(기계로 한국어 번역)

Xillybus host application programming guide for Linux

Xillybus Ltd.

www.xillybus.com

Version 3.0

이 문서는 영어에서 컴퓨터에 의해 자동으로 번역되었으므로 언어가 불분명할 수 있습니다. 이 문서는 원본에 비해 약간 오래되었을 수 있습니다.

가능하면 영문 문서를 참고하시기 바랍니다.

This document has been automatically translated from English by a computer, which may result in unclear language. This document may also be slightly outdated in relation to the original.

If possible, please refer to the document in English.

1 소개 **5**

2 동기식 streams 대 비동기식 streams **7**

2.1 개요 7
2.2 비동기식 streams에 대한 동기 7
2.3 FPGA에서 host로 Streams 8
2.4 host에서 FPGA로 Streams 9
2.5 불확실성 대 latency 10

3 I/O 프로그래밍 사례 **11**

3.1 개요 11
3.2 데이터 읽기 지침 12
3.3 데이터 쓰기 지침 14
3.4 비동기식 downstreams에서 flush 수행 16
3.5 select() 및 nonblocking I/O 17
3.6 driver의 buffers에 있는 데이터 양 모니터링 19
3.7 XillyUSB: 물리적 data link의 품질을 모니터링해야 할 필요성 19

4 고속의 연속 I/O **21**

4.1 기본 21
4.2 대형 driver의 buffers 22
4.3 user space의 RAM buffers 23
4.4 fifo.c 데모 애플리케이션 개요 24
4.5 fifo.c 수정 참고 사항 25
4.6 RAM FIFO 기능 26
4.6.1 fifo_init() 27
4.6.2 fifo_destroy() 27
4.6.3 fifo_request_drain() 27
4.6.4 fifo_drained() 28
4.6.5 fifo_request_write() 28
4.6.6 fifo_wrote() 29

4.6.7 fifo_done() 29
4.6.8 FIFO_BACKOFF define variable 29

5 주기적 frame buffers 31
5.1 소개 31
5.2 FIFO 예제 코드 조정 31
5.3 frames 떨어뜨리고 반복하기 33

6 특정 프로그래밍 기술 34
6.1 Seekable streams 34
6.2 양방향 streams 동기화 36
6.3 패킷 통신 36
6.4 hardware interrupts 에뮬레이션 37
6.5 Timeout 38
6.6 Coprocessing/ Hardware acceleration 41

A 내부: streams 구현 방법 43
A.1 소개 43
A.2 “Classic” DMA 대 Xillybus 43
A.3 FPGA host(upstream) 44
A.3.1 개요 44
A.3.2 #1 단계: Application logic 중급 FIFO 45
A.3.3 #2 단계: 중급 FIFO DMA buffer 45
A.3.4 스테이지 #3: DMA buffer에서 사용자 소프트웨어 애플리케이션으로 45
A.3.5 일부만 채워진 buffers 인계 조건 47
A.3.6 예 47
A.3.7 실용적인 결론 49
A.4 Host FPGA(downstream) 49
A.4.1 개요 49
A.4.2 #1 단계: DMA buffer에 사용자 소프트웨어 적용 50
A.4.3 #2 단계: DMA buffer 중급 FIFO 51

A.4.4 #3 단계: 중급 FIFO application logic 52
A.4.5 예 52
A.4.6 실용적인 결론 52

1 소개

Xillybus는 자연스럽고 예상되는 동작을 가진 간단하고 잘 알려진 인터페이스를 Linux host에 제공하도록 설계되었습니다. host driver는 named pipes처럼 동작하는 device files를 생성합니다. 그것들은 다른 파일과 마찬가지로 열리고, 읽히고, 쓰여지지만 프로세스 또는 TCP/IP streams 사이에서 pipes와 매우 유사하게 작동합니다. host에서 실행되는 프로그램의 차이점은 stream의 다른 쪽이 다른 프로세스(네트워크 또는 동일한 컴퓨터를 통해)가 아니라 FPGA의 FIFO라는 점입니다. TCP/IP stream와 마찬가지로 Xillybus stream은 고속 데이터 전송과 가끔 도착하거나 전송되는 단일 바이트에서 잘 작동하도록 설계되었습니다.

Xillybus와의 인터페이스는 모든 파일처럼 액세스되는 device files를 통해 이루어지기 때문에 일반적으로 특별한 모듈, 확장 또는 기타 조정 없이 모든 실용적인 프로그래밍 언어를 사용할 수 있습니다. 선택한 언어로 파일을 열 수 있는 경우 해당 파일을 사용하여 FPGA에서 Xillybus까지 액세스할 수 있습니다.

하나의 driver binary는 모든 Xillybus IP core 구성을 지원합니다. streams 및 해당 속성은 장치를 초기화할 때 driver에 의해 자동 감지되고 이에 따라 device files가 생성됩니다. 이러한 device files는 `/dev/xillybus_something`(또는 XillyUSB가 있는 `/dev/xillyusb_something`)로 액세스됩니다.

작동하는 동안 FPGA와 host 사이의 핸드셰이크 프로토콜은 연속적인 data stream처럼 보입니다. 무대 뒤에서 driver의 buffers가 채워지고 처리됩니다. TCP/IP streaming에 사용된 것과 유사한 기술은 buffers의 효율적인 활용을 보장하는 동시에 작은 데이터 조각에 대한 응답성을 유지하는 데 사용됩니다.

Xillybus I/O는 Linux의 모든 device file I/O와 마찬가지로 수행되므로 일반적인 프로그래밍 방식을 사용할 수 있으므로 프로그래밍 가이드가 필요하지 않습니다.

그럼에도 불구하고 FPGA와의 통신에는 종종 파일 I/O에 일반적이지 않은 작업이 포함됩니다. 이 가이드는 일반적인 FPGA 관련 프로젝트를 구현하는 방법과 최적의 성능을 달성

하는 방법을 제안합니다. 숙련된 프로그래머는 동일한 성공으로 다른 방법을 선택할 수 있습니다.

이 가이드의 대부분은 UNIX 시스템에서 I/O가 얼마나 강력하고 효율적인지에 대한 개요에 불과합니다. 이러한 기술에 익숙한 사람들은 이 가이드에서 중복되는 여러 부분을 발견할 수 있으며 실제로 그렇습니다. Xillybus는 새로운 API를 발명하기 위해 설계된 것이 아니라 숙련된 프로그래머가 기대하는 것처럼 작동하도록 설계되었습니다.

이 가이드의 예제는 명확성을 위해 일반 C로 제공됩니다. 저수준 시스템 호출과 밀접하게 관련된 것으로 알려진 함수 집합이 있기 때문입니다. 설명된 기술은 특히 host 및 FPGA 작업 간의 성능 및 동기화 요구 사항이 덜 엄격한 경우 Perl 및 Python와 같은 script 언어를 비롯한 여러 다른 언어로 구현할 수 있습니다.

일부 I/O는 shell scripts 및 단일 라이너로도 수행할 수 있습니다.

2 동기식 streams 대 비동기식 streams

2.1 개요

각 Xillybus stream에는 동기식 또는 비동기식으로 동작하는지 여부를 결정하는 플래그가 있습니다. 이 플래그의 값은 FPGA의 logic에 고정되어 있습니다.

stream이 비동기식으로 표시되면 해당 device file이 열려 있는 한 user space software의 개입 없이 FPGA와 host의 kernel driver 간에 데이터 통신이 허용됩니다.

비동기식 streams는 특히 데이터 흐름이 연속적일 때 더 나은 성능을 제공합니다. 동기식 streams는 다루기 쉬우며 user space application의 동작과 FPGA에서 일어나는 일 사이에 긴밀한 동기화가 필요할 때 선호되는 선택입니다.

IP Core Factory에서 생성된 사용자 지정 IP cores에서 각 stream을 동기식 또는 비동기식으로 만드는 것 사이의 선택은 “autoset internals”가 활성화될 때 도구 사용자가 선언한 대로 stream의 의도된 용도에 대한 정보를 기반으로 자동으로 결정됩니다. autoset 옵션이 꺼져 있으면 사용자가 명시적으로 이 선택을 합니다.

어느 쪽이든, IP Core Factory에서 다운로드한 번들에 포함된 “readme” 파일은 각 stream(다른 속성 중에서)에 대한 동기 또는 비동기 플래그를 지정합니다.

모든 demo bundles에서 xillybus_read_* 및 xillybus_write_*와 관련된 streams는 비동기식입니다. xillybus_mem_8은 seekable이므로 동기식입니다. XillyUSB를 사용하는 경우 해당 xillyusb_* 파일에도 동일하게 적용됩니다.

2.2 비동기식 streams에 대한 동기

Linux 및 Microsoft Windows와 같은 멀티태스킹 운영 체제는 CPU 시간 공유를 기반으로 합니다. 프로세스는 CPU의 시간 조각을 가져오고 일부 일정 알고리즘은 주어진 순간에 어떤 프로세스가 CPU를 가져오는지를 결정합니다.

프로세스에 대한 우선 순위를 설정할 가능성이 있더라도 프로세스가 계속 실행되거나 preemption 기간이 제한된 기간이 있다는 보장은 없으며 다중 프로세서 컴퓨터에서도 마찬가지입니다. 운영 체제의 기본 가정은 모든 프로세스가 CPU starvation의 모든 기간을 허용할 수 있다는 것입니다. 실시간 지향 응용 프로그램(예: 사운드 응용 프로그램 및 비디오 플레이어)에는 이 문제에 대한 명확한 솔루션이 없습니다. 대신 운영 체제의 일반적인 사실상 동작에 의존하고 I/O 버퍼링으로 preemption periods를 보완합니다.

비동기식 streams는 애플리케이션이 preempted이거나 다른 작업으로 바쁜 동안 데이터가 계속 흐르도록 하여 이 문제를 해결합니다. 어느 방향에서든 streams에 대한 이것의 정확한 중요성은 다음에 논의됩니다.

2.3 FPGA에서 host로 Streams

upstream 방향(FPGA host)에서 stream이 비동기식이면 FPGA의 IP core는 가능할 때마다 host driver의 buffers를 채우려고 시도합니다. 즉, 파일이 열려 있을 때 데이터를 사용할 수 있고 해당 buffers에 여유 공간이 있습니다.

반면에 stream이 동기식이면 host의 user application software에 file descriptor에서 해당 데이터를 읽으려는 보류 중인 요청이 있는 경우에만 IP core가 user application logic(일반적으로 FIFO에서)에서 데이터를 가져옵니다. 즉, user application software가 read() 함수 호출 중일 때입니다.

동기 streams는 주로 다음 두 가지 이유로 고대역폭 애플리케이션에서 피해야 합니다.

- 응용 프로그램이 preempted이거나 다른 작업을 수행하는 동안 데이터 흐름이 중단되므로 특정 기간 동안 물리적 채널이 사용되지 않은 상태로 유지됩니다. 대부분의 경우 이로 인해 대역폭 성능이 크게 저하됩니다.
- 이 시간 간격 동안 FPGA의 FIFO에서 overflow가 발생할 수 있습니다. 예를 들어, 채우기 비율이 100 MB/sec인 경우 2 kByte가 있는 일반적인 FPGA FIFO는 약 0.02 ms에서 비어 있는 상태에서 가득 찬 상태로 이동합니다. 실제로 이것은 user space program의 모든 preemption이 잠재적으로 FPGA에서 FIFO의 overflow를 유발할 수 있음을 의미합니다.

이러한 단점에도 불구하고 동기식 streams는 FPGA에서 데이터가 수집된 시간이 중요한 경우에 유용합니다. 특히 메모리와 같은 인터페이스에는 동기 인터페이스가 필요합니다.

application logic에서 FPGA의 Xillybus IP core가 수신한 데이터는 stream이 동기식인지 비동기식인지에 관계없이 host의 user space application에서 즉시 읽을 수 있습니다.

2.4 host에서 FPGA로 Streams

downstream 방향(host FPGA)에서 stream이 비동기적이라는 것은 host application의 write() 함수 호출이 대부분의 시간에 즉시 반환된다는 것을 의미합니다. 보다 정확하게는 데이터가 driver의 buffers에 완전히 저장될 수 있는 경우 device file에 쓰는 함수에 대한 호출이 즉시 반환됩니다. 그런 다음 데이터는 host의 application software에 관여하지 않고 FPGA에서 user application logic이 요청한 속도로 FPGA로 전송됩니다.

XillyUSB와 다른 Xillybus IP cores 사이에는 데이터가 FPGA에 대한 비동기식 streams를 대신하여 FPGA로 얼마나 빨리 전송되는지와 관련하여 약간의 차이가 있습니다.

PCIe 또는 AXI를 기반으로 하는 IP cores의 경우 다음 중 하나가 발생한 경우에만 데이터가 FPGA로 전송됩니다.

- 현재 DMA buffer가 가득 찼습니다(각 stream에 대해 여러 buffers가 있음).
- flush는 application software에 의해 device file에서 명시적으로 요청됩니다(단락 3.4 참조).
- file descriptor는 폐쇄 중입니다.
- 타이머가 만료되어 특정 시간(일반적으로 10 ms) 동안 stream에 아무 것도 기록되지 않은 경우 자동 flush를 강제 실행합니다.

XillyUSB stream을 사용하면 데이터가 거의 즉시 전송됩니다. 보다 정확하게는 driver는 고정 크기(일반적으로 64 kB)의 USB transfers를 큐에 넣으려고 시도하지만 전송할 데이터가 있고 관련 stream에 대해 큐에 대기 중인 다른 전송이 없는 경우 더 작은 전송이 큐에 대기됩니다. 따라서 각 stream에 대해 고정 크기보다 작은 대기열 전송은 한 번도 없지만 전송할 데이터가 있는 한 항상 최소 하나의 전송이 진행 중입니다. 그 결과 USB 전송을 효율적으로 사용하고 짧은 데이터 세그먼트에 빠르게 응답할 수 있습니다.

대체로 모든 IP cores(XillyUSB 및 기타 Xillybus IP cores)의 비동기식 streams는 거의 동일하게 동작하며 XillyUSB는 짧은 데이터 세그먼트(10 ms 지연 없음)에서 더 빠른 응답 시간을 가집니다.

반면에 stream이 동기식이면 device file에 쓰는 하위 수준 함수에 대한 호출은 모든 데이터가 FPGA에서 user application의 logic에 도달할 때까지 반환되지 않습니다. 일반적인 응용 프로그램에서 이는 write()에 대한 함수 호출이 반환될 때 FPGA의 IP core에 연결된 FIFO에 데이터가 도착했음을 나타냅니다.

중요한:

fwrite()와 같은 상위 레벨 I/O 기능에는 library functions에 의해 생성된 buffer layer가 포함됩니다. 따라서 동기식 streams의 경우에도 데이터가 FPGA에 도착하기 전에 fwrite() 및 유사한 기능이 반환될 수 있습니다.

동기 streams는 주로 다음 두 가지 이유로 고대역폭 애플리케이션에서 피해야 합니다.

- 응용 프로그램이 preempted이거나 다른 작업을 수행하는 동안 데이터 흐름이 중단되므로 특정 기간 동안 물리적 채널이 사용되지 않습니다. 대부분의 경우 이는 상당한 대역폭 성능 저하로 이어집니다.
- 이 시간 간격 동안 FPGA의 FIFO에서 underflow가 발생할 수 있습니다. 예를 들어 배수 속도가 100 MB/sec인 경우 2 kByte가 있는 일반적인 FPGA FIFO는 약 0.02 ms에서 전체에서 비어 있게 됩니다. 실제로 이것은 user space program의 모든 preemption이 잠재적으로 FPGA에서 FIFO의 underflow를 유발할 수 있음을 의미합니다.

이러한 단점에도 불구하고 동기식 streams는 데이터가 FPGA에 도착했음을 애플리케이션이 알아야 할 때 유용합니다. 이것은 stream이 다른 작업(예: 하드웨어 구성)이 발생하기 전에 실행해야 하는 명령을 전송하는 데 사용되는 경우입니다.

2.5 불확실성 대 latency

데이터 간의 동기화를 위해 비동기식 streams에서 낮은 latency가 필요한 일반적인 실수입니다. 예를 들어 응용 프로그램이 모뎀인 경우 일반적으로 수신된 샘플과 전송된 샘플 간에 동기화할 필요가 있습니다.

이것은 종종 동기화의 불확실성이 전체 latency보다 반드시 작다는 개념에 기반하여 잘못된 design로 이어집니다. 불확실성을 낮게 유지하기 위해 latency와 buffers는 가능한 한 작게 만들어 real-time 요구 사항이 까다로운 전체 시스템으로 이어집니다.

Xillybus를 사용하면 단락 6.2에 설명된 대로 동기화가 쉽게 (단일 샘플 수준에서) 완벽해집니다. 따라서 latency의 한계는 그러한 필요가 있는 경우 도착하는 데이터에 신속하게 대응해야 하는 필요성에서 파생됩니다.

예를 들어 모뎀에서 최대 latency는 애플리케이션의 데이터 소스가 전송되는 데이터에 응답하는 속도에 영향을 미칩니다. 카메라 애플리케이션에서 host는 변화하는 조명 조건을 보상하기 위해 shutter speed를 조정하도록 카메라를 프로그래밍할 수 있습니다. 더 큰 latency와 함께 도착하는 데이터는 이 control loop을 느리게 합니다.

이것들은 취해야 할 실제 고려 사항이며 여전히 latency와의 혼합 불확실성에 대한 오해에서 파생된 것보다 일반적으로 훨씬 덜 엄격합니다.

3 I/O 프로그래밍 사례

3.1 개요

Xillybus는 파일에 액세스할 수 있는 모든 프로그래밍 언어에서 제대로 작동하며 파일 액세스를 위한 모든 API가 적합합니다.

이 가이드에서는 open(), read(), write() 및 close()와 같은 기능을 기반으로 하는 낮은 수준의 API 세트에 중점을 둡니다. 이 집합은 다른 잘 알려진 집합(예: fopen(), fwrite(), fprintf() 등)보다 선택됩니다. 하위 수준 API의 기능에는 buffers의 추가 계층이 없기 때문입니다. 이러한 buffers는 성능에 긍정적인 영향을 미칠 수 있지만 실제 I/O 작업을 제어할 수 없습니다.

이는 데이터가 지속적으로 전송되고 소프트웨어 작동과 하드웨어가 있는 I/O 사이에 직접적인 관계가 예상되지 않는 경우 덜 중요합니다.

추가 buffer 레이어도 혼란을 일으켜 소프트웨어 버그가 없는 것처럼 보이게 할 수 있습니다. 예를 들어, fwrite()에 대한 함수 호출은 파일이 닫힐 때까지 I/O 작업을 수행하지 않고 RAM buffer에 데이터를 저장할 수 있습니다. 이를 인식하지 못하는 개발자는 실제로 데이터가 buffer에서 대기 중일 때 FPGA 측에서 아무 일도 일어나지 않았기 때문에 fwrite()가 실패했다고 생각하도록 오도할 수 있습니다.

이 섹션에서는 낮은 수준의 C run-time library 기능을 사용하여 권장되는 UNIX 프로그래밍 방법을 설명합니다. 이러한 관행에 대해 Xillybus에 특정한 것이 없기 때문에 이 설명은 완전성을 위해 여기에 제공됩니다.

코드 조각은 Getting started with Xillybus on a Linux host에 설명된 데모 애플리케이션에서 가져왔습니다. 이 예에서 device file 이름은 PCIe / AXI용 Xillybus IP core의 이름입니다. XillyUSB의 경우 접두사는 xillybus_* 대신 xillyusb_00_*입니다.

3.2 데이터 읽기 지침

변수가 다음과 같이 선언되었다고 가정합니다.

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

device file은 하위 수준 open로 열립니다(file descriptor는 integer 형식임).

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

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

device file이 이미 다른 프로세스에서 읽기 위해 열려 있으면 “Device or resource busy” (errno = EBUSY) 오류가 발생합니다(요청 시 비독점적 파일 열기 가능). “No such device” (errno = ENODEV)가 발생하면 쓰기 전용 stream을 열려고 할 가능성이 큽니다.

파일이 성공적으로 열리고 `buf`가 메모리에 할당된 buffer를 가리키면 다음을 사용하여 데이터를 읽습니다.

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

`numbytes`는 읽을 최대 바이트 수입니다.

반환된 값 `rc`에는 실제로 읽은 바이트 수가 포함됩니다(또는 함수 호출이 비정상적으로 완료된 경우 음수 값).

read()는 `numbytes`에서 요청한 데이터 양이 사용 가능한 경우 항상 즉시 반환됩니다. 그렇지 않으면 사용 가능한 데이터가 있으면 약 10 ms 후에 반환됩니다. 사용 가능한 데이터가 전혀 없으면 read()는 데이터와 함께 반환될 때까지 휴면합니다.

driver는 IP core가 FPGA의 application logic에서 해당 데이터를 수신했다는 의미에서 데이터의 가용성을 확인합니다. DMA buffers의 메커니즘은 함수 read()의 호출자에게 투명하며 부록의 섹션 A.3.5에 설명된 대로 DMA buffer가 가득 차지 않았기 때문에 read() 함수 호출에 대한 데이터 전달을 지연하지 않습니다.

중요한:

read()가 성공적으로 반환되더라도 요청된 모든 바이트가 파일에서 읽혔다는 보장은 없습니다. 완료된 데이터 양이 만족스럽지 않은 경우 read()에 대한 다른 함수 호출을 수행하는 것은 호출자의 책임입니다.

read()에 대한 함수 호출은 아래와 같이 반환 값을 확인해야 합니다(“continue” 및 “break”

문은 while 루프 컨텍스트를 가정함).

```
  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
}
```

첫 번째 if 문은 signal로 인해 read()가 조기에 반환되었는지 확인합니다. 이는 운영 체제에서 signal을 수신하는 프로세스의 결과입니다.

이것은 실제로 오류가 아니라 driver가 제어를 애플리케이션에 즉시 반환하도록 강제하는 조건입니다. EINTR 오류 번호를 사용하는 것은 데이터를 읽지 않았음을 함수 호출자에게 알리는 방법일 뿐입니다. 프로그램은 “continue” 문으로 응답하여 동일한 매개변수를 사용하여 read() 함수 호출을 새로 시도합니다.

signal이 도착했을 때 buffer에 데이터가 있으면 driver는 `rc`에서 이미 읽은 바이트 수를 반환합니다. 응용 프로그램은 signal이 도착했음을 알지 못하며 UNIX 프로그래밍 규칙에 따라 신경 쓸 이유가 없습니다. signal에 작업이 필요한 경우(예: 키보드의 CTRL-C로 인한 SIGINT) 이 작업에 대한 책임은 다음 중 하나에 있습니다. 운영 체제 또는 등록된 signal handler.

일부 signals는 실행 흐름에 영향을 미치지 않아야 하므로 위에 표시된 대로 signals가 감지되지 않으면 프로그램이 명백한 이유 없이 갑자기 오류를 보고할 수 있습니다.

EINTR 시나리오를 처리하는 것도 프로세스를 중지하고(CTRL-Z와 같이) 적절하게 재개할 수 있도록 하는 데 필요합니다.

두 번째 if 문은 사용자가 읽을 수 있는 오류 메시지를 보고한 후 실제 오류가 발생한 경우 루프를 종료합니다.

세 번째 if 문은 end of file에 도달했는지 감지하며 이는 반환 값 0으로 표시됩니다. Xillybus device file에서 읽을 때 이런 일이 발생하는 유일한 이유는 application logic이 stream의 _eof 핀(FPGA의 IP core 인터페이스의 일부)을 올렸기 때문입니다.

3.3 데이터 쓰기 지침

변수가 다음과 같이 선언되었다고 가정합니다.

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

device file은 하위 수준 open로 열립니다(file descriptor는 integer 형식임).

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

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

device file이 이미 다른 프로세스에서 쓰기 위해 열려 있으면 “Device or resource busy” (errno = EBUSY) 오류가 발생합니다(요청 시 비독점적 파일 열기 가능). “No such device” (errno = ENODEV)가 발생하면 읽기 전용 stream을 열려고 할 가능성이 큽니다.

파일이 성공적으로 열리고 `buf`가 메모리에 할당된 buffer를 가리키면 데이터가 다음과 같이 기록됩니다.

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

`numbytes`는 기록할 최대 바이트 수입니다.

반환된 값 `rc`에는 실제로 기록된 바이트 수가 포함됩니다(또는 함수 호출이 비정상적으로 완료된 경우 음수 값).

중요한:

write()가 성공적으로 반환된 경우에도 요청된 모든 바이트가 파일에 기록되었다는 보장은 없습니다. 완료된 데이터 양이 만족스럽지 않은 경우 write()에 대한 다른 함수 호출을 수행하는 것은 호출자의 책임입니다.

write()에 대한 함수 호출은 아래와 같이 반환 값을 확인해야 합니다(“continue” 및 “break” 문은 while 루프 컨텍스트를 가정함).

```
  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
}
```

첫 번째 if 문은 signal로 인해 write()가 조기에 반환되었는지 확인합니다. 이는 운영 체제에서 signal을 수신하는 프로세스의 결과입니다.

이것은 실제로 오류가 아니라 driver가 응용 프로그램에 즉시 제어를 반환하도록 강제하는 조건입니다. EINTR 오류 번호를 사용하는 것은 함수 호출자에게 쓰여진 데이터가 없다는 것을 알리는 방법일 뿐입니다. 프로그램은 “continue” 문으로 응답하여 동일한 매개변수를 사용하여 write() 함수 호출을 새로 시도합니다.

signal이 도착하기 전에 일부 데이터가 기록된 경우 driver는 `rc`에 이미 기록된 바이트 수를 반환합니다. 응용 프로그램은 signal이 도착했음을 알지 못하며 UNIX 프로그래밍 규칙에 따라 신경 쓸 이유가 없습니다. signal에 작업이 필요한 경우(예: 키보드의 CTRL-C로 인한 SIGINT), 이 작업에 대한 책임은 운영 체제 또는 등록된 signal handler.

일부 signals는 실행 흐름에 영향을 미치지 않아야 하므로 위에 표시된 대로 signals가 감지되지 않으면 프로그램이 명백한 이유 없이 갑자기 오류를 보고할 수 있습니다.

EINTR 시나리오를 처리하는 것도 프로세스를 중지하고(CTRL-Z와 같이) 적절하게 재개할 수 있도록 하는 데 필요합니다.

두 번째 if 문은 사용자가 작성할 수 있는 오류 메시지를 보고한 후 실제 오류가 발생한 경우 루프를 종료합니다.

세 번째 if 문은 end of file에 도달했는지 감지하며 이는 반환 값 0으로 표시됩니다. Xillybus device file에 쓸 때 이런 일이 발생해서는 안됩니다.

3.4 비동기식 downstreams에서 flush 수행

2.4 단락에서 언급했듯이 PCIe / AXI IP core의 비동기식 stream에 기록된 데이터는 DMA buffer가 가득 차 있지 않는 한 FPGA로 즉시 전송되지 않습니다(여러 개의 DMA buffers 있음). 이 동작은 할당된 buffer 공간이 활용되도록 하여 성능을 향상시킵니다. 이것은 또한 PCIe / AXI bus에서 전송되는 패킷의 효율성을 향상시킵니다.

이미 언급했듯이 XillyUSB IP cores는 stream이 비동기식인 경우에도 USB 인터페이스에 효율적인 배열이 있기 때문에 데이터를 거의 즉시 보냅니다. 따라서 flush를 수행하는 것은 전송이 완료될 때까지 기다리는 것과 관련된 경우에만 XillyUSB IP cores에서 의미가 있습니다.

Streams에서 FPGA는 file descriptor를 닫을 때 자동으로 flush를 거치지만 이것은 신뢰할 수 없는 최선의 메커니즘입니다. close()에 대한 함수 호출은 write() 함수 호출이 동기식 streams에서 지연되는 방식과 유사한 방식으로 모든 데이터가 FPGA에 도착할 때까지 지연됩니다. 중요한 차이점은 close()는 flush가 완료될 때까지 최대 1초를 기다립니다. 그 때까지 flush가 완료되지 않으면 close()는 어쨌든 반환하고 system log에 경고 메시지를 표시합니다. 그러나 일부 드문 시나리오에서 file descriptor를 닫는 동안 남은 데이터의 마지막 몇 단어가 경고 없이 손실될 수 있습니다.

길이가 0인 buffer로 write() 함수를 호출하여 비동기식 stream의 flush를 명시적으로 요청할 수도 있습니다.

```
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
}
```

다음 사항에 유의하십시오.

- UNIX용 manual page는 count가 0일 때 write() 함수 호출이 수행해야 하는 작업을 정의하지 않으므로 선택은 각 device driver에 맡겨집니다. flushing에 대한 이 방법은 Xillybus에만 해당됩니다.
- close()와 달리 위에 표시된 write()는 FPGA에서 데이터가 소비되는 시점에 관계없

이 즉시 반환됩니다.

- 이 때문에 이런 종류의 write()는 XillyUSB에서 무의미합니다. 아무 관련이 없으며 실제로 아무 것도 하지 않습니다. 데이터는 어쨌든 거의 즉시 전송되며 write() 함수 호출은 어떤 경우에도 기다리지 않습니다.
- buffer에서 데이터를 읽지 않기 때문에 write() 함수 호출의 buffer 인수는 위에서 설명한 것처럼 NULL을 포함한 모든 값을 사용할 수 있습니다.
- 길이가 0인 buffer와 함께 더 높은 수준의 API를 사용하면 전혀 효과가 없을 수 있습니다. 예를 들어, 이 함수가 일반적으로 하는 일은 C run-time library에 의해 생성된 buffer에 데이터를 추가하는 것이기 때문에 0바이트를 쓰기 위해 함수 fwrite()를 호출하면 아무 작업도 수행하지 않고 단순히 반환될 수 있습니다.
- fflush()는 관련이 없습니다. 상위 수준 buffer의 flush를 수행하지만 하위 수준 driver에 flush 명령을 보내지 않습니다.
- streams에서 flush를 다른 방향(FPGA에서 host로)으로 수행할 필요가 없으며 그렇게 할 방법도 없습니다. 이는 host의 데이터 읽기 시도가 프로세스를 절전 모드(즉, block)로 전환하려고 할 때 이러한 streams의 flush가 자동으로 수행되기 때문입니다.

3.5 select() 및 nonblocking I/O

권장되지는 않지만 Linux용 Xillybus driver는 nonblocking calls 및 select() 기능을 지원합니다. Windows용 driver는 유사한 것을 지원하지 않으므로 이 기능을 사용하면 필요한 경우 애플리케이션을 이식하기가 더 어려워집니다. 여러 소스를 처리하는 데 권장되는 방법은 4.4 단락에서 논의된 fifo.c 예제 프로그램에서 설명한 것처럼 여러 threads(및 바람직하게는 RAM FIFOs)를 사용하는 것입니다.

select(), pselect() 및 poll()에 대한 함수 호출은 읽기 및 쓰기를 위해 모든 UNIX file descriptor와 마찬가지로 사용할 수 있습니다.

nonblocking calls 및 select() 기능은 IP Core Factory에서 “Windows only”로 설정된 Xillybus IP cores에서 활성화되지 않습니다.

완전성을 위해 nonblocking 읽기를 사용하여 단락 3.2의 데이터 읽기를 위한 코드 개요를 다시 살펴보겠습니다. 이 코드는 UNIX의 파일에서 nonblocking을 읽는 일반적인 방법을 보여줍니다.

파일은 O_NONBLOCK 플래그로 열립니다.

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

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

파일을 읽는 방법, 인수 또는 반환 값의 의미에는 차이가 없습니다.

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

그러나 이제 반환 값에 대한 또 다른 확인이 있습니다. `rc`가 음수이고 EAGAIN이 오류 코드로 제공되면 읽을 것이 없다는 의미입니다. 더 정확하게는 driver의 buffers에는 데이터가 없고 FPGA의 FIFO는 비어 있습니다.

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

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

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

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

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

함수 호출이 EAGAIN와 함께 반환될 때 의미 있는 작업이 수행되지 않는 한 위의 코드는 의미가 없습니다. 그렇지 않으면 읽을 데이터가 없을 때 잠자는 대신 while 루프에서 회전하여 CPU time을 낭비합니다.

nonblocking 쓰기의 경우 3.3 단락의 예에서 각각 변경합니다.

3.6 driver의 buffers에 있는 데이터 양 모니터링

이 항목은 “Monitoring the amount of buffered data”라는 섹션의 Xillybus FPGA designer’s guide에서 설명합니다.

3.7 XillyUSB: 물리적 data link의 품질을 모니터링해야 할 필요성

PCIe와 달리 USB 3.0와 함께 사용되는 물리적 data link은 bit errors를 생성하는 것으로 관찰되었습니다. 이것은 드문 일이며 관련된 구성 요소 중 하나(host의 USB 포트 또는 케이블)에 문제가 있음을 나타냅니다.

USB 프로토콜은 bit errors가 발생할 때 이를 극복하기 위한 다양한 메커니즘을 제공하지만 이러한 오류의 무작위 특성으로 인해 link protocol은 거의 도달하지 못하는 상태가 됩니다. 결과적으로 이것은 host의 USB controller에 있는 버그를 드러낼 수 있습니다. 이러한 버그는 존재하는 한 일반적으로 숨겨져 있으며 다양한 이상한 행동을 유발합니다.

따라서 물리적 data link이 빈번한 bit errors로 인해 문제가 발생하면 USB 연결이 중단되거나 자발적으로 연결이 끊기거나 드물게는 애플리케이션 데이터에 오류가 발생할 수 있는 상당한 위험이 있습니다.

XillyUSB는 전용 device file, /dev/xillyusb_NN_diagnostics를 통해 물리적 data link의 상태를 모니터링하는 수단을 제공합니다. showdiagnostics 유틸리티(이 web page 에서 설명)는 이 문제에 대해 수집된 정보를 노출합니다.

XillyUSB 기반 애플리케이션은 showdiagnostics 유틸리티에 의해 표시되는 처음 5개 카운터(잘못된 패킷, 감지된 오류 및 Recovery 요청과 관련)를 지속적으로 모니터링하고 증가하지 않는지 확인하는 것이 좋습니다. 그렇게 하는 경우, 특히 반복적으로 증가하는 경우 응용 프로그램 소프트웨어는 다음 중 하나와 같은 수정 조치를 제안해야 합니다.

- USB 플러그를 분리했다가 다른 포트에 다시 연결합니다. 일부 마더보드에는 다른 브랜드의 USB host 컨트롤러에 연결된 다른 포트가 있기 때문에 도움이 될 수 있습니다(일반적으로 USB 3.x 프로토콜의 이후 버전을 지원하기 위해).
- 동일한 포트에서 USB 플러그를 분리했다가 다시 연결합니다. 이것은 analog signal equalizer(물리적 신호 경로를 유발하는 감쇠 및 반사를 취소함)가 차선책 상태로 끝나는 경우 도움이 될 수 있습니다.
- 다른 USB 케이블을 사용해 보십시오.

bit errors가 있는 경우에도 응용 프로그램이 계속해서 완벽하게 작동할 가능성이 큽니다. 따라서 수정 조치에 대한 제안은 사용자가 눈에 보이는 문제를 경험하지 않을 수 있다는 점을 고려하면서 가장 잘 수행됩니다.

showdiagnotics.pl 유틸리티는 참조 코드로 사용할 수 있는 Perl script입니다. 또는 C 소스 코드로 제공되는 Windows용 진단 유틸리티를 참조할 수 있습니다.

이러한 문제 중 어느 것도 XillyUSB에만 해당되는 것은 아닙니다. 오히려 이러한 문제는 모든 USB 3.0 장치에 영향을 미칠 가능성이 높지만 XillyUSB는 이러한 문제를 감지할 수 있는 수단을 제공합니다. 또한 PCIe 링크는 더 잘 제어된 물리적 연결 및 신호 라우팅으로 인해 유사한 문제를 겪지 않는 것으로 알려져 있습니다.

4 고속의 연속 I/O

4.1 기본

host와 FPGA 사이에 고속 연속 데이터 흐름을 달성하는 데 거의 필수적인 4가지 방법이 있습니다.

- 비동기식 streams 사용
- driver의 buffers가 user space application의 I/O 작업 사이의 시간 간격을 보상할 만큼 충분히 큰지 확인합니다.
- user space application이 사용 가능한 데이터가 있는 즉시 device file에서 데이터를 읽거나 buffers에 사용 가능한 공간이 생기는 즉시 데이터를 쓰도록 합니다.
- FPGA가 데이터를 계속 삽입하거나 비우는 동안 device files를 닫았다가 다시 열지 마십시오.

XillyUSB는 이 web page에 설명된 대로 데이터의 지속적인 흐름을 유지하는 데 추가적인 과제를 제시합니다.

주어진 시간에 driver의 buffers에 얼마나 많은 데이터가 있는지 모니터링하는 방법은 Xillybus FPGA designer's guide의 "Monitoring the amount of buffered data" 섹션에서 설명합니다.

비동기 streams를 사용하는 위 목록의 첫 번째 항목은 2 섹션에서 설명합니다. 두 번째와 세 번째는 이 섹션의 나머지 부분에서 설명합니다.

네 번째 항목을 이해하려면 비동기식 streams의 장점이 user space application의 개입 없이 FPGA와 host 사이에서 데이터가 실행된다는 점을 기억하십시오. 이 흐름은 파일이 닫힐 때 중지됩니다.

특히 host에서 FPGA로의 stream의 경우 파일을 닫으면 buffers에 있는 모든 데이터의 flush가 강제로 실행되고 파일은 완료된 후에만(또는 1초 후에) 닫힙니다. 결과적으로 파일이 닫힌 순간부터 파일이 다시 열릴 때까지(그리고 데이터가 file descriptor에 기록될 때까지) 데이터 흐름이 없는 시간 간격이 있습니다.

FPGA의 streams의 경우 파일을 닫으면 FPGA의 application logic에서 host의 user space application(즉, FPGA의 FIFO 및 driver의 buffers)로 이동하는 pipe의 모든 데이터가 손실됩니다. 이 손실을 피하는 유일한 방법은 파일을 닫기 전에 이 pipe의 모든 데이터를 비우는 것입니다. 다시 한 번, 파일을 닫고 다시 여는 사이에 데이터가 흐르지 않는 시간 간격이 있습니다.

일반적인 실수는 EOF 기능을 사용하여 데이터 청크(예: 완전한 video frames)를 표시하고 이를 통해 host가 알려진 경계에서 device file을 닫았다가 다시 열도록 하는 것입니다. 그러나 이것은 FPGA의 FIFO에서 overflow의 위험을 상당히 증가시킵니다.

운영 체제는 주어진 순간(preemption)에 user space application에서 CPU를 제거할 수 있으므로 프로그램의 후속 함수 호출 사이에 몇 밀리초, 때로는 수십 밀리초의 시간 간격이 발생할 수 있음을 명심하는 것이 중요합니다.

4.2 대형 driver의 buffers

FPGA와 host 간에 고속으로 데이터를 전송하는 데 있어 가장 큰 문제 중 하나는 지속적인 흐름을 유지하는 것입니다. data acquisition 및 재생과 관련된 응용 프로그램에서 overflow 또는 데이터 부족으로 인해 시스템이 작동하지 않습니다. 이를 피하기 위해 driver는 자체 사용을 위해 host에 큰 RAM buffers를 할당합니다. 이러한 buffers는 애플리케이션이 데이터 전송을 처리할 수 없는 시간 간격을 보완합니다.

Xillybus는 거대한 driver의 buffers를 할당할 수 있지만 이 메모리는 운영 체제의 kernel RAM 풀에서 할당되어야 합니다. 일부 시스템(특히 32비트 시스템)에서 이러한 메모리의 주소 지정 공간은 사용 가능한 총 RAM이 훨씬 더 크더라도 Linux 운영 체제에 의해 1 GB로 제한됩니다. RAM이 1 GB(특히 embedded Linux)보다 작은 시스템에서는 모든 메모리가 driver의 buffers에 사용될 수 있습니다.

이 페이지에서 설명하는 것처럼 향상된 host driver를 사용할 때 훨씬 더 큰 buffers를 64비트 시스템에 할당할 수 있습니다.

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

XillyUSB를 제외하고 driver의 buffers는 Xillybus driver가 로드될 때 할당되고(일반적으로 boot 프로세스 초기) driver가 kernel에서 언로드될 때만 해제됩니다(일반적으로 system shutdown 동안). buffers가 클 때 이는 일반적으로 kernel의 RAM 풀의 상당 부분이 driver의 buffers에 의해 점유되고 있음을 의미합니다. 이러한 buffers를 사용하는 응용 프로그램이 실행 중인 컴퓨터의 주요 목적일 가능성이 높기 때문에 상당히 합리적인 설정

입니다.

거대한 buffers의 잠재적인 문제는 물리적 RAM의 연속 세그먼트를 차지한다는 것입니다. 이는 userspace 프로그램에 할당된 buffer와 반대되는 것으로 virtual address space에서 연속적이지만 물리적 메모리 전체에 퍼질 수 있거나 물리적 RAM을 전혀 차지하지 않을 수도 있습니다.

사용 가능한 메모리 풀은 운영 체제가 실행됨에 따라 조각화됩니다. 이것이 Xillybus driver가 가능한 한 빨리 buffers를 할당하고 활발히 사용되지 않는 경우에도 유지하는 이유입니다. 같은 이유로 driver를 언로드하고 나중 단계에서 다시 로드하려고 하면 실패할 수 있습니다.

XillyUSB는 물리적 메모리 조각화에 더 관대한 메모리 할당 방식을 사용합니다. 이것은 driver가 device file이 열릴 때 buffers에 RAM을 할당하고 파일이 닫힐 때 해제하는 이유 중 하나입니다.

그러나 kernel RAM이 부족하지 않도록 예방 조치를 취해야 합니다. IP Core Factory의 자동 메모리 할당(“autoset internals”) 알고리즘은 최신 PC에 RAM의 1 GB 이상이 설치되어 있다는 가정에 따라 관련 메모리 풀(예: PC 컴퓨터의 경우 512 MB)의 50% 이상을 사용하지 않도록 설계되었습니다. buffer 크기를 수동으로 설정하여 수행할 수 있는 75%만큼 높게 가는 것이 안전할 수 있습니다.

buffers를 과도하게 할당하면 시스템이 불안정해질 수 있습니다. 특히, 운영 체제는 kernel pool에서 RAM을 할당하지 못할 때마다 분명히 무작위로 프로세스를 종료할 가능성이 높습니다.

4.3 user space의 RAM buffers

32비트 시스템에서 512 MB보다 큰 buffers가 필요한 응용 프로그램의 경우 user space RAM에서 일부 버퍼링을 수행하는 것이 좋습니다. 64비트 시스템에서 이 옵션은 원하는 buffer 크기가 매우 크고 2의 거듭제곱(2^N)이 아닌 경우를 제외하고 거의 관련이 없습니다. 예를 들어, stream에 62 GB의 buffer를 공급하는 것은 Xillybus DMA buffers 덕분에 가능하지 않지만 user space RAM로 달성할 수 있습니다.

user space application에 거대한 buffer를 할당하여 I/O 연속성 문제를 해결할 수 있다는 것은 직관에 어긋나는 것처럼 보일 수 있습니다. 실제로, 이 솔루션은 운영 체제가 CPU 시간의 애플리케이션을 굶주리게 할 때 도움이 되지 않습니다. 그러나 운영 체제의 scheduler가 상당히 잘 설계되고 우선 순위가 올바르게 설정되면 user space application은 부하가 높은 컴퓨터에서도 CPU 슬라이스를 충분히 자주 얻을 수 있습니다.

buffer의 첫 번째 채우기에 주의하는 것이 중요합니다. 최신 운영 체제는 user space application이 메모리를 요청할 때 물리적 RAM을 할당하지 않습니다. 대신 메모리 할당을 반영하도록 memory page tables를 설정했습니다. 실제 물리적 메모리는 응용 프로그

램이 사용하려고 할 때만 할당됩니다. 이것은 자원을 절약하는 훌륭한 방법이지만 data acquisition 애플리케이션에 치명적인 영향을 미칠 수 있습니다. 예를 들어 데이터 소스에서 데이터가 쇄도하기 시작할 때 어떤 일이 발생하는지 생각해 보십시오. 응용 프로그램은 방금 할당된 buffer에 데이터를 기록하지만 새 memory page에 액세스할 때마다 운영 체제는 새 physical memory page를 가져와야 합니다. 사용 가능한 물리적 RAM이 있거나 물리적 메모리를 해제하는 빠른 방법이 있는 경우(예: 이미 디스크와 동기화된 disk buffers) 이 메모리 저글링은 눈에 띄지 않게 지나갈 수 있습니다. 그러나 물리적 RAM의 즉각적인 소스가 없는 경우 디스크 작업(RAM swapping에서 디스크 또는 flushing disk buffers로)을 수행해야 할 수 있으며, 이로 인해 애플리케이션이 너무 오랫동안 중지될 수 있습니다.

정말 나쁜 소식은 데이터의 초기 로드를 수행하는 능력이 전체 시스템의 상태에 달려 있다는 것입니다. 따라서 다른 프로그램이 동일한 컴퓨터에서 데이터 집약적인 작업을 수행했기 때문에 일반적으로 작동하는 프로그램이 갑자기 실패할 수 있습니다.

자연적인 솔루션은 메모리 잠금입니다. mlock()은 운영 체제에 (가상) 메모리의 특정 청크가 물리적 RAM에 있어야 한다고 알려줍니다. 이렇게 하면 물리적 메모리가 즉시 할당되므로 함수 호출을 완료하는 데 디스크 작업이 필요한 경우 반환하는 데 시간이 걸릴 수 있습니다.

운영 체제는 전체 성능에 영향을 미치기 때문에 RAM의 큰 청크를 잠그는 것을 꺼립니다. 대부분의 경우 shell에서 일부 제한을 높이거나 구성 파일을 설정할 필요가 있습니다.

4.4 fifo.c 데모 애플리케이션 개요

Linux 및 Windows용으로 다운로드할 수 있는 데모 응용 프로그램 중에는 “fifo.c”라는 응용 프로그램이 있습니다. 32비트 및 64비트 플랫폼에서 테스트를 거친 두 개의 threads를 사용하여 RAM FIFO를 구현하는 방법의 예입니다.

데모 애플리케이션에 대한 자세한 내용은 Getting started with Xillybus on a Linux host를 참조하십시오.

문서의 다른 곳과 달리 이 섹션의 “FIFO”라는 단어는 FPGA의 FIFO가 아니라 host의 RAM buffer를 나타냅니다.

이 프로그램의 목적은 거대한 RAM buffer를 유지하기 위해 RAM FIFO가 필요한 빠른 streams를 테스트하는 것입니다. 즉, 16 GB보다 작은 buffer가 필요한 경우 이 프로그램이 필요하지 않을 가능성이 높습니다.

또한 사용자 지정 응용 프로그램에서 수정 및 채택을 위한 기반으로 사용할 수 있습니다. mutexes 없이 설계되었으므로 다른 thread가 lock을 보유한다고 해서 thread가 절전 모드로 전환되지 않습니다. 물론 FIFO의 상태가 이를 요구할 때(예: 빈 FIFO에서 읽기가 요청될 때) 잠자기(blocking)가 발생합니다.

mutexes 없이 구현하려면 reentrant가 아니므로 API 기능을 주의해서 사용해야 합니다. 그러나 이것은 하나의 읽기용 thread와 쓰기용 thread에서는 문제가 되지 않습니다.

device file에서 128 MB의 buffer가 있는 디스크 파일로 data acquisition용으로 실행하려면 다음과 같이 입력하십시오.

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

두 번째 인수로 파일 이름을 지정하지 않으면 프로그램은 standard input에서 읽습니다.

root 권한과 함께 shell prompt에서 'limit -l'을 사용하여 잠긴 메모리에 대한 제한을 해제해야 할 수도 있습니다(root로 "su - your-username"을 사용하여 권한을 일반 사용자에게 되돌리고 업데이트된 제한을 유지할 수 있음). 한도의 지속적인 변경에 대해서는 Linux 배포 문서를 참조하십시오.

프로그램은 세 개의 threads를 생성합니다.

- read_thread()는 standard input(또는 명령줄에 제공된 파일)에서 읽고 데이터를 FIFO에 씁니다.
- write_thread()는 FIFO에서 읽고 standard output에 씁니다.
- status_thread()는 standard error에 상태 표시줄을 반복적으로 인쇄합니다.

세 번째 thread는 기능적 의미가 없으며 제거할 수 있습니다. 메인 thread에서 읽기/쓰기 기능 중 하나를 실행하는 것도 가능합니다. 예를 들어, data acquisition 응용 프로그램에서 file descriptor에서 FIFO로 데이터를 이동하기 위해 read_thread()만 실행하는 것이 당연할 수 있지만 기본 응용 프로그램의 thread에서 FIFO의 데이터를 소비합니다.

4.5 fifo.c 수정 참고 사항

프로그램을 수정하려면 다음 사항에 유의하십시오.

- fifo_* 기능은 reentrant가 아닙니다. 각 thread가 다른 thread에서 사용하지 않는 일련의 기능을 사용할 때 사용하는 것이 안전합니다(자연스러운 사용).
- fifo_init() 함수는 반환하는 데 시간이 걸릴 수 있으며 비동기식 Xillybus device file이 열리기 전에 호출해야 합니다.
- 응용 프로그램에서 읽는 thread와 쓰는 thread는 항상 I/O 요청에서 허용되는 최대 바이트 수를 시도합니다. 이는 I/O 소스가 /dev/zero이고 대상이 /dev/null인 경우와 같은 일부 경우에 문제가 될 수 있습니다. 둘 다 한 번의 시도로 전체 요청을 완료하므로 FIFO는 완전히 비어 있는 상태에서 완전히 가득 찬 상태로 계속 이동합니다. 이

러한 경우 I/O 함수에 대한 호출에서 요청된 바이트 수를 제한하는 것이 더 합리적입니다.

4.6 RAM FIFO 기능

fifo.c 예제를 수정하는 것을 제외하고 소스 코드에서 기능 그룹을 채택하는 것이 가능합니다.

FIFO API 기능의 섹션은 fifo.c 파일에서 명확하게 구분됩니다. 이러한 기능은 예제를 따르고 아래 기능 설명에 따라 사용자 지정 응용 프로그램에서 사용할 수 있습니다.

중요한:

fifo_* 기능은 multi-threaded 환경에서 사용하기 위한 것이지만 이러한 기능은 ***reentrant가 아닙니다*** *. 즉, 하나의 thread는 FIFO에서 읽기와 관련된 함수를 호출해야 하고 다른 thread는 쓰기를 수행해야 하므로 각 thread는 별도의 함수 집합을 호출합니다.*

이니셜라이저, 디스트로이어 및 thread join 도우미를 제외하고 API에는 각 방향에 대해 2개씩 읽기 및 쓰기를 위한 4개의 기능이 있습니다. 이러한 기능 중 어느 것도 실제로 FIFO의 데이터에 액세스하지 않습니다. 그들은 단지 FIFO의 상태를 유지하고 읽기, 쓰기, 메모리 복사 등을 수행하는 데 필요한 정보를 제공합니다.

의도된 실행 절차는 다음과 같습니다. FIFO에서 읽는 thread는 읽을 수 있는 바이트 수에 대한 정보를 반환하는 함수 fifo_request_drain()와 데이터를 읽을 수 있는 pointer를 호출합니다. FIFO가 비어 있으면 데이터가 도착할 때까지 thread가 절전 모드로 전환됩니다.

그런 다음 사용자 응용 프로그램은 가리키는 데이터에 필요한 모든 것을 사용합니다. 데이터의 일부 또는 전체 소비를 마친 후(파일에 쓰기, 데이터 복사, 일부 알고리즘 실행 등) fifo_drained() 함수를 호출하여 실제로 소비된 바이트 수를 FIFO API에 알립니다. API는 FIFO에서 메모리의 관련 부분을 해제합니다. FIFO가 가득 차서 쓰는 thread가 잠자기 상태였다면 깨어납니다.

읽는 thread는 특정 바이트 수를 요구하지 않습니다. 오히려 fifo_request_drain()은 애플리케이션에 사용할 수 있는 바이트 수를 알려주고 애플리케이션은 fifo_drained()에서 소비하기로 선택한 바이트 수를 보고합니다.

반대 방향의 경우에도 비슷한 접근 방식을 취합니다. 쓰는 thread는 함수 fifo_request_write()를 호출합니다. 이 함수는 FIFO에 쓸 수 있는 바이트 수를 반환하거나 FIFO가 가득 찬 경우 휴면합니다. 사용자 응용 프로그램은 fifo_request_write()에서 가져온 주소에 필요한 만큼(fifo_request_write()에서 허용하는 만큼만) 바이트를 쓴 다음 fifo_wrote()에 수행한 작업을 다시 보고합니다.

이제 이러한 각 기능을 자세히 살펴보겠습니다.

4.6.1 fifo_init()

fifo_init(struct xillyfifo *fifo, unsigned int size)– 이 함수는 FIFO의 정보 구조를 초기화하고 FIFO용 메모리도 할당합니다. 또한 FIFO의 virtual memory를 물리적 RAM로 잠그려고 시도하여 즉각적인 빠른 쓰기를 준비하고 swapped to disk이 되는 것을 방지합니다.

fifo_init()은 `size` 바이트의 buffer에 메모리를 할당합니다. `size`는 모든 integer가 될 수 있지만(즉, 2의 거듭제곱, 2^N일 필요는 없음) 시스템에서 `int`로 간주하는 것의 배수가 권장됩니다.

이 함수는 반환하는 데 몇 초가 걸릴 수 있습니다. 물리적 RAM의 많은 부분에 대한 요청은 운영 체제에서 강제로 다른 프로세스의 RAM pages를 디스크로 바꾸거나 disk cache flushing을 강제로 실행할 수 있습니다. 두 경우 모두 fifo_init()은 반환하기 전에 많은 데이터가 디스크에 기록될 때까지 기다려야 할 수 있습니다.

함수는 성공하면 0을 반환하고 그렇지 않으면 0이 아닌 값을 반환합니다.

4.6.2 fifo_destroy()

fifo_destroy(struct xillyfifo *fifo)– 잠금을 해제한 후 FIFO의 메모리를 해제하고 thread synchronization 리소스를 해제합니다. 이 함수는 주 프로그램이 종료될 때 호출 **되어야** 합니다. thread synchronization 리소스가 Linux의 현재 구현에서 자동으로 해제되더라도 해당 API가 이를 보장하지 않기 때문입니다.

이 함수는 void 유형입니다(따라서 아무 것도 반환하지 않음).

4.6.3 fifo_request_drain()

fifo_request_drain(struct xillyfifo *fifo, struct xillyinfo *info)– FIFO에서 info->addr로 데이터를 읽을 수 있도록 pointer를 공급하고 info->bytes에서 해당 pointer부터 읽을 수 있는 바이트 수를 알려줍니다.

`info` 구조는 fifo_request_write()에 대한 함수 호출에 사용되는 것과 동일 **하지** 않아야 합니다. 각 thread는 이 구조에 대해 자체 로컬 변수를 유지해야 합니다.

중요한:

반환된 바이트 수는 FIFO에서 읽기 위해 남아 있는 데이터의 양을 나타내지 ***않습니다*** *. FIFO의 메모리 buffer가 끝날 때까지 남은 바이트 수를 반영할 수도 있습니다. 따라서 pointer가 buffer의 끝에 가까워지면 훨씬 더 낮은 숫자가 가능합니다.*

이 함수는 또한 FIFO의 현재 읽기 위치를 0과 size-1 사이의 값으로 나타내도록 fifo->position을 설정합니다. 여기서 `size`는 fifo_init()에 제공된 값입니다. 0이 아닌 fifo->slept는 호출 시 FIFO가 비어 있음을 나타냅니다.

이 함수는 읽기에 허용된 바이트 수를 반환합니다(info->taken와 동일). 그러나 함수 fifo_done()이 호출되고 FIFO가 비어 있으면 fifo_request_drain()은 0을 반환합니다.

4.6.4 fifo_drained()

fifo_drained(struct xillyfifo *fifo, unsigned int req_bytes)– 이 함수는 req_bytes 바이트 소비를 반영하도록 FIFO의 상태를 변경합니다. FIFO가 가득 차서 fifo_request_write()가 잠자기 상태였다면 깨어날 것입니다.

중요한:

*req_bytes **에는 온전성 검사가 없습니다** . req_bytes가 fifo_request_drain()에 대한 마지막 함수 호출에서 반환된 info->bytes보다 크지 않은지 확인하는 것은 사용자 애플리케이션의 책임입니다.*

이 함수는 void 유형입니다(따라서 아무 것도 반환하지 않음).

4.6.5 fifo_request_write()

fifo_request_write(struct xillyfifo *fifo, struct xillyinfo *info)– pointer를 공급하여 FIFO에 info->addr로 데이터를 기록하고 info->bytes에서 해당 pointer부터 시작할 수 있는 바이트 수를 알려줍니다.

`info` 구조는 fifo_request_drain()에 대한 함수 호출에 사용되는 것과 동일 **하지** 않아야 합니다. 각 thread는 이 구조에 대해 자체 로컬 변수를 유지해야 합니다.

중요한:

*반환된 바이트 수는 FIFO에 쓰기 위해 남은 데이터 양을 나타내지 **않습니다** . FIFO의 메모리 buffer가 끝날 때까지 남은 바이트 수를 반영할 수도 있습니다. 따라서 pointer가 buffer의 끝에 가까워지면 훨씬 더 낮은 숫자가 가능합니다.*

이 기능은 또한 FIFO의 현재 쓰기 위치를 0과 size-1 사이의 값으로 나타내도록 fifo->position을 설정합니다. 여기서 `size`는 fifo_init()에 제공된 값입니다. 0이 아닌 fifo->slept는 호출 시 FIFO가 가득 찼음을 나타냅니다.

이 함수는 쓰기에 허용된 바이트 수를 반환합니다(info->taken와 동일). 그러나 함수 fifo_done()이 호출된 경우 FIFO가 가득 차지 않았더라도 fifo_request_write()는 0을 반환합니다(읽지 않을 FIFO에 데이터를 쓰는 포인트가 없음).

4.6.6 fifo_wrote()

fifo_wrote(struct xillyfifo *fifo, unsigned int req_bytes)– 이 함수는 req_bytes 바이트의 삽입을 반영하도록 FIFO의 상태를 변경합니다. FIFO가 비어 있기 때문에 fifo_request_drain()이 잠자기 상태였다면 깨어날 것입니다.

중요한:

*req_bytes **에는 온전성 검사가 없습니다** . req_bytes가 fifo_request_write()에 대한 마지막 함수 호출에서 반환된 info->bytes보다 크지 않은지 확인하는 것은 사용자 애플리케이션의 책임입니다.*

이 함수는 void 유형입니다(따라서 아무 것도 반환하지 않음).

4.6.7 fifo_done()

fifo_done(struct xillyfifo *fifo)– 이 기능은 선택 사항이며 threads(읽기 또는 쓰기)가 완료된 경우 응용 프로그램이 정상적으로 종료되도록 도와줍니다. FIFO의 구조에 플래그를 설정하고 두 threads가 잠자기 상태인 경우 깨우기만 합니다. 이렇게 하면 FIFO가 비어 있는 경우 fifo_request_drain()은 절전 모드가 아닌 0을 반환하고 fifo_request_write()는 관계없이 0을 반환합니다.

이런 식으로 이러한 함수의 호출자는 FIFO가 더 이상 사용되지 않는다는 것을 알고 필요한 것으로 작동할 수 있으며, 이는 thread의 실행을 중지할 가능성이 가장 높습니다.

pipe에 공급하는 데이터 소스가 종료되거나(예: EOF에 도달) 데이터 소비자가 더 이상 수신하지 않을 때(예: broken pipe) 이 함수를 호출하십시오.

이 함수는 void 유형입니다(따라서 아무 것도 반환하지 않음).

4.6.8 FIFO_BACKOFF define variable

때로는 FIFO가 마지막 바이트까지 가득 차도록 두는 것이 바람직하지 않습니다. 이를 피하는 명백한 이유가 없더라도 데이터가 기록되는 위치와 데이터를 읽는 위치 사이에 작은 간격을 유지하는 것이 바람직할 수 있습니다.

예를 들어, FIFO_BACKOFF는 8로 설정될 수 있으므로 FIFO에 기록된 마지막 바이트는 읽기에 유효한 첫 번째 바이트와 64비트 워드를 공유하지 않습니다. 이것은 다소 무리한

예방책이지만 8바이트 메모리라는 저렴한 가격으로 제공됩니다.

Xillybus 또는 XillyUSB로 작업할 때는 이 기능이 필요하지 않습니다.

5 주기적 frame buffers

5.1 소개

일부 응용 프로그램, 특히 비디오 이미지의 실시간 처리에서는 각 buffer가 고정된 크기를 갖도록 여러 buffers를 유지해야 하는 경우가 많습니다. 비디오 처리 응용 프로그램에서 이러한 각 buffer에는 하나의 frame이 포함됩니다. 이를 통해 필요에 따라 frames를 건너뛰거나 두 번 이상 재생할 수 있습니다.

frame grabber 응용 프로그램에서 buffer가 비어 있을 때까지 하나 이상의 frames를 건너뛰어 overflow 상태를 처리할 수 있습니다. 예를 들어, 라이브 뷰 응용 프로그램에서 이러한 overflow 상태는 보기 창을 이동하거나 크기를 조정할 때 발생할 수 있습니다. 이와 같이 frames를 삭제하면 작은 latency를 유지하면서 비디오 소스의 지속적인 데이터 흐름이 중단되는 것을 방지할 수 있습니다.

frame replay 응용 프로그램(예: live output을 표시하는 화면)에서 표시할 최신 frame이 없을 때 출력 이미지가 반복됩니다. 이것은 소스(예: 디스크)가 일시적으로 정지되어 표시된 이미지가 잠시 동안 정지되는 상황을 해결합니다. 완전히 우아하지는 않지만 stream이 동기화되지 않는 것보다 낫습니다. 많은 경우에 이미지 반복 메커니즘은 다소 투박하지만 frame rates의 차이를 극복하는 데 잘 작동합니다. 특히 출력의 frame rate가 입력의 frame rate(예: 30 fps 60 fps)보다 상당히 높을 때 그렇습니다.

이 섹션에서는 4.4 단락에서 소개된 FIFO 데모 애플리케이션을 수정하여 이러한 종류의 buffers 세트를 관리하는 방법에 대해 설명합니다.

5.2 FIFO 예제 코드 조정

frame buffers와 FIFO의 순환 세트를 유지하는 것 사이에는 유사점이 있습니다. 실제로 FIFO의 각 바이트가 frame buffer를 나타내는 경우 FIFO의 특정 바이트를 읽거나 쓸 준비가 된 상태는 전체 frame buffer를 읽거나 쓸 준비가 된 것과 같습니다.

예를 들어, 수신된 이미지 데이터를 포함하기 위해 4개의 frame buffers가 할당된 frame grabber 애플리케이션을 가정합니다. 4바이트의 FIFO가 다음과 같이 4개의 frame buffers를 관리하는 데 도움이 되도록 설정되어 있다고 가정합니다.

데이터를 수신하는 thread는 첫 번째 frame buffer부터 시작하여 주기적으로 다음 frame buffer까지 계속됩니다. 새 frame buffer에 쓰기를 시작하기 전에 이 thread는 4바이트 FIFO가 가득 차지 않았는지 확인합니다. frame buffer를 완료한 후 FIFO에 바이트를 쓰고 FIFO가 가득 차지 않으면 다음 바이트로 이동합니다.

이미지 데이터를 소비하는 thread는 동일한 순서로 frame buffer를 순환합니다. 새 frame buffer에서 읽기를 시도하기 전에 4바이트 FIFO가 비어 있지 않은지 확인합니다. frame buffer로 작업을 마치고 다음으로 이동할 준비가 되면 FIFO에서 바이트를 읽습니다.

이 규칙을 고수함으로써 데이터를 수신하는 thread가 소비되지 않은 frame buffer를 절대 오버런하지 않고 소비하는 thread가 잘못된 데이터가 포함된 frame buffer에서 읽기를 시도하지 않을 것임을 보장합니다. 사실 FIFO의 바이트 수는 세트에서 유효한 frame buffers의 수를 나타냅니다.

쓴 바이트와 읽은 바이트의 값은 차이가 없으므로 실제로 이 4바이트의 메모리를 할당하고 데이터를 저장할 필요가 없습니다. FIFO의 핸드셰이크 메커니즘만이 역할을 합니다.

따라서 단락 4.6에 설명된 FIFO API는 다음과 같이 채택될 수 있습니다.

- 크기 매개변수를 frame buffers의 수로 사용하여 함수 fifo_init()을 호출합니다(`size`는 임의의 integer일 수 있음을 기억하십시오). fifo_init()은 FIFO에 대한 메모리를 할당하고 잠그며, 이는 절대 사용되지 않습니다(각 바이트는 frame buffer를 상징하기 때문에). 이 메모리 낭비는 무시할 수 있지만 코드의 관련 부분은 향후 혼동을 피하기 위해 제거할 수 있습니다.
- fifo_request_drain() 함수를 호출하여 읽을 frame buffer를 가져옵니다. info->position에는 사용할 frame buffer에 대한 인덱스가 포함됩니다(번호는 0부터 시작). 준비된 frame buffer가 없으면 fifo_request_drain()은 준비가 될 때까지 잠자기 상태가 됩니다.
- buffer에서 읽은 후 bytes_req=1로 함수 fifo_drained()를 호출합니다.
- fifo_request_write() 및 fifo_wrote() 함수는 thread가 frame buffers에 쓰는 것과 같은 방식으로 호출됩니다.
- FIFO_BACKOFF는 0으로 설정해야 합니다. frame buffers에서는 이 기능이 의미가 없습니다.

5.3 frames 떨어뜨리고 반복하기

데이터 소비자가 항상 충분히 빠르게 데이터를 수집하지 않을 수 있다는 점을 감안할 때 overflow 상태에 도달해서는 안 되는 image frames의 연속 소스의 경우를 살펴보겠습니다.

아이디어는 데이터 소스에서 frame buffers로 데이터를 전송하는 thread에서 차단을 방지하는 것입니다. 이를 달성하려면 들어오는 각 frame에 대해 다음 시퀀스를 반복해야 합니다.

- fifo_request_write() 함수를 호출하여 쓸 frame buffer를 찾습니다.
- info->position이 가리키는 frame buffer에 쓰기
- 쓰기가 끝나면 fifo_request_write() 함수를 다시 호출합니다. 이 함수 호출은 이전 호출 이후에 작성된 것으로 보고된 buffer가 없기 때문에 확실히 잠자기 상태가 아닙니다(block).
- fifo_request_write()가 1보다 큰 값을 반환했다면, fifo_wrote()(물론 req_bytes=1와 함께) 함수를 호출하십시오. fifo_request_write()에 대한 후속 함수 호출은 확실히 휴면하지 않을 것입니다(block). 스페어 buffer가 두 개 이상 있고 하나만 소비되었기 때문입니다. 사실, fifo_request_write()에 대한 다음 함수 호출은 다음 frame buffer를 선택하여 대체할 수 있습니다.
- 반면에 fifo_request_write()가 1만 반환하면 함수 fifo_wrote()를 호출하지 마십시오. 대신, 들어오는 데이터를 받기 위해 실행하는 다음 루프에서 현재 buffer를 다시 사용하거나 데이터 소스에서 특정 대상 없이 전체 frame을 배출하십시오.

이 사용은 차단을 방지하므로 fifo_request_write() 구현에서 while() 루프가 호출되지 않으므로 삭제할 수 있습니다. 관련 semaphore와 초기화 및 소멸 코드를 제거하여 추가 코드 축소가 가능합니다. 코드에 그대로 두는 것은 최소한의 영향을 미치므로 이 최적화는 대부분 코드를 읽을 수 있도록 유지하는 문제입니다.

FIFO에서 쓰는 thread에서 frames를 반복하는 유사한 접근 방식을 취할 수 있습니다. fifo_drained() 함수를 호출하기 직전에 fifo_request_drain() 함수를 다시 호출하고, 2 미만을 반환하는 경우 현재 frame을 반복합니다.

6 특정 프로그래밍 기술

6.1 Seekable streams

동기식 Xillybus stream은 seekable로 구성할 수 있습니다. stream의 위치는 FPGA의 application logic에 주소로 별도의 와이어로 표시되므로 demo bundle 및 예제 코드에서 볼 수 있듯이 메모리 어레이 또는 FPGA의 registers를 인터페이스하는 것은 간단합니다.

이 기능은 특히 FPGA에서 control registers를 설정하는 데 유용합니다. stream의 동기 특성은 FPGA의 register가 하위 수준 I/O 함수가 반환되기 전에 설정되도록 합니다.

다음 코드 스니펫은 `len` 바이트의 데이터를 메모리의 `address` 주소 또는 FPGA의 register space에 쓰는 방법을 보여줍니다. 이러한 두 변수가 미리 설정되어 있다고 가정합니다.

```
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`는 또한 쓰기 또는 읽기-쓰기를 위해 파일이 열린 open()에 대한 함수 호출에서 반환된 값과 쓸 데이터가 포함된 buffer를 가리키는 `buf`로 가정됩니다.

이 예는 3.3 단락에 표시된 예의 확장입니다.

이 코드에서 유일한 특별한 것은 주소를 설정하는 lseek()에 대한 함수 호출입니다. lseek() 함수를 호출할 때 SEEK_SET 옵션만 세 번째 인수로 사용해야 합니다.

후속 함수 호출은 I/O stream의 위치에 따라 주소를 업데이트하므로 함수 lseek()을 호출한 후 여러 순차 쓰기에 제한이 없습니다.

FPGA에서 16비트 또는 32비트 워드로 액세스되는 streams의 경우 lseek()에 지정된 주소는 각각 2 또는 4의 배수여야 합니다. FPGA에서 application logic에 제공된 주소는 항상 stream의 I/O 위치(초기에는 lseek()에 제공됨)를 각각 2 또는 4로 나눈 값으로 유지됩니다. 더 넓은 단어의 경우 동일한 로그 규칙이 적용됩니다.

tell() 함수 는 stream에서 올바른 위치(즉, 현재 주소)를 반환할 수 있지만 이 정보에 대한 신뢰할 수 있는 출처는 아닙니다. 확실하지 않으면 함수 lseek()을 다시 호출하십시오.

lseek()은 데이터를 읽는 것과 같은 방식으로 사용할 수 있습니다. 데모 애플리케이션 번들의 memwrite.c 및 memread.c(및 Getting started with Xillybus on a Linux host의 설명)를 참조하십시오.

6.2 양방향 streams 동기화

특정 응용 프로그램에서는 여러 streams를 아마도 반대 방향으로 동기화해야 합니다. 예를 들어, 무선 전송 시스템은 host에 구현되어 RF 수신기에 연결된 A/D converter에서 디지털 샘플을 수신할 수 있습니다. 마찬가지로 RF 송신기에 연결된 D/A converter로 디지털 샘플을 보낼 수 있습니다. 이러한 종류의 시나리오에서는 수신된 샘플과 관련하여 전송 시간을 알 수 있도록 전송할 디지털 샘플을 생성해야 하는 경우가 많습니다. 수신된 신호의 정확한 시간을 아는 것도 중요할 수 있습니다.

운 좋게도 간단한 FPGA logic로 구현할 수 있습니다. 이러한 솔루션 중 하나는 전송을 위한 첫 번째 샘플이 FPGA에 도착할 때까지 수신된 디지털 샘플을 무시하는 것입니다.

host는 FPGA에서 샘플을 읽기 위해 stream을 여는 것으로 시작합니다. 이 stream은 이 단계에서 유휴 상태입니다. FPGA가 수신 샘플을 삭제하기 때문입니다. 그런 다음 host는 FPGA에 전송할 샘플을 쓰기 위해 stream을 열고 데이터 쓰기를 시작합니다. 첫 번째 샘플이 FPGA에 도착하면 수신된 샘플을 무시하지 않고 host로 보내기 시작합니다.

결과적으로 FPGA에서 읽을 첫 번째 샘플은 FPGA에 쓰여진 첫 번째 샘플과 일치합니다. 따라서 host의 애플리케이션은 각 stream에서의 위치를 일치시키는 것만으로 수신된 샘플과 전송할 샘플의 timing을 일치시킬 수 있습니다. FPGA의 latency와 A/D 및 D/A의 지연을 보상하기 위해 약간의 수정이 필요할 수 있지만 이러한 latency는 일정하고 알려져 있습니다.

streams는 항상 연속성을 유지해야 합니다. 이를 달성하는 방법은 4 섹션에서 논의되었습니다.

이 솔루션은 전송과 수신 간의 상대적인 시간 관계를 유지하는 것으로 충분하다면 충분합니다. 샘플을 외부 이벤트 또는 다른 시간 기준과 동기화해야 하는 경우 샘플을 건너뛰는 동일한 원칙을 원하는 결과를 얻기 위해 필요에 따라 조정할 수 있습니다.

주어진 시간에 driver의 buffers에 얼마나 많은 데이터가 있는지 모니터링하는 방법은 Xillybus FPGA designer's guide의 "Monitoring the amount of buffered data" 섹션에서 설명합니다.

6.3 패킷 통신

일부 응용 프로그램에서는 data stream을 다양한 길이의 패킷으로 분할해야 합니다. 제안된 솔루션은 두 개의 별도 streams를 사용하며 채널을 통해 패킷 자체를 제출하기 시작할 때 데이터 발신자가 패킷 길이를 알 필요가 없습니다.

길이가 알려진 고정된 패킷의 사소한 경우는 하나의 stream에서 하나씩 전송함으로써 간단히 해결됩니다. 다른 쪽의 수신기는 각 패킷에 대해 고정된 수의 단어만 읽습니다. 이것은 video frame grabber 또는 video replay 애플리케이션의 일반적인 솔루션입니다.

다양한 길이의 패킷의 경우 upstream 애플리케이션을 살펴보겠습니다. 여기서 FPGA는 바이트 패킷을 host로 보냅니다. FPGA는 마지막 바이트가 도착했을 때만 패킷의 길이를 알고 있다고 가정해 봅시다.

FPGA 측(즉, 발신자 측)의 구현은 다음과 같습니다.

- FPGA는 패킷의 모든 바이트를 첫 번째 Xillybus stream에 씁니다.
- FPGA는 패킷의 첫 번째 바이트를 기록할 때 바이트 카운터를 재설정하고 기록하는 추가 바이트마다 바이트 카운터를 증가시킵니다.
- 패킷의 마지막 바이트가 기록되면 FPGA는 두 번째 Xillybus stream에 카운터 값을 보냅니다. 패킷의 길이(-1)를 포함합니다.

이 솔루션의 중요한 속성은 FPGA가 패킷을 보내기 전에 전체 패킷을 저장할 필요가 없다는 것입니다. 데이터가 도착할 때만 전달합니다.

host의 사용자 애플리케이션은 다음과 같이 루프를 실행합니다.

- 다음 패킷의 바이트 수를 포함하는 두 번째 stream에서 한 단어를 읽습니다.
- 필요한 경우 요청된 크기의 buffer에 대한 메모리를 할당합니다.
- 첫 번째 stream에서 패킷 전용 buffer로 주어진 바이트 수를 읽습니다.

host는 데이터에 액세스하기 전에 읽을 바이트 수를 가져오지만 FPGA는 이를 역순으로 streams에 썼습니다. 별도의 Xillybus streams를 사용하면 이러한 반전이 가능합니다.

패킷이 host에서 FPGA로 전송될 때도 유사한 배열이 적용됩니다. 2개의 streams를 사용하는 원칙은 하나는 데이터용이고 다른 하나는 바이트 수입니다. FPGA의 application logic은 이제 다른 stream에서 데이터를 가져오기 전에 하나의 stream에서 바이트 수를 읽을 수 있습니다.

이 배열은 또한 데이터가 아닌 stream의 다른 메타데이터를 전달하도록 확장할 수 있습니다. 예를 들어 패킷의 대상 또는 일부 네트워크의 라우팅(첫 바이트가 도착할 때 알 수 없는 경우가 있음).

6.4 hardware interrupts 에뮬레이션

소규모 마이크로컨트롤러 프로젝트에서는 hardware interrupts를 사용하여 소프트웨어에 문제가 발생했으며 소프트웨어가 조치를 취해야 한다고 알리는 것이 일반적입니다. 소프트웨어가 Linux에서 userspace 프로세스로 실행될 때 hardware interrupts는 의심의 여지가 없으며, 모든 비동기 이벤트와 마찬가지로 software interrupts도 다루기 쉽지 않습니다.

Xillybus 기반 시스템에 대해 제안된 솔루션은 메시지 전달을 위해 특수 stream을 할당하는 것입니다. 가장 간단한 형태로 hardware interrupt는 해당 전용 stream에서 단일 바이트를 전송하여 에뮬레이트됩니다.

host 측에서 userspace application은 stream에서 데이터 읽기를 시도합니다. 결과적으로 “interrupt”가 신호를 받지 않으면 바이트가 도착하고 깨울 때까지 애플리케이션이 잠자기(blocking)됩니다. 애플리케이션은 이벤트를 처리한 다음 전용 stream에서 다른 바이트를 읽으려고 시도하므로 필요한 경우 다시 잠자기 상태가 되는 식입니다.

기본 응용 프로그램과 interrupt routine 간의 적절한 상호 작용을 달성하기 위해 이 전용 stream은 별도의 software thread 또는 프로세스에서 읽을 수 있습니다. 이 배열에서 메인 코드는 전용 메시지 stream에서 읽는 thread와 관계없이 흐르고 후자는 보낸 메시지에 따라 휴면 및 웨이크업합니다.

이 방법의 변형은 전송된 바이트 값을 사용하여 에뮬레이트된 interrupt의 특성에 대한 정보를 전달합니다. 또한 구현에서 의미가 있는 경우 각 메시지는 단일 바이트보다 길 수 있습니다.

이 방법은 logic 리소스의 낭비로 보일 수 있지만 Xillybus는 원래 이와 같은 솔루션을 합리적으로 만들기 위해 추가된 각 stream에 대해 logic을 많이 소비하지 않도록 설계되었습니다.

6.5 Timeout

특정 응용 프로그램에서는 I/O 작업이 blocking 상태로 유지될 수 있는 시간을 제한하고자 합니다. 특히 데이터 흐름이 중지되는 일부 하드웨어 오류가 발생할 가능성이 있는 경우에 그렇습니다.

Xillybus 자체는 데이터가 이러한 방식으로 중지되는 이유가 결코 아님을 확인하기 위해 광범위하게 테스트되었지만 데이터 소스 및 데이터 소비자는 다양한 이유로 중지될 수 있습니다.

이 문제를 해결하는 덜 선호되는 방법은 select() 또는 pselect() 기능을 사용하는 것입니다. 여러 file descriptors를 기다릴 때 필요하지만 timeout 기능도 있습니다. 중요하지 않은 인터페이스가 버그의 원인이 될 수 있으므로 이러한 기능을 사용하는 것은 권장되지 않습니다. 특히 timeout가 잡아야 하는 특별한 경우에는 그렇습니다.

보다 자연스러운 방법은 Linux의 alarm 기능을 사용하는 것입니다. 프로세스별 timeout 메커니즘으로, 만료될 때 signal(software interrupt)를 프로세스에 보냅니다. signal은 즉시 제어를 반환하기 위해 잠자기 상태인 read() 또는 write() 함수 호출을 강제 실행한다는 점을 기억하십시오(3.2 및 3.3 단락 참조). 이 함수는 음수 값으로 반환되고 errno는 EINTR로 설정됩니다. 이전 예에서 이러한 인터럽트는 방해에 불과했지만 그럼에도 불구하고 timeout를 구현하는 데 유용합니다.

모든 프로세스는 기능과 관련이 없는 여러 signals를 수신할 수 있습니다. signal을 수신하는 것은 그 자체로 timeout 상태를 나타내는 것이 아닙니다. 말할 수 있는 방법은 여러 가지가 있지만 가장 안전한 방법은 그 질문에 전혀 의존하지 않는 것입니다. I/O 작업이 일정 시간 이상 걸리면 timeout입니다. 따라서 가장 간단한 전략은 다음 예제에서와 같이 시간을 측정하는 것입니다. 이는 단락 3.2에서 함수 read()를 호출하는 것을 기반으로 합니다.

이 예에서 include files의 일반적인 목록은 약간 깁니다.

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

이 예와 관련하여 다음 선언이 필요합니다.

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

데이터 읽기를 위한 while 루프는 이제 다음과 같이 시작됩니다.

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

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

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

시간은 clock_gettime()로 함수 read()를 호출하기 전과 후에 측정됩니다. 이것은 단조 시간 측정에 액세스할 수 있기 때문에 시간 차이 측정에 선호되는 기능입니다(시스템 유틸리티에 의해 수정되는 system clock와 반대). 이 함수는 `-lrt` 플래그가 gcc의 인수에 추가되어야 하므로 필요한 라이브러리를 로드해야 합니다.

alarm()에 대한 함수 호출은 2초 후에 signal을 요청합니다(인수는 초 수). 각 프로세스에는 alarm timer가 하나만 있으므로 동일한 timer의 다른 사용을 무시하지 않도록 주의해야 합니다(예: 일부 Linux 구현에서 sleep()).

이 코드는 다음과 같습니다.

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

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

시차를 계산하여 `elapsed`에 저장합니다. 이 간단한 예에서 단어 길이 이식성 문제를 피하기 위한 double-precision floating point 변수입니다. 그러나 이것은 integer로도 가능합니다.

조건은 간단합니다. 시간 측정 사이에 2초 이상이 경과하면 timeout입니다. read()가 반환된 이유는 검사하지 않습니다. signal이거나 데이터가 결국 도착했지만 너무 늦었을 수 있습니다. 두 경우 모두 error입니다.

alarm()에 대한 함수 호출은 처음 측정이 발생한 후에 이루어졌으므로 timeout는 시간 차이를 최소 2초 이상 만들도록 보장됩니다.

while 루프는 이전과 같이 계속됩니다.

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

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

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

위에서 볼 수 있듯이 signals는 여전히 무시됩니다. timer가 프로세스를 깨우면 시간 차이가 timeout 상태를 나타내고 종료되어야 합니다.

timeout를 구현하는 이 방법은 UNIX signal을 기반으로 하므로 multi-threaded 환경에서 복잡한 문제가 됩니다. 여러 threads가 배포된 경우 그 중 하나를 다른 watchdog로 만드는

것이 가장 쉽습니다.

또한 위의 예에서 timeout로 인해 프로세스가 종료되므로 이 작업을 수행하는 signal handler로 구현하기가 더 쉽습니다. 위의 방법은 실행 중인 프로세스 내에서 수정 대응을 수행할 때 더 적합합니다.

timeout 간격의 정밀도를 높이려면 대신 setitimer()를 사용하는 것이 좋습니다.

6.6 Coprocessing/ Hardware acceleration

Coprocessing(*hardware acceleration*라고도 함)는 애플리케이션이 logic fabric의 유연성을 활용하여 주어진 processor보다 에너지 소비가 적거나 더 효율적으로 특정 작업을 더 빠르고 저렴하게 수행할 수 있도록 하는 기술입니다. 동기가 무엇이든 효율적인 데이터 전송 흐름은 coprocessing을 적합한 솔루션으로 만드는 데 중요합니다.

coprocessing 기반 애플리케이션의 데이터 흐름은 일반적인 프로그래밍 데이터 흐름과 근본적으로 다르다는 것을 깨닫는 것이 중요합니다. 이 차이를 설명하기 위해 floating point 표현에서 숫자의 제곱근을 계산해야 하는 컴퓨터 프로그램을 예로 들어 보겠습니다.

프로그래머의 직접적인 방법은 숫자를 sqrt()에 인수로 전달하고 호출하고 함수가 반환될 때까지 기다리는 것입니다.

대신 FPGA의 logic fabric에서 제곱근을 계산하려고 한다고 가정합니다. 일반적인 *실수* 는 sqrt()을 계산을 위한 값을 FPGA로 보내고 완료될 때까지 기다린 다음 결과와 함께 반환하는 특수 함수로 대체하는 것입니다. 이것은 실제로 sqrt()의 간단한 드롭인 대체품이지만 원래 sqrt()보다 *느리고* 효율성이 떨어질 가능성이 가장 높습니다. 데이터가 bus를 통해 양방향으로 이동하는 데 걸리는 시간에 FPGA가 계산을 수행하는 데 걸리는 시간은 sqrt()에 필요한 processor cycles보다 훨씬 더 길 것입니다. 데이터 흐름이 올바르게 설계된 경우 FPGA에서 제곱근을 계산하는 것이 훨씬 빠를 수 있습니다.

bus와 FPGA의 logic이 부과한 latencies를 극복하기 위해서는 소프트웨어를 재정비할 필요가 있다. 특히, 하나의 thread가 있는 프로그램의 작업은 두 개 이상의 threads(또는 프로세스)로 분할되어야 합니다. 여러 threads가 가능하지 않거나 바람직하지 않은 경우 다른 프로그래밍 기술을 사용하여 multi-threading의 동작을 모방할 수 있지만 프로그래밍 패러다임은 그럼에도 불구하고 multi-threaded입니다.

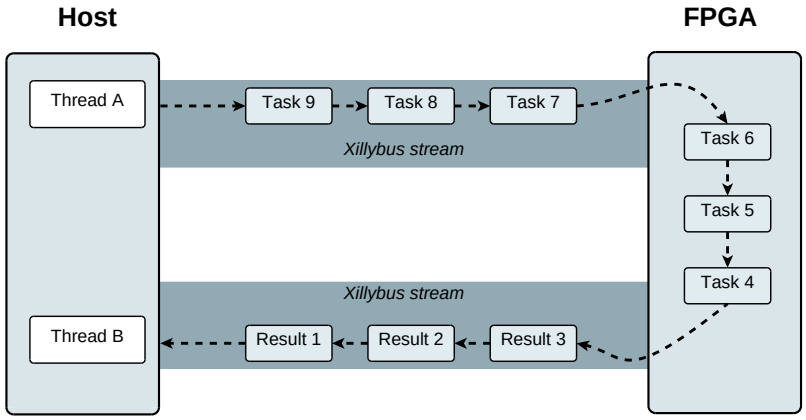
sqrt()의 예로 돌아가서 이 함수에 대한 호출은 두 개의 threads로 나뉩니다. 첫 번째 thread는 제곱근 계산을 위한 데이터를 하드웨어(또는 연산 요청을 나타내는 다른 형식의 데이터 구조)로 보냅니다. 두 번째 thread는 하드웨어에서 결과를 수신하고 알고리즘의 해당 지점에서 처리를 계속합니다.

이것은 단일 데이터 조각을 볼 때 그다지 의미가 없어 보이지만 coprocessing에 대한 동기는 처리해야 할 데이터 항목이 많다는 것을 의미합니다. 따라서 첫 번째 thread는 계산을

위한 데이터 *흐름* 을 보내고 두 번째 thread는 결과 *흐름* 을 받습니다.

*pipelining*의 이 기술은 하드웨어의 latency의 영향을 최소화합니다. threads 중 어느 것도 이 latency의 시간을 효과적으로 기다리지 않기 때문입니다. 대신 latency는 두 threads 사이에 있는 처리 항목의 양에 영향을 주지만 *처리량* 은 두 threads와 FPGA logic의 처리 능력에만 의존합니다.

다음 개념 그림은 아이디어를 요약한 것입니다.



sqrt()의 가속 계산은 비교적 간단한 예이지만 coprocessing을 활용하는 데 있어 많은 문제를 다룹니다. 거의 항상 컴퓨터 프로그램의 많은 부분을 다시 작성하여 모든 것이 pipeline의 데이터 흐름에 의해 구동되도록 해야 합니다.

알아야 할 또 다른 문제는 Xillybus가 read() 및 write()와 함께 작동하기 때문에 FPGA를 향해 stream에 쓰기 전에 계산을 위해 여러 데이터 항목을 그룹화하는 것이 유리할 수 있다는 것입니다. 마찬가지로 각 read() 호출에서 둘 이상의 결과 항목을 읽으려고 하면 성능이 향상될 수 있습니다. 이에 대한 근거는 read() 및 write()가 특정 오버헤드가 있는 system calls라는 것입니다. 데이터 요소가 작고 빠른 속도로 전송되는 경우 이러한 system call의 오버헤드는 상당할 수 있습니다. sqrt()의 경우가 이에 대한 좋은 예입니다. double float의 길이는 일반적으로 8바이트입니다. 이 길이의 I/O system call은 매우 비효율적이므로 단일 system call에 대해 여러 double float 요소를 연결하면 차이가 생깁니다.

모든 애플리케이션이 일정한 길이의 데이터 청크를 포함하는 것은 아니라는 점도 언급할 가치가 있습니다. 예를 들어, 임의의 문자열의 hashes(예: SHA1)를 계산하기 위해 coprocessing을 사용하면 다른 길이로 처리하기 위한 데이터 요소가 포함될 수 있습니다. 섹션 6.3는 이에 대한 솔루션을 제안합니다.

A

내부: streams 구현 방법

A.1 소개

Xillybus를 사용하는 데 구현 세부 사항에 대한 이해가 필요하지는 않지만 일부 설계자는 호기심이나 특정 솔루션의 적격성을 확인하기 위해 내부에서 어떤 일이 발생하는지 아는 것을 선호합니다.

이 섹션에서는 DMA buffers를 기반으로 연속 streams를 생성하기 위해 구현된 주요 기술을 간략하게 설명합니다. PCIe / AXI용 Xillybus에는 적용되지만 다른 메커니즘을 사용하는 XillyUSB에는 적용되지 않습니다.

Xillybus 설계 방법의 목표는 기본 메커니즘을 사용자에게 투명하게 만드는 것이며 대부분은 이를 인식할 이유가 없습니다. Xillybus를 IP core로 사용하기 위해 필요하지 않을 가능성이 매우 높기 때문에 아래의 기술적인 세부 사항으로 내려갈 때 이것을 염두에 두십시오. 이 부분은 작동 방식에 관한 것이며 사용자가 알아야 할 사항은 적습니다.

아래에는 upstream 흐름에 대한 섹션과 downstream에 대한 섹션의 두 가지 주요 섹션이 있습니다. 유사한 기술이 양방향으로 사용되기 때문에 한 섹션의 많은 부분이 다른 섹션의 반복입니다.

단순함을 위해 설명은 달리 명시되는 경우를 제외하고 비동기식 streams에 중점을 둡니다. end-of-file 신호와 non-blocking I/O 옵션은 여기에서 논의되지 않습니다.

A.2 “Classic” DMA 대 Xillybus

전통적으로 하드웨어와 소프트웨어 간의 데이터 전송은 고정된 크기의 여러 buffers 형식을 취합니다. 데이터는 완전히 채워질 수도 있고 채워지지 않을 수도 있는 고정 길이의 buffers로 구성됩니다. buffer가 준비될 때마다 일종의 신호가 다른 쪽으로 전송됩니다. 예를 들어, 하드웨어가 buffer에 쓰기를 마치면 interrupt를 processor로 보내 데이터가 처리

할 준비가 되었음을 소프트웨어에 알릴 수 있습니다. 소프트웨어는 데이터를 소비하고 일반적으로 일부 메모리 매핑된 register에 쓰는 방식으로 buffer에 다시 쓸 수 있음을 하드웨어에 알립니다. 일반적으로 양측은 round-robin 방식으로 buffers에 액세스합니다.

Xillybus는 FPGA와 소프트웨어 쪽 모두에서 사용자 인터페이스에 대한 지속적인 stream 전송을 제공합니다. 내부적으로 Xillybus는 DMA buffers 세트와 함께 전통적인 round-robin 패러다임을 사용합니다.

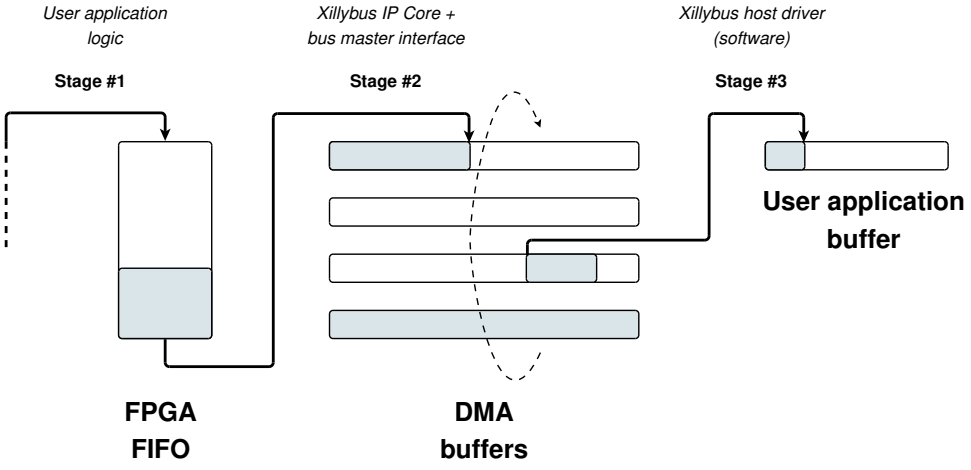
그러나 아래에 설명된 기술은 사용자가 기본 데이터 전송의 존재를 무시할 수 있도록 연속 stream의 환상을 만드는 데 사용됩니다. 특히, 응용 프로그램이 고정된 크기의 청크 단위로 데이터를 보내는 것으로 구성되어 있더라도 아래 설명과 같이 DMA buffers의 크기를 응용 프로그램 데이터와 일치시킬 필요가 없습니다.

A.3 FPGA host(upstream)

A.3.1 개요

아래 그림은 FPGA host (upstream) 방향의 흐름을 나타냅니다. 음영 처리된 영역은 해당 스토리지 요소에서 아직 소비되지 않은 데이터를 나타냅니다.

이 예에서는 IP Core Factory에서 그 수를 구성할 수 있음에도 불구하고 4개의 DMA buffers가 표시됩니다.



데이터는 다음에 자세히 설명하는 것처럼 3단계로 host로 흐릅니다.

A.3.2 #1 단계: Application logic 중급 FIFO

FPGA의 user application logic은 user application logic와 Xillybus IP core 사이를 연결하는 FIFO로 데이터를 푸시합니다. overflow를 피하기 위해 FIFO의 “full” 신호를 존중하는 것을 제외하고는 언제 또는 얼마나 많은 데이터가 푸시되는지에 대한 요구 사항은 없습니다.

A.3.3 #2 단계: 중급 FIFO DMA buffer

이 단계에서 Xillybus IP core는 FIFO의 데이터를 host의 RAM에 있는 DMA buffer로 복사합니다. 이를 달성하기 위해 core는 host의 processor의 개입 없이 host의 메모리에 직접 데이터를 쓰기 위해 일부 bus master 인터페이스(PCIe, AXI4 등)를 사용합니다.

DMA buffers의 풀은 host의 RAM에 할당됩니다. 각 DMA buffer의 수명 주기는 많은 유사한 설정과 같습니다. 처음에는 모든 DMA buffers가 비어 있고 개념적으로 하드웨어에 속합니다. 하드웨어는 round-robin 방식으로 buffers에 데이터를 씁니다. 특정 buffer에 쓰기를 완료하면 host에 buffer를 사용할 준비가 되었음을 알립니다(buffer는 host로 전달됨). 그 후 다음 buffer에서 계속 작성합니다. 그런 다음 host는 전달된 buffer의 데이터를 소비한 후 buffer를 다시 쓸 수 있음을 하드웨어에 알립니다(host는 buffer를 하드웨어로 반환).

스테이지 #2의 데이터 흐름은 FIFO의 “empty” 신호와 DMA buffers 풀의 공간 가용성에 의해 제어됩니다. Xillybus IP core가 FIFO에서 낮은 “empty” 신호를 감지하고 일부 DMA buffer에 공간이 남아 있으면 FIFO에서 데이터를 가져와 DMA buffer에 씁니다. FIFO가 다시 비어 있거나 DMA buffer에 공간이 없으면 IP core의 내부 상태 머신은 일시적으로 데이터 가져오기를 중지한 다음 DMA buffer에서 중단된 위치에서 계속합니다.

데이터 흐름이 중단되는 동안 IP core는 다른 활동으로 바쁠 수 있습니다. 예를 들어 다른 stream을 대신하여 데이터를 복사하는 것입니다(즉, 다른 중간 FIFO를 배수). 결과적으로 FIFO가 “empty” 신호를 로우로 변경하는 시간과 데이터 가져오기 재개 사이에 임의의 지연이 있을 수 있습니다. 이 지연은 다양하지만 전반적으로 IP core는 512워드의 FIFO가 overflow 상태에 도달하지 않도록 보장합니다(평균 속도가 제한 내에 있는 한).

각 DMA buffer는 host로 넘기기 전에 완전히 채워지거나 부분적으로 채워진 host에 제출될 수 있습니다. 부분적으로 채워진 buffer를 넘겨주는 조건은 나중에 자세히 설명됩니다(섹션 A.3.5). 소프트웨어 동작에 대한 약간의 이해가 필요하기 때문입니다.

동기식 streams의 경우는 Xillybus IP core가 중간 FIFO에서 데이터를 가져오기 전에 특정 양의 데이터에 대한 명시적 요청을 기다린다는 점을 제외하면 매우 유사합니다.

A.3.4 스테이지 #3: DMA buffer에서 사용자 소프트웨어 애플리케이션으로

이 단계는 read() system calls(또는 Microsoft Windows의 해당 IRP)에 응답하여 host의

Xillybus driver에서 구현됩니다. 잘 정립된 API에 따르면 read() 요청에는 user application에서 제공하는 buffer와 읽을 최대 바이트 수인 buffer의 크기가 포함됩니다. 함수 호출은 최대 바이트 수(완료) 이하를 읽은 후 반환될 수 있습니다.

driver는 read() 요청의 완전한 이행을 허용하기 위해 DMA buffers에서 소비하기에 충분한 데이터가 있는지 확인하기 위해 전달된 DMA buffers를 확인하는 것으로 시작합니다. 그렇다면 데이터를 사용자의 buffer로 복사하고 DMA buffers를 하드웨어로 반환하고 system call에서 반환할 수 있습니다.

그렇지 않으면 read() 함수 호출을 위한 표준 API를 사용하면 driver가 요청된 바이트 수보다 적은 수로 반환하거나 임의의 기간 동안 대기(절전)할 수 있습니다. driver는 데이터가 거의 없는 상태에서 너무 자주 반환하지 않도록 설계되었으며(각각 데이터가 거의 없는 많은 read() 함수 호출이 발생하여 CPU 주기를 낭비할 수 있음) 불필요한 latency도 방지합니다. 딜레마는 read() 함수 호출에 필요한 것보다 DMA buffers에 데이터가 적은 경우 수행할 작업입니다.

선택한 전략은 더 많은 데이터를 위해 최대 10 ms까지 기다린 다음 사용 가능한 모든 것으로 반환하는 것입니다(또는 표준 API에서 요구하는 대로 사용 가능한 데이터가 없는 경우 무기한 대기). 결과적으로 응답 시간이 상당히 빠르지만 read() 함수 호출자가 항상 사용 가능한 데이터보다 많은 데이터를 요청하는 경우 오버헤드는 초당 100개의 read() 함수 호출로 제한됩니다.

이것은 read() 함수 호출에 반드시 10 ms의 latency가 있어야 한다는 말은 아닙니다. user space application이 준비해야 하는 바이트 수를 미리 알고 있는 경우 해당 수 이상을 요청할 수 없습니다. 그렇게 함으로써 마이크로초 단위의 latency를 보장합니다.

그러나 까다로운 부분이 있습니다. host는 전달된 DMA buffers에 대해 알고 있지만 host가 인식하지 못하는 부분적으로 채워진 DMA buffer가 있을 수 있습니다. 따라서 부분적으로 채워진 DMA buffer를 고려하면 실제로 read() 함수 호출을 완전히 수행하기에 충분한 데이터가 있을 수 있습니다.

이 경우를 적절하게 처리하기 위해 driver는 누락된 바이트 수가 부분적으로 채워진 buffer에 맞는지 여부를 확인합니다. 이것이 사실이라면 하드웨어에 얼마나 많은 데이터가 충분한지 알려줍니다. 그런 다음 driver는 10 ms 대기를 시작합니다. 이것은 하드웨어가 부분적으로 채워진 buffer를 즉시 보낼 수 있는 기회를 제공합니다. 만약 그것이 정말로 read() 함수 호출을 완전히 완료할 수 있게 한다면 말입니다.

부분적으로 채워진 buffer가 필요한 양(즉시 가능)에 도달하면 하드웨어가 host로 이를 넘겨 read() 함수 호출을 즉시 완료합니다.

10 ms 기간이 끝나면 driver는 사용 가능한 데이터만큼 반환합니다. 데이터가 전혀 없으면 driver는 하드웨어에 요청을 보내 부분적으로 채워진 buffer를 전달합니다. 목적은 10 ms 기간이 이미 끝났기 때문에 데이터가 있는 즉시 반환하는 것입니다.

모든 상황에서 DMA buffer가 완전히 소모되면 driver는 이를 하드웨어로 반환합니다(즉, 하드웨어에 다시 쓸 수 있음을 알립니다).

동기 streams에 대한 몇 마디: read() 함수 호출이 호출될 때 DMA buffers에서 데이터를 사용할 수 없다는 점을 제외하면 흐름은 원칙적으로 동일합니다. 하드웨어는 지시가 있을 때만 FPGA의 FIFO에서 데이터를 복사할 수 있기 때문입니다. 따라서 동기식 streams에 대한 read() 함수 호출에는 복사해야 하는 데이터의 양을 하드웨어에 알리는 것이 포함됩니다. 대기 메커니즘은 동일하게 유지됩니다: 먼저 10 ms, 그 다음 부분적으로 채워진 buffer가 필요합니다.

A.3.5 일부만 채워진 buffers 인계 조건

위에서 부분적으로 채워진 buffers를 인계하는 경우를 유추할 수 있으며 편의를 위해 여기에 나열됩니다.

일반적인 규칙은 하드웨어에 이러한 조기 제출로 인해 read() 함수 호출이 즉시 반환된다는 정보를 받은 경우 부분 buffer가 host로 넘겨지는 것입니다. 이 경우 다음 세 가지 조건 중 하나에서 발생합니다.

- host는 현재 read() 함수 호출을 처리하고 있으며 현재 부분적으로 채워진 buffer가 전달될 때 완전히 수행됩니다.
- read() 함수 호출은 0바이트이며 시간 제한(즉, 10 ms)에 도달했습니다.
- 동기식 streams에만 해당: 하드웨어가 host에서 요청한 양의 가져오기를 완료했을 때.

FIFO가 비어 있을 때 그 자체로 DMA buffer 제출의 이유는 *아닙니다* .

A.3.6 예

다음과 같은 8비트 비동기식 stream의 간단한 경우를 살펴보겠습니다. stream이 데이터를 포함하지 않고 시작하고 그 후에 FIFO가 단일 요소(즉, 1바이트)로 채워진다고 가정합니다. 그런 다음 host의 응용 프로그램은 read() 함수를 호출하여 1바이트를 요청합니다. 다음은 가능한 일련의 이벤트입니다.

- Xillybus IP core는 낮은 “empty” 신호를 감지하여 FIFO에서 단일 바이트를 가져온 다음 다시 비게 됩니다.
- 바이트는 DMA로 DMA buffer의 첫 번째 위치에 기록됩니다. buffer가 가득 차지 않았으므로 host에 알림이 표시되지 않습니다.

- read() 함수 호출이 host에서 호출되어 1바이트를 요청합니다.
- driver에는 데이터를 가져올 DMA buffer가 없습니다. 데이터(1바이트)를 포함하는 유일한 DMA buffer는 하드웨어에만 알려져 있습니다.
- driver는 필요한 데이터 양이 DMA buffer의 크기보다 작다는 것을 감지하여 하드웨어에 부분적으로 채워진 buffer를 넘겨주도록 지시합니다(적어도 1바이트가 있는 경우).
- driver는 10 ms 절전 모드를 시작하여 어떤 일이 일어나기를 기다립니다.
- 하드웨어는 부분적으로 채워진 buffer를 host로 넘기면서 즉시 응답합니다.
- driver는 즉시 깨어나서 요청된 1바이트를 read() 함수 호출과 함께 제공된 buffer에 복사하고 반환합니다.

이 간단한 예는 데이터 크기가 DMA buffer보다 훨씬 작음에도 불구하고 read() 함수 호출이 사실상 즉시 반환되는 방법을 보여줍니다.

한 가지 작은 차이점이 있는 예제를 다시 살펴보겠습니다. read() 함수 호출은 FIFO에 1바이트만 기록되더라도 2바이트를 요청합니다. 순서는 다음과 같습니다.

- Xillybus IP core는 낮은 “empty” 신호를 감지하여 FIFO에서 단일 바이트를 가져온 다음 다시 비게 됩니다.
- 바이트는 DMA로 DMA buffer의 첫 번째 위치에 기록됩니다. buffer가 가득 차지 않았으므로 host에 알림이 표시되지 않습니다.
- read() 함수 호출은 host에서 호출되며 *2* 바이트를 요청합니다.
- driver에는 데이터를 가져올 DMA buffer가 없습니다. 데이터(1바이트)를 포함하는 유일한 DMA buffer는 하드웨어에만 알려져 있습니다.
- driver는 필요한 데이터 양이 DMA buffer의 크기보다 작다는 것을 감지하여 하드웨어에 부분적으로 채워진 buffer를 넘겨주도록 지시합니다(적어도 *2바이트* 가 있는 경우).
- driver는 10 ms 절전 모드를 시작하여 어떤 일이 일어나기를 기다립니다.
- 하드웨어는 DMA buffer에 1바이트만 있기 때문에 아무 작업도 수행하지 않지만 2개가 요청되었습니다.
- driver는 10 ms 이후에 깨어나고 아무것도 없습니다. 비어 있지 않는 한 부분적으로 채워진 buffer를 가능한 한 빨리 넘겨주도록 하드웨어에 요청을 보냅니다.

- 하드웨어는 부분적으로 채워진 buffer를 host로 넘기면서 즉시 응답합니다.
- driver는 즉시 깨어나서 요청된 1바이트를 함수 호출자의 buffer에 복사하고 반환합니다.

이 두 번째 예는 실제로 하나만 있을 때 2바이트를 요청한 결과를 보여줍니다. 함수 호출은 10 ms 이후에만 1바이트로 반환됩니다. 그러나 이 지연은 대부분의 실제 시나리오에서 눈에 띄지 않습니다.

A.3.7 실용적인 결론

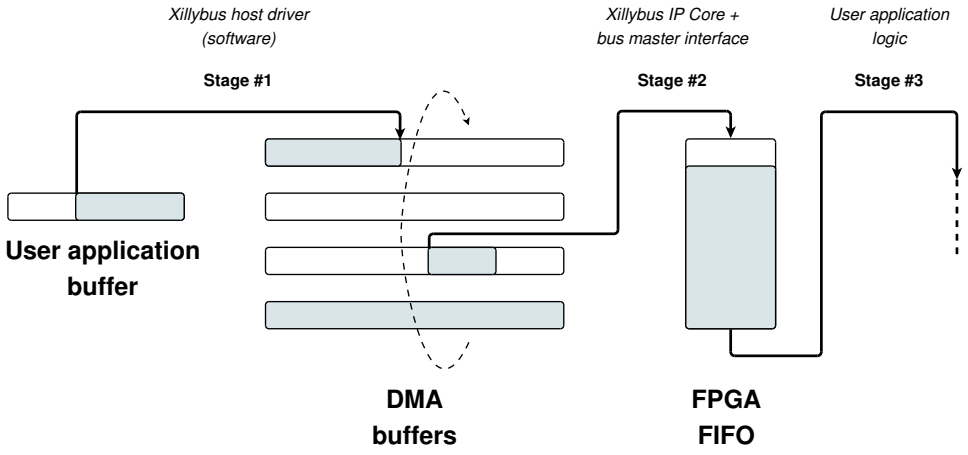
- 응용 프로그램 수준 데이터가 항상 N 바이트 청크로 구성되더라도 DMA buffer 크기를 어떤 식으로든 조정할 이유가 없습니다. user application software는 read() 함수 호출이 필요에 따라 정확히 데이터 양을 요청하도록 해야 하며, 부분적인 buffer 메커니즘은 데이터가 FPGA의 FIFO에 매우 낮은 latency로 푸시될 때 함수 호출이 반환되는지 확인합니다.
- 연속적인 streams 데이터의 경우에도 작은 buffers로 read() 함수 호출을 수행하여 latency를 줄일 수 있지만 운영 체제의 추가 오버헤드가 발생합니다. DMA buffers의 크기에 관계없이 latency는 데이터 속도와 read() 함수 호출에서 요청된 바이트 수에만 의존합니다. read() 함수 호출을 완전히 수행할 수 없는 경우 read() 함수 호출이 최대 10 ms까지 계속 대기하므로 DMA buffer 크기를 줄이는 것은 도움이 되지 않습니다.
- 10 ms가 허용 가능한 latency인 경우 반환할 데이터가 전혀 없는 경우를 제외하고 read() 함수 호출이 이 기간 후에 반환되도록 보장되므로 최적화의 의미가 없습니다.

A.4 Host FPGA(downstream)

A.4.1 개요

아래 그림은 host에서 FPGA(downstream) 방향으로의 데이터 흐름을 보여줍니다. 음영 처리된 영역은 해당 스토리지 요소에서 아직 소비되지 않은 데이터를 나타냅니다.

이 예에서는 IP Core Factory에서 그 수를 구성할 수 있음에도 불구하고 4개의 DMA buffers가 표시됩니다.



이전과 마찬가지로 데이터는 host에서 FPGA로 3단계로 흐릅니다.

A.4.2 #1 단계: DMA buffer에 사용자 소프트웨어 적용

이 단계는 write() system calls(또는 Microsoft Windows의 대응 IRP)에 응답하여 Xillybus의 driver(host)에서 구현됩니다. 잘 정립된 API에 따르면 write() 함수 호출 요청에는 사용자 애플리케이션에서 제공하는 buffer와 쓸 수 있는 최대 바이트 수인 buffer의 크기가 포함됩니다. 함수 호출은 최대 바이트 수(완전한 이행) 이하를 기록한 후 반환될 수 있습니다.

DMA buffers의 풀은 host의 RAM 메모리에 할당됩니다. 각 DMA buffer의 수명 주기는 많은 유사한 설정과 같습니다. 처음에는 모든 DMA buffers가 비어 있고 개념적으로 host에 속합니다. host는 round-robin 방식으로 buffers에 데이터를 씁니다. 특정 buffer에 쓰기를 완료하면 buffer를 사용할 준비가 되었음을 하드웨어에 알립니다(buffer는 하드웨어로 인계됨). 그 후 다음 buffer에서 계속 작성합니다. 그런 다음 하드웨어는 buffer의 데이터를 소비한 후 host에 buffer를 다시 쓸 수 있음을 알립니다(하드웨어는 buffer를 host로 반환함).

Xillybus의 driver는 가능한 한 많은 데이터를 DMA buffers에 복사하려고 시도하여 write() 함수 호출에 응답합니다. DMA buffer가 완전히 채워지면 하드웨어로 전달됩니다. 즉, host는 하드웨어에 buffer를 사용할 수 있음을 알리고 하드웨어가 buffer를 host로 반환하기 전에 다시 쓰지 않도록 보장합니다.

driver가 DMA buffer 공간이 부족해지기 전에 최소 1바이트를 쓸 수 있었다면 write() 함수 호출은 쓰여진 바이트 수와 함께 반환됩니다. 그렇지 않으면 DMA buffer가 쓰기에 사용 가능해질 때까지 무기한 대기(예: “blocking”)한 다음 가능한 한 많은 데이터를 DMA buffer에 쓰고 반환합니다.

DMA buffer가 부분적으로 채워진 경우 write() 함수 호출이 끝날 때 하드웨어로 전달되지

않으므로 하드웨어가 인식하지 못하는 데이터가 하나의 DMA buffer에 있을 수 있습니다. "flush" 작업은 부분적으로 채워진 buffer를 넘겨주며 다음 네 가지 경우 중 하나에서 발생합니다.

- 쓸 바이트가 0인 write() 함수 호출로 인해 발생하는 명시적 flush입니다. 이 write() 함수 호출은 즉시 반환됩니다(즉, FPGA가 데이터를 소비할 때까지 기다리지 않음).
- 자동 flush는 마지막 write() 함수 호출 후 10 ms가 시작됩니다.
- 파일을 닫으면 flush가 발생합니다. 이 시나리오에서 close() 함수 호출은 반환되기 전에 FPGA가 데이터를 완전히 소비할 때까지 최대 1초 동안 기다립니다.
- 동기식 streams에서 write()에 대한 모든 함수 호출은 flush()로 끝나며 flush()는 데이터가 FPGA에서 완전히 소비될 때까지 무기한 대기합니다.

길이가 0인 buffer가 있는 write() 함수 호출은 명시적 flush를 강제 실행하여 작성된 모든 데이터를 FPGA에서 사용할 수 있도록 합니다. 그러나 FPGA에서 데이터를 소비하는 시점에 대한 표시는 애플리케이션 소프트웨어에 제공하지 않습니다. 이러한 동기화가 필요한 경우 동기식 stream을 사용해야 합니다.

A.4.3 #2 단계: DMA buffer 중급 FIFO

이 단계에서 Xillybus IP core는 host의 RAM에 있는 DMA buffers에서 FPGA의 FIFO로 데이터를 복사합니다. 이를 달성하기 위해 core는 host의 processor의 개입 없이 host의 메모리에서 직접 데이터를 읽기 위해 일부 bus master 인터페이스(PCIe, AXI4 등)를 사용합니다.

스테이지 #2의 데이터 흐름은 FIFO의 "full" 신호와 FPGA에 속하는 DMA buffers 풀의 데이터 가용성에 의해 제어됩니다. Xillybus IP core가 FIFO에서 낮은 "full" 신호를 감지하고 일부 DMA buffer에 데이터가 준비되어 있으면 DMA buffer에서 데이터를 가져와 FIFO에 씁니다. FIFO가 다시 가득 차거나 DMA buffers가 비어 있으면 IP core의 내부 상태 머신은 일시적으로 데이터 가져오기를 중지한 다음 DMA buffer 풀에서 중단된 위치에서 계속합니다.

데이터 흐름이 중단되는 동안 IP core는 다른 활동, 예를 들어 다른 stream을 대신하여 데이터 복사(즉, 다른 중간 FIFO 채우기)로 바쁠 수 있습니다. 결과적으로 FIFO가 "full" 신호를 로우로 변경하는 시간과 데이터 복사를 재개하는 시간 사이에 임의의 지연이 있을 수 있습니다. 이 지연은 다양하지만 전반적으로 IP core는 512단어의 FIFO가 충분히 깊음을 보장합니다.

하드웨어는 물론 부분적으로 채워진 DMA buffers를 인식하고 각각에 포함된 데이터의 양을 추적합니다.

A.4.4 #3 단계: 중급 FIFO application logic

FPGA의 user application logic은 user application logic와 Xillybus IP core를 연결하는 FIFO에서 데이터를 가져옵니다. underflow를 피하기 위해 FIFO의 “empty” 신호를 존중하는 것을 제외하고 언제 또는 얼마나 많은 데이터를 가져오는지에 대한 요구 사항은 없습니다.

A.4.5 예

다음과 같은 8비트 비동기식 stream의 간단한 경우를 살펴보겠습니다. stream이 데이터를 포함하지 않은 상태로 시작하고 host의 애플리케이션이 device file에 단일 바이트를 씁니다.

이벤트 순서는 다음과 같습니다.

- driver의 write() 함수 호출은 1바이트 쓰기 요청과 함께 호출됩니다.
- stream에는 데이터가 포함되어 있지 않으므로 DMA buffers에는 분명히 공간이 있습니다. 따라서 driver는 바이트를 첫 번째 DMA buffer에 복사하고 반환합니다.
- 10 ms 중에는 아무 일도 일어나지 않습니다.
- autoflush 메커니즘은 10 ms 이후에 트리거되어 driver가 DMA buffer에 1바이트가 포함된 정보와 함께 DMA buffer를 하드웨어로 넘겨줍니다.
- Xillybus IP core는 DMA buffer에서 바이트를 읽고 이를 중간 FIFO에 씁니다.
- application logic은 마음대로 FIFO에서 바이트를 읽을 수 있습니다.

A.4.6 실용적인 결론

- 응용 프로그램 수준 데이터가 항상 N 바이트 청크로 구성되더라도 DMA buffer 크기를 어떤 식으로든 조정할 이유가 없습니다. user application software는 각 청크의 끝에서 0바이트를 요청하는 write() 함수 호출과 함께 데이터의 flush를 요청하기만 하면 됩니다. 마이크로초 단위의 latency는 이러한 방식으로 달성됩니다.
- 연속적인 streams 데이터의 경우에도 latency는 작은 buffers로 write() 함수 호출을 수행한 다음 flush(0바이트의 write() 함수 호출)를 수행하여 추가 운영 체제의 오버헤드를 희생하여 줄일 수 있습니다. DMA buffers의 크기에 관계없이 latency는 데이터 속도와 flush 요청 간의 데이터 양에만 의존합니다.

- flush가 항상 주어진 데이터 청크 이후에 발생하므로 DMA buffer가 특정 수준 이상으로 채워지지 않는다는 것을 미리 알고 있다면 DMA buffer 크기를 줄이는 것이 합리적일 수 있습니다. 그러나 그렇게 하는 것의 유일한 장점은 host에서 RAM의 양을 절약할 수 있다는 것입니다.
- 10 ms가 허용 가능한 latency라면, autoflushing 메커니즘이 활동이 없는 10 ms 이후에 시작되기 때문에 최적화의 의미가 없습니다.