing some interface logic for the FPGA, and write a simple ad-hoc driver for the kernel.

Xillybus Overview

Xillybus is an IP core and a Linux driver. Together, they form a kit for elementary data transport between an FPGA and the host, providing pipe-like data streams with a straightforward user interface. It's intended as a low- effort solution for mixed FPGA-host projects, for which it makes sense to have the project-specific part of the driver running in a user-space program.

Since the communication requirements may vary significantly from one FPGA project to another (the number of data pipes needed in each direction and their attributes), there isn't one specific chunk of logic being the Xillybus IP core. Rather, the IP core is configured and built based upon a specification given by its end user.

Xillybus presents independent data streams, which resemble pipes or TCP/IP communication to the user. At the host side, a character device file is used just like any pipe file. On the FPGA side, hardware FIFOs are used to stream the data. This is contrary to a common method of communicating through fixed- sized buffers (even though such buffers are used by Xillybus under the hood). There may be more than a hundred of these streams on a single IP core, but also no more than one, depending on the configuration.

In order to ease the deployment of the Xillybus IP core, it contains a simple data structure which completely defines the core's configuration. The Linux driver fetches this data structure during its initialization process, and sets up the DMA buffers and character devices accordingly. As a result, a single driver is used to work out of the box with any Xillybus IP core.

The data structure just mentioned should not be confused with PCI's configuration space or the Flattened Device Tree.

Usage

User interface

On the host, all interface with Xillybus is done through /dev/xillybus_* device files, which are generated automatically as the drivers loads. The names of these files depend on the IP core that is loaded in the FPGA (see Probing below). To communicate with the FPGA, open the device file that corresponds to the hardware FIFO you want to send data or receive data from, and use plain write() or read() calls, just like with a regular pipe. In particular, it makes perfect sense to go:

```
$ cat mydata > /dev/xillybus_thisfifo

$ cat /dev/xillybus_thatfifo > hisdata
```

possibly pressing CTRL-C as some stage, even though the xillybus_* pipes have the capability to send an EOF (but may not use it).

The driver and hardware are designed to behave sensibly as pipes, including:

- Supporting non-blocking I/O (by setting O_NONBLOCK on open()).
- Supporting poll() and select().
- Being bandwidth efficient under load (using DMA) but also handle small pieces of data sent across (like TCP/IP) by autoflushing.

A device file can be read only, write only or bidirectional. Bidirectional device files are treated like two independent pipes (except for sharing a "channel" structure in the implementation code).

Synchronization

Xillybus pipes are configured (on the IP core) to be either synchronous or asynchronous. For a synchronous pipe, write() returns successfully only after some data has been submitted and acknowledged by the FPGA. This slows down bulk data transfers, and is nearly impossible for use with streams that require data at a constant rate: There is no data transmitted to the FPGA between write() calls, in particular when the process loses the CPU.

When a pipe is configured asynchronous, write() returns if there was enough room in the buffers to store any of the data in the buffers.

For FPGA to host pipes, asynchronous pipes allow data transfer from the FPGA as soon as the respective device file is opened, regardless of if the data has been requested by a read() call. On synchronous pipes, only the amount of data requested by a read() call is transmitted.

In summary, for synchronous pipes, data between the host and FPGA is transmitted only to satisfy the read() or write() call currently handled by the driver, and those calls wait for the transmission to complete before returning.

Note that the synchronization attribute has nothing to do with the possibility that read() or write() completes less bytes than requested. There is a separate configuration flag ("allowpartial") that determines whether such a partial completion is allowed.

Seekable pipes

A synchronous pipe can be configured to have the stream's position exposed to the user logic at the FPGA. Such a pipe is also seekable on the host API. With this feature, a memory or register interface can be attached on the FPGA side to the seekable stream. Reading or writing to a certain address in the attached memory is done by seeking to the desired address, and calling read() or write() as required.

Internals

Source code organization

The Xillybus driver consists of a core module, xillybus_core.c, and modules that depend on the specific bus interface (xillybus_of.c and xillybus_pcie.c).

The bus specific modules are those probed when a suitable device is found by the kernel. Since the DMA mapping and synchronization functions, which are bus dependent by their nature, are used by the core module, a xilly_endpoint_hardware structure is passed to the core module on initialization. This structure is populated with pointers to wrapper functions which execute the DMA-related operations on the bus.

Pipe attributes

Each pipe has a number of attributes which are set when the FPGA component (IP core) is built. They are fetched from the IDT (the data structure which defines the core's configuration, see Probing below) by xilly_setupchannels() in xillybus_core.c as follows:

- is_writebuf: The pipe's direction. A non-zero value means it's an FPGA to host pipe (the FPGA "writes").
- channelnum: The pipe's identification number in communication between the host and FPGA.
- format: The underlying data width. See Data Granularity below.
- allowpartial: A non-zero value means that a read() or write() (whichever applies) may return with less than the requested number of bytes. The common choice is a non-zero value, to match standard UNIX behavior.
- synchronous: A non-zero value means that the pipe is synchronous. See Synchronization above.
- bufsize: Each DMA buffer's size. Always a power of two.
- bufnum: The number of buffers allocated for this pipe. Always a power of two.
- exclusive_open: A non-zero value forces exclusive opening of the associated device file. If the device file is bidirectional, and already opened only in one direction, the opposite direction may be opened once.
- seekable: A non-zero value indicates that the pipe is seekable. See Seekable pipes above.
- supports_nonempty: A non-zero value (which is typical) indicates that the hardware will send the messages that are necessary to support select() and poll() for this pipe.

Host never reads from the FPGA

Even though PCI Express is hotpluggable in general, a typical motherboard doesn't expect a card to go away all of the sudden. But since the PCIe card is based upon reprogrammable logic, a sudden disappearance from the bus is quite likely as a result of an accidental reprogramming of the FPGA while the host is up. In practice, nothing happens immediately in such a situation. But if the host attempts to read from an address that is mapped to the PCI Express device, that leads to an immediate freeze of the system on some motherboards, even though the PCIe standard requires a graceful recovery.

In order to avoid these freezes, the Xillybus driver refrains completely from reading from the device's register space. All communication from the FPGA to the host is done through DMA. In particular, the Interrupt Service Routine doesn't follow the common practice of checking a status register when it's invoked. Rather, the FPGA prepares a small buffer which contains short messages, which inform the host what the interrupt was about.

This mechanism is used on non-PCIe buses as well for the sake of uniformity.

Channels, pipes, and the message channel

Each of the (possibly bidirectional) pipes presented to the user is allocated a data channel between the FPGA and the host. The distinction between channels and pipes is necessary only because of channel 0, which is used for interrupt- related messages from the FPGA, and has no pipe attached to it.

Data streaming

Even though a non-segmented data stream is presented to the user at both sides, the implementation relies on a set of DMA buffers which is allocated for each channel. For the sake of illustration, let's take the FPGA to host direction: As data streams into the respective channel's interface in the FPGA, the Xillybus IP core writes it to one of the DMA buffers. When the buffer is full, the FPGA informs the host about that (appending a XILLYMSG_OPCODE_RELEASEBUF message channel 0 and sending an interrupt if necessary). The host responds by making the data available for reading through the character device. When all data has been read, the host writes on the FPGA's buffer control register, allowing the buffer's overwriting. Flow control mechanisms exist on both sides to prevent underflows and overflows.

This is not good enough for creating a TCP/IP-like stream: If the data flow stops momentarily before a DMA buffer is filled, the intuitive expectation is that the partial data in buffer will arrive anyhow, despite the buffer not being completed. This is implemented by adding a field in the XILLYMSG_OPCODE_RELEASEBUF message, through which the FPGA informs not just which buffer is submitted, but how much data it contains.

But the FPGA will submit a partially filled buffer only if directed to do so by the host. This situation occurs when the read() method has been blocking for XILLY_RX_TIMEOUT jiffies (currently 10 ms), after which the host commands the FPGA to submit a DMA buffer as soon as it can. This timeout mechanism balances between bus bandwidth efficiency (preventing a lot of partially filled buffers being sent) and a latency held fairly low for tails of data.

A similar setting is used in the host to FPGA direction. The handling of partial DMA buffers is somewhat different, though. The user can tell the driver to submit all data it has in the buffers to the FPGA, by issuing a write() with the byte count set to zero. This is similar to a flush request, but it doesn't block. There is also an autoflushing mechanism, which triggers an equivalent flush roughly XILLY_RX_TIMEOUT jiffies after the last write(). This allows the user to be oblivious about the underlying buffering mechanism and yet enjoy a stream-like interface.

Note that the issue of partial buffer flushing is irrelevant for pipes having the "synchronous" attribute nonzero, since synchronous pipes don't allow data to lay around in the DMA buffers between read() and write() anyhow.

Data granularity

The data arrives or is sent at the FPGA as 8, 16 or 32 bit wide words, as configured by the "format" attribute. Whenever possible, the driver attempts to hide this when the pipe is accessed differently from its natural alignment. For example, reading single bytes from a pipe with 32 bit granularity works with no issues. Writing single bytes to pipes with 16 or 32 bit granularity will also work, but the driver can't send partially completed words to the FPGA, so the transmission of up to one word may be held until it's fully occupied with user data.

This somewhat complicates the handling of host to FPGA streams, because when a buffer is flushed, it may contain up to 3 bytes don't form a word in the FPGA, and hence can't be sent. To prevent loss of data, these leftover bytes need to be moved to the next buffer. The parts in xillybus_core.c that mention "leftovers" in some way are related to this complication.

Probing

As mentioned earlier, the number of pipes that are created when the driver loads and their attributes depend on the Xillybus IP core in the FPGA. During the driver's initialization, a blob containing configuration info, the Interface Description Table (IDT), is sent from the FPGA to the host. The bootstrap process is done in three phases:

1. Acquire the length of the IDT, so a buffer can be allocated for it. This is done by sending a quiesce command to the device, since the acknowledge for this command contains the IDT's buffer length.
2. Acquire the IDT itself.
3. Create the interfaces according to the IDT.

Buffer allocation

In order to simplify the logic that prevents illegal boundary crossings of PCIe packets, the following rule applies: If a buffer is smaller than 4kB, it must not cross a 4kB boundary. Otherwise, it must be 4kB aligned. The xilly_setupchannels() functions allocates these buffers by requesting whole pages from the kernel, and diving them into DMA buffers as necessary. Since all buffers' sizes are powers of two, it's possible to pack any set of such buffers, with a maximal waste of one page of memory.

All buffers are allocated when the driver is loaded. This is necessary, since large continuous physical memory segments are sometimes requested, which are more likely to be available when the system is freshly booted.

The allocation of buffer memory takes place in the same order they appear in the IDT. The driver relies on a rule that the pipes are sorted with decreasing buffer size in the IDT. If a requested buffer is larger or equal to a page, the necessary number of pages is requested from the kernel, and these are used for this buffer. If the requested buffer is smaller than a page, one single page is requested from the kernel, and that page is partially used. Or, if there already is a partially used page at hand, the buffer is packed into that page. It can be shown that all pages requested from the kernel (except possibly for the last) are 100% utilized this way.

The "nonempty" message (supporting poll)

In order to support the "poll" method (and hence select()), there is a small catch regarding the FPGA to host direction: The FPGA may have filled a DMA buffer with some data, but not submitted that buffer. If the host waited for the buffer's submission by the FPGA, there would be a possibility that the FPGA side has sent data, but a select() call would still block, because the host has not received any notification about this. This is solved with XILLYMSG_OPCODE_NONEMPTY messages sent by the FPGA when a channel goes from completely empty to containing some data.

These messages are used only to support poll() and select(). The IP core can be configured not to send them for a slight reduction of bandwidth.