(機械で日本語に翻訳)

Xillybus host application programming guide for Linux

Xillybus Ltd.

www.xillybus.com

Version 3.0

この文書はコンピューターによって英語から自動的に翻訳されているため、言語が不明瞭になる可能性があります。このドキュメントは、元のドキュメントに比べて少し古くなっている可能性もあります。

可能であれば、英語のドキュメントを参照してください。

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 This document may also be slightly outdated in relation to the original.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

序章

Xillybus は、Linux host にシンプルでよく知られているインターフェイスを提供し、自然で期待どおりの動作をするように設計されています。 host driver は、named pipes のように動作する device files を生成します。それらは、他のファイルと同じように開かれ、読み書きされますが、プロセス間または TCP/IP streams 間の pipes のように動作します。 host で実行されているプログラムとの違いは、stream の反対側が別のプロセス (ネットワークまたは同じコンピューター上) ではなく、FPGA 内の FIFO であることです。 TCP/IP stream と同様に、Xillybus stream は、高速データ転送だけでなく、時々受信または送信される単一バイトでもうまく機能するように設計されています。

Xillybus とのインターフェイスはすべて device files を介しており、ファイルと同じようにアクセスできるため、通常は実用的なプログラミング言語を使用でき、特別なモジュール、拡張機能、またはその他の適応は必要ありません。選択した言語でファイルを開くことができる場合、そのファイルを使用して FPGA から Xillybus にアクセスできます。

1 つの driver binary が任意の Xillybus IP core 構成をサポートします。streams とその属性は、デバイスの初期化時に driver によって自動検出され、それに応じて device files が作成されます。これらの device files は `/dev/xillybus_something` (または `/dev/xillyusb_something` と XillyUSB) としてアクセスされます。

動作中、FPGA と host の間のハンドシェイク プロトコルは、継続的な data stream の錯覚を引き起こします。舞台裏では、driver の buffers が埋められ、処理されます。 TCP/IP streaming に使用されているものと同様の手法を使用して、buffers を効率的に利用しながら、小さなデータの応答性を維持します。

Xillybus I/O は Linux の device file I/O と同じように実行されるため、一般的なプログラミング手法を使用できるため、プログラミング ガイドは明らかに必要ありませ

ん。

それでも、FPGA との通信には、ファイル I/O には一般的ではないタスクが含まれることがよくあります。このガイドでは、一般的な FPGA 関連のプロジェクトを実装する方法と、最適なパフォーマンスを実現する方法を提案します。経験豊富なプログラマーは、同じように成功するさまざまな方法を選択する場合があります。

このガイドの大部分は、堅牢で効率的な I/O が UNIX システムに実装される方法の概要にすぎません。このような手法に精通している人は、このガイドのいくつかの部分が冗長であることに気付くかもしれません。 Xillybus は、新しい API を発明するためではなく、経験豊富なプログラマーが期待するように動作するように設計されました。

このガイドの例は、わかりやすくするために、また低レベルのシステム コールに密接に関連していることが知られている一連の関数を持っているため、単純な C で示されています。説明されている手法は、Perl や Python などの script 言語を含む他のいくつかの言語で実装できます。特に、host アクションと FPGA アクション間のパフォーマンスと同期の要件がそれほど厳密ではない場合です。

一部の I/O は、shell scripts とワンライナーでも実行できます。

2

同期 streams 対非同期 streams

2.1 概要

各 Xillybus stream には、同期または非同期のどちらで動作するかを決定するフラグがあります。このフラグの値は、FPGA の logic で固定されています。

stream が非同期とマークされている場合、それぞれの device file が開いている限り、user space software の関与なしに FPGA と host の kernel driver の間でデータを通信できます。

非同期 streams は、特にデータ フローが連続している場合に、パフォーマンスが向上します。同期 streams は扱いやすく、user space application の動作と FPGA で発生する動作との間に厳密な同期が必要な場合に適しています。

IP Core Factory で生成されたカスタム IP cores では、各 stream を同期にするか非同期にするかの選択は、“autoset internals” が有効になっているときにツールのユーザーが宣言した stream の使用目的に関する情報に基づいて自動的に行われます。autoset オプションがオフになっている場合、ユーザーはこの選択を明示的に行います。

いずれにせよ、IP Core Factory からダウンロードされたバンドルに含まれる “readme” ファイルは、各 stream の同期フラグまたは非同期フラグを (他の属性と共に) 指定します。

すべての demo bundles で、xillybus_read_* と xillybus_write_* に関連する streams は非同期です。 xillybus_mem_8 は seekable であるため、同期的です。 XillyUSB を使用する場合、それぞれの xillyusb_* ファイルにも同じことが当てはまります。

2.2 非同期 streams の動機

Linux や Microsoft Windows などのマルチタスク オペレーティング システムは、CPU タイム シェアリングに基づいています。プロセスは CPU のタイム スライスを取得し、特定の時点でどのプロセスが CPU を取得するかを決定するスケジューリング アルゴリズムを使用します。

プロセスに優先順位を設定することはできますが、プロセスが継続的に実行されるという保証や、マルチプロセッサ コンピューターであっても、preemption の期間が限られているという保証はありません。オペレーティング システムの基本的な前提は、どのプロセスも CPU starvation の任意の期間を受け入れることができるということです。リアルタイム指向のアプリケーション (サウンド アプリケーションやビデオ プレーヤーなど) には、この問題に対する明確な解決策がありません。代わりに、オペレーティング システムの典型的な事実上の動作に依存し、I/O バッファリングで preemption periods を補います。

非同期 streams は、アプリケーションが preempted であるか、他のタスクでビジーである間、データが継続的に流れることを可能にすることで、この問題に取り組みます。どちらの方向の streams に対するこれの正確な意味については、次に説明します。

2.3 FPGAからhostへのStreams

upstream 方向 (FPGA から host) では、stream が非同期の場合、FPGA 内の IP core は可能な限り host driver の buffers を埋めようとします。つまり、ファイルが開いている場合、データが利用可能であり、それらの buffers に空き容量があります。

一方、stream が同期の場合、host 上の user application software が file descriptor からそのデータを読み取る保留中の要求を持っている場合にのみ、IP core は user application logic から (通常は FIFO から) データをフェッチします。つまり、user application software が read() 関数呼び出しの途中にある場合です。

主に次の 2 つの理由から、高帯域幅アプリケーションでは同期 streams を避ける必要があります。

- アプリケーションが preempted またはその他の処理を行っている間はデータフローが中断されるため、物理チャネルは特定の期間使用されないままになります。ほとんどの場合、これにより帯域幅のパフォーマンスが大幅に低下します。

- これらのタイム ギャップ中に、FPGA の FIFO で overflow が発生する場合があります。たとえば、そのフィル レートが 100 MB/sec の場合、2 kByte を搭載した一般的な FPGA FIFO は、0.02 ms 前後で空から満杯になります。実際には、これは user space program の preemption が FPGA で FIFO の overflow を引き起こす可能性があることを意味します。

これらの欠点にもかかわらず、同期 streams は、FPGA でデータが収集された時間が重要な場合に役立ちます。特に、メモリのようなインターフェイスには同期インターフェイスが必要です。

application logic から FPGA 上の Xillybus IP core によって受信されたデータは、stream が同期か非同期かに関係なく、host の user space application によってすぐに読み取ることができます。

2.4 hostからFPGAへのStreams

downstream 方向 (host から FPGA) では、stream が非同期であることは、host application の write() 関数呼び出しがほとんどの場合すぐに戻ることを意味します。より正確には、driver の buffers にデータを完全に保存できる場合、device file に書き込む関数の呼び出しはすぐに戻ります。データは、host の application software の関与なしに、FPGA で user application logic によって要求されたレートで FPGA に送信されます。

XillyUSB と他の Xillybus IP cores の間には、FPGA への非同期 streams に代わってデータが FPGA に送信されるまでの時間に関して、わずかな違いがあります。

PCIe または AXI に基づく IP cores の場合、次のいずれかが発生した場合にのみ、データが FPGA に送信されます。

- 現在の DMA buffer は満杯です (各 stream に対して複数の buffers があります)。
- flush は、application software によって device file で明示的に要求されます (段落 3.4 を参照)。
- file descriptor は閉鎖中です。
- stream に特定の時間 (通常は 10 ms) 何も書き込まれなかった場合、タイマーが期限切れになり、自動 flush が強制されます。

XillyUSB stream では、データはほぼ即時に送信されます。より正確には、driver は固定サイズ (通常は 64 kB) の USB transfers をキューに入れようとしますが、送信

するデータがある場合はより小さい転送がキューに入れられ、関連する stream に対してキューに入れられる他の転送はありません。したがって、stream ごとに、固定サイズ未満の転送が複数キューに入れられることはありませんが、送信するデータがある限り、常に少なくとも 1 つの転送が進行中です。これにより、USB 転送の効率的な使用と、短いデータ セグメントへの迅速な応答が実現します。

全体として、すべての IP cores (XillyUSB およびその他の Xillybus IP cores) 上の非同期 streams はほぼ同じように動作し、XillyUSB はデータの短いセグメントに対してより速い応答時間を持ちます (10 ms 遅延なし)。

一方、stream が同期の場合、device file に書き込む低レベル関数の呼び出しは、すべてのデータが FPGA 内の user application の logic に到達するまで返されません。一般的なアプリケーションでは、これは write() への関数呼び出しが返されたときに、FPGA 内の IP core に接続された FIFO にデータが到着したことを示します。

重要:

fwrite() などの高レベルの *I/O* 関数には、*library functions* によって作成された *buffer layer* が含まれます。したがって、*fwrite()* および同様の関数は、同期 *streams* の場合でも、データが *FPGA* に到着する前に戻る可能性があります。

主に次の 2 つの理由から、高帯域幅アプリケーションでは同期 streams を避ける必要があります。

- アプリケーションが preempted または他の処理を行っている間はデータ フローが中断されるため、物理チャネルは特定の期間使用されないままになります。ほとんどの場合、これにより帯域幅のパフォーマンスが大幅に低下します。
- これらのタイム ギャップ中に、FPGA の FIFO で underflow が発生する場合があります。たとえば、ドレイン レートが 100 MB/sec の場合、2 kByte を搭載した一般的な FPGA FIFO は、0.02 ms 前後でフルからエンプティになります。実際には、これは user space program の preemption が FPGA で FIFO の underflow を引き起こす可能性があることを意味します。

これらの欠点にもかかわらず、同期 streams は、データが FPGA に到着したことをアプリケーションが認識することが重要な場合に役立ちます。これは、stream を使用して、他の操作 (ハードウェアの構成など) を実行する前に実行する必要があるコマンドを送信する場合に当てはまります。

2.5 不確実性 vs. latency

データ間の同期のために、非同期 streams で低い latency を要求するのはよくある間違いです。たとえば、アプリケーションがモデムの場合、通常、受信サンプルと送信サンプルを同期する必要があります。

これは、同期の不確実性がlatencyの合計よりも必然的に小さいという考えに基づいて、designの誤解につながることがよくあります。不確実性を低く抑えるために、latency、したがって buffers は可能な限り小さく作られているため、システム全体で real-time の要件が厳しくなっています。

Xillybus では、段落 6.2 で説明されているように、(単一のサンプルのレベルで) 同期を簡単に完全に行うことができます。したがって、latency の制限は、到着するデータに迅速に応答する必要がある場合に、その必要性から派生したものです。

たとえば、モデムの場合、最大 latency は、アプリケーションのデータ ソースが送信されたデータに応答する速度に影響を与えます。カメラ アプリケーションでは、host は、変化する照明条件を補正するために shutter speed を調整するようにカメラをプログラムする場合があります。より大きな latency で到着するデータは、この control loop を遅くします。

これらは実際に考慮する必要がある考慮事項ですが、それでも、latency に不確実性が混在しているという誤解から導き出されたものよりも、通常は大幅に厳格ではありません。

3

I/O プログラミングの実践

3.1 概要

Xillybus は、ファイルにアクセスできる任意のプログラミング言語で適切に動作し、ファイルにアクセスするための任意の API が適しています。

このガイドでは、open()、read()、write()、close() などの機能に基づく低レベルの API セットに重点を置いています。低レベルの API の機能には buffers の余分なレイヤーがないため、このセットは他のよく知られたセット (fopen()、fwrite()、fprintf() など) よりも選ばれます。これらの buffers はパフォーマンスにプラスの効果をもたらす可能性がありますが、実際の I/O 操作を制御することはできません。

これは、データが常に送信され、ソフトウェア操作と I/O とハードウェアとの間に直接的な関係がないと予想される場合、それほど重要ではありません。

余分な buffer レイヤーも混乱を引き起こし、ソフトウェアのバグがないのにソフトウェアのバグがあるように見えます。たとえば、fwrite() への関数呼び出しは、ファイルが閉じられるまで I/O 操作を実行せずに、RAM buffer にデータを格納するだけです。これを認識していない開発者は、実際にはデータが buffer で待機しているときに、FPGA 側で何も起こらなかったために fwrite() が失敗したと誤解する可能性があります。

このセクションでは、低レベルの C run-time library 関数を使用した、推奨される UNIX プログラミング プラクティスについて説明します。これらのプラクティスのいずれについても Xillybus に固有のものは何もないため、この詳細は完全を期すためにここに記載されています。

コード スニペットは、Getting started with Xillybus on a Linux hostで説明されているデモ アプリケーションから取得されます。これらの例の device file 名は、PCIe / AXI の Xillybus IP core の名前です。 XillyUSB の場合、プレフィックスは xillybus_*

ではなく xillyusb_00_* です。

3.2 データ読み取りのガイドライン

変数が次のように宣言されていると仮定します。

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

device file は低レベルの open で開かれます (file descriptor は integer 形式です)。

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

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

device file が別のプロセスによって読み取り用に既に開かれている場合は、“Device or resource busy” (errno = EBUSY) エラーが発行されます (要求に応じて非排他的なファイルを開くことができます)。 “No such device” (errno = ENODEV) が発生した場合は、書き込み専用の stream を開こうとしている可能性があります。

ファイルが正常に開かれ、`buf` がメモリ内の割り当てられた buffer を指している場合、データは次のように読み取られます。

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

`numbytes` は、読み取る最大バイト数です。

戻り値 `rc` には、実際に読み取られたバイト数 (関数呼び出しが異常終了した場合は負の値) が含まれます。

`numbytes` で要求された量のデータが利用可能な場合、read() は常にすぐに戻ることに注意してください。それ以外の場合、利用可能なデータがあれば約 10 ms 後に戻ります。利用可能なデータがまったくない場合、read() はデータを返すことができるまでスリープします。

driver は、IP core が FPGA の application logic からそのデータを受信したという意味で、データの可用性をチェックします。 DMA buffers のメカニズムは、関数 read() の呼び出し元に対して透過的であり、付録の A.3.5 セクションで説明されているように、DMA buffer がいっぱいではないため、read() 関数呼び出しへのデータの配信を遅らせることはありません。

重要:

read() が正常に返された場合でも、要求されたすべてのバイトがファイルから読み取られたという保証はありません。完了したデータ量が不十分な場合、*read()* への別の関数呼び出しを行うのは呼び出し元の責任です。

read() への関数呼び出しの後に、以下に示すようにその戻り値をチェックする必要があります (“continue” および “break” ステートメントは、while ループ コンテキストを想定しています)。

```
  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
}
```

最初の if ステートメントは、signal が原因で read() が時期尚早に返されたかどうかをチェックします。これは、プロセスがオペレーティング システムから signal を受信した結果です。

これは実際にはエラーではありませんが、driver が制御をすぐにアプリケーションに戻さなければならない状況です。 EINTR エラー番号の使用は、データが読み取られなかったことを関数の呼び出し元に伝える方法にすぎません。プログラムは “continue” ステートメントで応答し、その結果、同じパラメーターを使用して関数 read() を再度呼び出そうとします。

signal が到着したときに buffer に何らかのデータがある場合、driver は `rc` で既に読み取られたバイト数を返します。アプリケーションは、signal が到着したことを認識せず、UNIX プログラミング規則に従って、気にする理由はありません。signal がアクションを必要とする場合 (たとえば、キーボード上の CTRL-C から生じる SIGINT)、このアクションの責任は、オペレーティング システム、または登録済みの signal handler。

一部の signals は実行フローに影響を与えるべきではないことに注意してください。そのため、上記のように signals が検出されない場合、プログラムは明らかな理由もなく突然エラーを報告することがあります。

EINTR シナリオの処理も、プロセスを停止して (CTRL-Z と同様に) 適切に再開できるようにするために必要です。

2 番目の if ステートメントは、ユーザーが読み取り可能なエラー メッセージを報告した後に実際のエラーが発生した場合、ループを終了します。

3 番目の if ステートメントは、end of file に到達したかどうかを検出します。これは、戻り値ゼロによって示されます。 Xillybus device file から読み取る場合、これが発生する唯一の理由は、application logic が stream の _eof ピン (FPGA 上の IP core のインターフェースの一部) を上げたことです。

3.3 データ書き込みのガイドライン

変数が次のように宣言されていると仮定します。

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

device file は低レベルの open で開かれます (file descriptor は integer 形式です)。

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

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

device file が別のプロセスによって書き込み用に既に開かれている場合、“Device or resource busy” (errno = EBUSY) エラーが発行されます (要求に応じて非排他的なファイルを開くことができます)。 “No such device” (errno = ENODEV) が発生した場合は、読み取り専用の stream を開こうとしている可能性があります。

ファイルが正常に開かれ、`buf` がメモリ内の割り当てられた buffer を指している場合、データは次のように書き込まれます。

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

`numbytes` は、書き込まれる最大バイト数です。

戻り値 `rc` には、実際に書き込まれたバイト数 (関数呼び出しが異常終了した場合は負の値) が含まれます。

重要:

write() が正常に返された場合でも、要求されたすべてのバイトがファイルに書き込まれたという保証はありません。完了したデータ量が不十分な場合、*write()* に対して別の関数呼び出しを行うのは呼び出し元の責任です。

write() への関数呼び出しの後に、以下に示すようにその戻り値をチェックする必要があります (“continue” および “break” ステートメントは、while ループ コンテキストを想定しています)。

```
  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
}
```

最初の if ステートメントは、signal が原因で write() が時期尚早に返されたかどうかをチェックします。これは、プロセスがオペレーティング システムから signal を受信した結果です。

これは実際にはエラーではありませんが、driver が制御をすぐにアプリケーションに戻さなければならない状況です。 EINTR エラー番号の使用は、データが書き込まれていないことを関数の呼び出し元に伝える方法にすぎません。プログラムは “continue” ステートメントで応答し、その結果、同じパラメーターを使用して関数 write() を再度呼び出そうとします。

signal が到着する前に何らかのデータが書き込まれた場合、driver は `rc` に既に書き込まれたバイト数を返します。アプリケーションは、signal が到着したことを認識せず、UNIX プログラミング規則に従って、気にする理由はありません。signal がアクションを必要とする場合 (たとえば、キーボード上の CTRL-C から生じる SIGINT)、このアクションの責任は、オペレーティング システム、または登録済みの signal handler。

一部の signals は実行フローに影響を与えるべきではないことに注意してください。そのため、上記のように signals が検出されない場合、プログラムは明らかな理由もなく突然エラーを報告することがあります。

EINTR シナリオの処理も、プロセスを停止して (CTRL-Z と同様に) 適切に再開できるようにするために必要です。

2 番目の if ステートメントは、ユーザーが書き込み可能なエラー メッセージを報告した後に実際のエラーが発生した場合、ループを終了します。

3 番目の if ステートメントは、end of file に到達したかどうかを検出します。これは、戻り値 0 によって示されます。 Xillybus device file に書き込む場合、これは決して起こらないはずです。

3.4 非同期 downstreams で flush を実行する

2.4 の段落で述べたように、PCIe / AXI IP core 上の非同期 stream に書き込まれたデータは、DMA buffer がいっぱいでない限り (複数の DMA buffers がある)、必ずしもすぐに FPGA に送信されるとは限りません。この動作により、割り当てられた buffer スペースが確実に使用されるようになるため、パフォーマンスが向上します。これにより、PCIe / AXI bus で送信されるパケットの効率も向上します。

すでに述べたように、stream が非同期の場合でも、XillyUSB IP cores は事実上すぐにデータを送信します。これは、USB インターフェイスを使用した効率的な配置があるためです。したがって、flush を実行することは、送信が完了するのを待つ必要がある場合にのみ、XillyUSB IP cores で意味があります。

Streams から FPGA は、file descriptor を閉じるときに自動的に flush を受けますが、これは信頼できないベスト エフォート メカニズムです。 close() への関数呼び出しは、write() 関数呼び出しが同期 streams で遅延されるのと同様の方法で、すべてのデータが FPGA に到着するまで遅延されます。大きな違いは、close() は flush が完了するまで最大 1 秒待機することです。それまでに flush が完成しない場合、close() はとにかく戻り、system log で警告メッセージを発行します。ただし、まれに、file descriptor を閉じるときに、残りのデータの最後の数ワードが何の警告もなしに失われる場合があることに注意してください。

長さがゼロの buffer で関数 write() を呼び出すことにより、非同期 stream の flush を明示的に要求することもできます。

```
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
}
```

次の点に注意してください：

- UNIX の manual page は、count がゼロの場合に write() 関数呼び出しが何をすべきかを定義せず、選択は各 device driver に任せます。 flushing のこのメソッドは、Xillybus に固有です。
- close() とは異なり、上記の write() は、FPGA でデータがいつ消費されたかに関係なく、すぐに戻ります。
- このため、この種の write() は XillyUSB では意味がありません。何もする必要はなく、実際には何もしません。とにかく、データは事実上すぐに送信され、write() 関数呼び出しはどのような場合でも待機しません。
- buffer からデータが読み取られないため、write() 関数呼び出しの buffer 引数は、上記で示したように、NULL を含む任意の値を取ることができます。
- 長さゼロの buffer でより高いレベルの API を使用しても、まったく効果がない場合があります。たとえば、関数 fwrite() を呼び出して 0 バイトを書き込むと、何もせずに単に戻る場合があります。これは、この関数が通常行うことは、C run-time library によって作成された buffer にデータを追加することだからです。
- fflush() は関係ありません。上位レベルの buffer の flush を実行しますが、下位レベルの driver に flush コマンドを送信しません。
- streams で flush を別の方向 (FPGA から host へ) で実行する必要はなく、実行する方法もありません。これは、このような streams の flush が、host がデータを読み取ろうとしてプロセスをスリープ状態にしようとしているときに自動的に実行されるためです (つまり、 block)。

3.5 select()およびnonblocking I/O

推奨されていませんが、Linux 用の Xillybus driver は、nonblocking calls および select() 機能をサポートしています。 Windows 用の driver は同様のものをサポートしていないため、この機能を使用すると、必要に応じてアプリケーションの移植が難しくなります。複数のソースを処理するための推奨される方法は、段落 4.4 で説明されている fifo.c サンプル プログラムで示されているように、複数の threads (およびできれば RAM FIFOs) を使用することです。

select()、pselect()、および poll() への関数呼び出しは、UNIX file descriptor と同様に、読み取りと書き込みの両方で使用できます。

nonblocking calls および select() 機能は、IP Core Factory で “Windows only” として設定された Xillybus IP cores では有効になりません。

完全を期すために、nonblocking 読み取りを使用して、段落 3.2 のデータを読み取るためのコード アウトラインを再検討します。このコードは、UNIX のファイルから読み取った nonblocking の従来の方法を示しているだけです。

ファイルは O_NONBLOCK フラグで開かれます。

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

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

ファイルの読み取り方法、引数、または戻り値の意味に違いはありません。

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

`rc` が負で、EAGAIN がエラー コードとして与えられた場合、これは読み取るものが何もないことを意味します。より正確には、driver の buffers にはデータがなく、FPGA の FIFO は空です。

```
  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
}
```

上記のコードは、関数呼び出しが EAGAIN で戻るときに何か意味のあることが行われない限り、意味をなさないことに注意してください。そうしないと、読み取るデータがないときにスリープする代わりに、while ループでスピンすることによって CPU time を浪費するだけです。

nonblocking の書き込みについては、段落 3.3 の例でそれぞれの変更を行います。

3.6 driver の buffers のデータ量の監視

このトピックは、Xillybus FPGA designer's guideの “Monitoring the amount of buffered data” という名前のセクションで説明されています。

3.7 XillyUSB: 物理的な data link の品質を監視する必要性

PCIe とは異なり、USB 3.0 で使用される物理 data link は bit errors を生成することが確認されています。これは一般的ではなく、関係するコンポーネントの 1 つ (host の USB ポートまたはケーブル) に問題があることを示しています。

USB プロトコルは、bit errors が発生した場合にそれを克服するためのさまざまなメカニズムを提供しますが、これらのエラーのランダムな性質により、link protocol

はめったに到達しない状態になります。その結果、host の USB controller のバグが明らかになる可能性があります。このようなバグは、存在する限り、通常は隠され、さまざまな奇妙な動作を引き起こします。

したがって、物理 data link で bit errors が頻繁に発生する場合、USB 接続がスタックしたり、自然に切断されたり、まれにアプリケーション データにエラーが発生したりする重大なリスクがあります。

XillyUSB は、専用の device file、/dev/xillyusb_NN_diagnostics によって、物理的な data link の正常性を監視する手段を提供します。 showdiagnostics ユーティリティ (この web page で説明) は、この件に関して収集された情報を公開します。

XillyUSB に基づくアプリケーションは、showdiagnostics ユーティリティによって表示される最初の 5 つのカウンター (不良パケット、検出されたエラー、および Recovery 要求に関連するもの) を継続的に監視し、それらが増加しないようにすることを強くお勧めします。その場合、特に繰り返し増加する場合、アプリケーション ソフトウェアは、おそらく次のいずれかの是正措置を提案する必要があります。

- USB プラグを取り外して、別のポートに再接続します。一部のマザーボードには異なるブランドの USB host コントローラに接続された異なるポートがあるため、これが役立つ場合があります (通常、USB 3.x プロトコルの新しいバージョンをサポートするため)。
- 同じポートの USB プラグを取り外して再接続します。これは、analog signal equalizer (物理的な信号経路に起因する減衰と反射をキャンセルする) が最終的に最適でない状態になった場合に役立つ可能性があります。
- 別の USB ケーブルを使用してみます。

bit errors が存在する場合でも、アプリケーションが問題なく動作し続ける可能性は十分にあります。したがって、是正措置の提案は、ユーザーがおそらく目に見える問題を経験していないことを考慮して行うのが最善です。

showdiagnotics.pl ユーティリティは、参照コードとして使用できる Perl script です。または、C のソース コードとして提供されている Windows の診断ユーティリティを参照することもできます。

これらの問題はいずれも XillyUSB に固有のものではないことに注意してください。むしろ、これらの問題はどの USB 3.0 デバイスにも影響を与える可能性がありますが、XillyUSB はそれらを検出する手段を提供しています。また、PCIe リンクで同様の問題が発生することは知られていないことを繰り返し述べておく必要があります。これは、物理接続と信号ルーティングが適切に制御されているためと考えられます。

4

高速で連続 I/O

4.1 基礎

host と FPGA の間で高速で連続的なデータ フローを実現するためにほぼ不可欠な 4 つのプラクティスがあります。

- 非同期 streams の使用
- driver の buffers が、user space application の I/O 操作間の時間ギャップを補うのに十分な大きさであることを確認します。
- user space application に、利用可能なデータがあればすぐに device file からデータを読み取らせるか、buffers で利用可能なスペースがあればすぐに device file にデータを書き込んでもらいます。
- FPGA がデータの挿入または排出を続けている間は、device files を閉じて再度開くことはありません。

XillyUSB は、この web page で説明されているように、データの継続的なフローを維持するという追加の課題を提示します。

任意の時点で driver の buffers に保持されているデータ量の監視については、Xillybus FPGA designer's guideの “Monitoring the amount of buffered data” というセクションで説明されています。

上記のリストの最初の項目である、非同期 streams の使用については、セクション 2 で説明されています。 2 番目と 3 番目については、このセクションの残りの部分で説明します。

4 番目の項目を理解するために、非同期 streams の利点は、user space application の介入なしに FPGA と host の間でデータが実行されることを思い出してくださ

い。ファイルが閉じられると、このフローは停止します。

特に host から FPGA への stream の場合、ファイルを閉じると buffers 内のすべてのデータの flush が強制され、ファイルはそれが終了した後 (または 1 秒後) にのみ閉じられます。その結果、ファイルが閉じられた瞬間からファイルが再び開かれるまで (そしてデータが file descriptor に書き込まれるまで) に、データ フローのない時間のギャップが生じます。

FPGA からの streams に関しては、ファイルを閉じると、FPGA の application logic から host の user space application (つまり、FPGA の FIFO および driver の buffers) に移動する pipe のデータが失われます。この損失を回避する唯一の方法は、ファイルを閉じる前に、この pipe からすべてのデータを排出することです。ここでも、ファイルを閉じてから再度開くまでの間に、データが流れない時間のギャップがあります。

よくある間違いは、EOF 機能を使用してデータ チャンク (完全な video frames など) をマークすることです。これにより、host が既知の境界で device file を強制的に閉じて再度開くようになります。ただし、これにより、FPGA の FIFO での overflow のリスクが大幅に増加します。

オペレーティング システムが任意の時点 (preemption) で CPU を user space application から削除する可能性があることを覚えておくことが重要です。そのため、プログラム内の後続の関数呼び出しの間に数ミリ秒、場合によっては数十ミリ秒の時間差が発生する可能性があります。

4.2 大型driverのbuffers

FPGA と host の間でデータを高速で転送する際の最大の課題の 1 つは、継続的なフローを維持することです。 data acquisition と再生を含むアプリケーションでは、overflow またはデータの不足により、システムが機能しなくなります。これを回避するために、driver は host に大きな RAM buffers を割り当てて、独自に使用します。これらの buffers は、アプリケーションがデータ転送を処理できない時間のギャップを補います。

Xillybus では巨大な driver の buffers を割り当てることができますが、このメモリはオペレーティング システムの kernel RAM のプールから割り当てる必要があります。一部のシステム (特に 32 ビット システム) では、使用可能な RAM の合計がかなり大きい場合でも、このようなメモリのアドレス空間は Linux オペレーティング システムによって 1 GB に制限されます。 RAM が 1 GB 未満のシステム (特に embedded Linux) では、すべてのメモリが driver の buffers に使用される場合があります。

このページで説明されているように、拡張された host driver を使用すると、64 ビット システムでさらに大きな buffers を割り当てることができます。

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

XillyUSB の場合を除き、driver の buffers は、Xillybus driver がロードされたとき (通常は boot プロセスの初期) に割り当てられ、driver が kernel からアンロードされたとき (通常は system shutdown の間) にのみ解放されます。 buffers が巨大な場合、これは通常、kernel の RAM プールの大部分が driver の buffers によって占有されていることを意味します。これらの buffers を使用するアプリケーションは、それが実行されているマシンの主な目的である可能性が高いため、これはかなり妥当な設定です。

巨大な buffers の潜在的な問題は、それらが物理 RAM の連続したセグメントを占有することです。これは、virtual address space で連続している userspace プログラムで割り当てられた buffer とは対照的ですが、物理メモリ全体に分散するか、物理 RAM をまったく占有しないことさえあります。

オペレーティング システムが実行されると、使用可能なメモリのプールが断片化されます。これが、Xillybus driver が buffers をできるだけ早く割り当て、アクティブに使用されていない場合でもそれらを保持する理由です。 driver をアンロードして後の段階で再ロードしようとすると、同じ理由で失敗する場合があります。

XillyUSB はメモリ割り当てに対して異なるアプローチを採用しており、物理メモリの断片化に対してより耐性があります。これが、device file が開かれたときに driver が buffers に RAM を割り当て、ファイルが閉じられたときにそれを解放する理由の 1 つです。

ただし、kernel RAM が不足しないように注意する必要があります。 IP Core Factory の自動メモリ割り当て (“autoset internals”) アルゴリズムは、最近の PC には 1 GB よりも多くの RAM がインストールされているという仮定に基づいて、関連するメモリ プールの 50% (PC コンピュータの 512 MB など) を超えて消費しないように設計されています。 buffer のサイズを手動で設定することで、75% まで上げてもおそらく安全です。

buffers を過剰に割り当てると、システムが不安定になる可能性があります。特に、オペレーティング システムは、kernel pool から RAM の割り当てに失敗するたびに、明らかにランダムにプロセスを強制終了する可能性があります。

4.3 user spaceのRAM buffers

32 ビット マシンで 512 MB より大きい buffers を必要とするアプリケーションの場合、user space RAM でバッファリングの一部を行うことをお勧めします。 64 ビッ

ト マシンでは、必要な buffer サイズが非常に大きく、2 の累乗 (2^N) でない場合を除いて、このオプションはほとんど関係ありません。たとえば、stream に 62 GB の buffer を供給することは、Xillybus DMA buffers のおかげでは不可能ですが、user space RAM では実現できます。

user space application に巨大な buffer を割り当てることで、I/O の連続性の問題を解決できるというのは直感に反するように思えるかもしれません。実際、オペレーティング システムが CPU 時間のアプリケーションを枯渇させている場合、この解決策は役に立ちません。しかし、オペレーティング システムの scheduler が適切に設計され、優先順位が適切に設定されている場合、user space application は、負荷の高いコンピューター上でも CPU スライスを十分に頻繁に取得します。

buffer の最初のフィルに注意を払うことが重要です。最新のオペレーティング システムは、user space application がメモリを要求したときに、物理的な RAM を割り当てません。代わりに、メモリ割り当てを反映するように memory page tables をセットアップするだけです。実際の物理メモリは、アプリケーションが使用しようとしたときにのみ割り当てられます。これはリソースを節約するための優れた方法ですが、data acquisition アプリケーションに壊滅的な影響を与える可能性があります。たとえば、データ ソースからデータが殺到し始めるとどうなるかを考えてみてください。アプリケーションは割り当てられたばかりの buffer にデータを書き込みますが、新しい memory page がアクセスされるたびに、オペレーティング システムは新しい physical memory page を取得する必要があります。たまたま空いている物理 RAM がある場合、または物理メモリを解放する簡単な方法がある場合 (たとえば、既にディスクと同期している disk buffers)、このメモリのジャグリングは見過ごされる可能性があります。しかし、物理 RAM の直接のソースがない場合、ディスク操作 (RAM swapping からディスクまたは flushing disk buffers) を実行する必要があり、アプリケーションが長時間停止する可能性があります。

非常に悪いニュースは、データの初期ロードを実行できるかどうかがシステム全体の状態に依存することです。したがって、通常は動作するプログラムが突然失敗することがあります。これは、他のプログラムが同じコンピューターでデータ集約型の何かを実行したためです。

自然な解決策はメモリのロックです。mlock() は、(仮想) メモリの特定のチャンクを物理 RAM に保持する必要があることをオペレーティング システムに伝えます。これにより、物理メモリの割り当てが即座に強制されるため、関数呼び出しを完了するためにディスク操作が必要な場合は、戻るまでに時間がかかる場合があります。

オペレーティング システムは、全体的なパフォーマンスに影響を与えるため、RAM の大きなチャンクをロックすることに消極的です。ほとんどの場合、shell の制限を引き上げるか、構成ファイルをセットアップする必要があります。

4.4 fifo.c デモ アプリケーションの概要

Linux および Windows 用にダウンロードできるデモ アプリケーションの中には、“fifo.c” と呼ばれるものがあります。これは、2 つの threads を使用して RAM FIFO を実装する方法の例であり、32 ビットおよび 64 ビットのプラットフォームでテストされています。

デモ アプリケーションの詳細については、Getting started with Xillybus on a Linux hostを参照してください。

ドキュメントの他の部分とは異なり、このセクションの “FIFO” という単語は、FPGA の FIFO ではなく、host の RAM buffer を指すことに注意してください。

このプログラムの目的は、巨大な RAM buffer を維持するために RAM FIFO が必要な高速 streams をテストすることです。つまり、たとえば 16 GB よりも小さい buffer が必要な場合、このプログラムは必要ない可能性があります。

また、カスタム アプリケーションでの変更および採用の基礎として使用することもできます。 mutexes を使用しないように設計されているため、別の thread が lock を保持しているという理由だけで thread がスリープ状態になることはありません。もちろん、スリープ (blocking) は、FIFO の状態で必要な場合 (たとえば、空の FIFO から読み取りが要求された場合) に発生します。

mutexes を使用しないこの実装では、reentrant ではないため、API 関数を慎重に使用する必要があります。ただし、読み取り用に thread が 1 つ、書き込み用に thread が 1 つある場合、これは問題ありません。

device file から 128 MB の buffer を含むディスク ファイルに data acquisition 用に実行するには、次のように入力します。

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

2 番目の引数としてファイル名が指定されていない場合、プログラムは standard input から読み取ります。

’limit -l’ を shell prompt で使用し、root 特権を使用して、ロックされたメモリの制限を解除する必要がある可能性があります (おそらく、“su - your-username” を root として使用して、特権を通常のユーザーに戻して、更新された制限を保持します)。制限の一定の変更については、Linux ディストリビューションのドキュメントを参照してください。

プログラムは 3 つの threads を作成します。

- read_thread() は standard input (またはコマンド ラインで指定されたファイ

ル) から読み取り、データを FIFO に書き込みます。

- write_thread() は FIFO から読み取り、standard output に書き込みます
- status_thread() は standard error にステータス行を繰り返し出力します

3 番目の thread には機能上の重要性がないため、削除できます。メインの thread で実行される読み取り/書き込み機能の 1 つを持つことも可能です。たとえば、data acquisition アプリケーションでは、file descriptor から FIFO にデータを移動するために read_thread() のみを起動し、メイン アプリケーションの thread で FIFO からのデータを消費するのが自然な場合があります。

4.5 fifo.c 改造メモ

プログラムを変更する場合は、次の点に注意してください。

- fifo_* 機能は reentrant ではありません。各 thread が他の thread が使用しない関数のセットを使用する場合 (これは自然な使用法です)、それらを使用しても安全です。
- 関数 fifo_init() は戻るのに時間がかかる可能性があるため、非同期 Xillybus device file を開く前に呼び出す必要があります。
- アプリケーションで読み取る thread と書き込みを行う thread は、常に I/O 要求で許可されている最大バイト数を試行します。これは、I/O ソースが /dev/zero で宛先が /dev/null の場合など、場合によっては問題になる可能性があります。どちらも 1 回の試行で要求全体を完了するため、FIFO は完全に空から完全にいっぱいになり、何度も繰り返されます。このような場合、I/O 関数の呼び出しで要求されるバイト数を制限する方が賢明です。

4.6 RAM FIFO 関数

fifo.c の例を変更することを除いて、ソース コードから関数のグループを採用することができます。

FIFO API 関数のセクションは、fifo.c ファイルで明確に区別されます。これらの関数は、例に従い、以下の関数の説明に従って、カスタム アプリケーションで使用できます。

重要:

*fifo_** 関数は *multi-threaded* 環境での使用を意図していますが、これらの関数は***reentrant* ではありません**。これは、*1* つの *thread* が *FIFO* からの読み取りに関連する関数を呼び出す必要があり、別の *thread* が書き込みを行う必要があることを意味します。したがって、各 *thread* はそれぞれの関数セットを呼び出します。

イニシャライザー、デストロイヤー、および thread join ヘルパーを除いて、API には読み取りと書き込み用の 4 つの関数 (各方向に 2 つ) があります。これらの関数はどちらも、実際には FIFO のデータにアクセスしません。 FIFO の状態を維持し、読み取り、書き込み、メモリ コピーなどを実行するために必要な情報を提供するだけです。

意図した実行手順は次のとおりです。 FIFO から読み取る thread は、読み取れるバイト数に関する情報を返す関数 fifo_request_drain() と、データを読み取ることができる pointer を呼び出します。 FIFO が空の場合、thread はデータが到着するまでスリープします。

次に、ユーザー アプリケーションは、指定されたデータを使用して、必要なものを何でも使用します。一部またはすべてのデータの消費 (ファイルへの書き込み、データのコピー、何らかのアルゴリズムの実行など) が終了した後、関数 fifo_drained() を呼び出して、実際に消費されたバイト数を FIFO API に通知します。 API は、FIFO 内のメモリの関連部分を解放します。 FIFO がいっぱいだったために書き込みを行った thread がスリープ状態だった場合は、それが起こされます。

読み取る thread は特定のバイト数を要求しないことに注意してください。むしろ、fifo_request_drain() はアプリケーションに消費できるバイト数を伝え、アプリケーションは fifo_drained() で消費することを選択したバイト数を報告します。

反対方向については、同様のアプローチが取られます。書き込みを行う thread は関数 fifo_request_write() を呼び出します。この関数は、FIFO に書き込むことができるバイト数を返すか、FIFO がいっぱいの場合はスリープします。ユーザー アプリケーションは、fifo_request_write() から取得したアドレスに必要なバイト数 (ただし、fifo_request_write() で許可されたバイト数を超えることはありません) を書き込み、fifo_wrote() に行ったことを報告します。

これらの各機能について詳しく説明します。

4.6.1 fifo_init()

fifo_init(struct xillyfifo *fifo, unsigned int size) – この関数は、FIFO の情報構造を初期化し、FIFO にもメモリを割り当てます。また、FIFO の virtual memory を物理的な RAM にロックしようとして、すぐに高速書き込みできるようにし、swapped to disk になるのを防ぎます。

fifo_init() は、`size` バイトの buffer にメモリを割り当てます。 `size` は任意の integer にすることができます (つまり、2 の累乗である 2^N である必要はありません) が、システムが `int` と見なすものの倍数が推奨されます。

この関数が戻るのに数秒かかる場合があることに注意してください: 物理的な RAM の大部分に対する要求により、オペレーティング システムは強制的に他のプロセスの RAM pages をディスクにスワップするか、disk cache flushing を強制する場合があります。どちらの場合も、fifo_init() は大量のデータがディスクに書き込まれるのを待ってから復帰する必要があります。

この関数は、成功するとゼロを返し、それ以外の場合はゼロ以外を返します。

4.6.2 fifo_destroy()

fifo_destroy(struct xillyfifo *fifo) – ロック解除後に FIFO のメモリを解放し、thread synchronization リソースを解放します。 Linux の現在の実装では thread synchronization リソースが自動的に解放されますが、API ではこれが保証されないため、この関数はメイン プログラムの終了時に呼び出す**必要**があります。

この関数は void 型です (したがって、何も返されません)。

4.6.3 fifo_request_drain()

fifo_request_drain(struct xillyfifo *fifo, struct xillyinfo *info) – FIFO から info->addr としてデータを読み取る pointer を提供し、その pointer から開始して info->bytes で読み取ることができるバイト数を通知します。

`info` 構造体は、fifo_request_write() への関数呼び出しに使用されるものと同じであってはなりません。各 thread は、この構造体のために独自のローカル変数を維持する必要があります。

重要:

返されたバイト数は、*FIFO* で読み取るために残っているデータの量を示すものではありません。*FIFO* のメモリ *buffer* の最後まで残っているバイト数を反映している場合もあります。したがって、*pointer* が *buffer* の最後に近づくと、大幅に低い数になる可能性があります。

この関数は、fifo->position を設定して、FIFO の現在の読み取り位置を 0 size-1 の値で示します。ここで、`size` は fifo_init() に与えられた値です。ゼロ以外の fifo->slept は、呼び出し時に FIFO が空だったことを示します。

この関数は、読み取り可能なバイト数を返します (info->taken と同じ)。ただし、関数 fifo_done() が呼び出され、FIFO が空の場合、fifo_request_drain() はゼロを返します。

4.6.4 fifo_drained()

fifo_drained(struct xillyfifo *fifo, unsigned int req_bytes) – この関数は、req_bytes バイトの消費を反映するように FIFO の状態を変更します。 FIFO がいっぱいだったために fifo_request_write() がスリープしていた場合、それは起こされます。

重要:

req_bytes には **健全性チェックはありません**。 *req_bytes* が、*fifo_request_drain()* への最後の関数呼び出しによって返された *info->bytes* よりも大きくないことを確認するのは、ユーザー アプリケーションの責任です。

この関数は void 型です (したがって、何も返されません)。

4.6.5 fifo_request_write()

fifo_request_write(struct xillyfifo *fifo, struct xillyinfo *info) – info->addr として FIFO にデータを書き込む pointer を提供し、その pointer から info->bytes に書き込むことができるバイト数を通知します。

`info` 構造体は、fifo_request_drain() への関数呼び出しに使用されるものと同じであってはなりません。各 thread は、この構造体のために独自のローカル変数を維持する必要があります。

重要:

返されるバイト数は、*FIFO* に書き込むために残っているデータの量を示すものではありません。*FIFO* のメモリ *buffer* の最後まで残っているバイト数を反映している場合もあります。したがって、*pointer* が *buffer* の最後に近づくと、大幅に低い数になる可能性があります。

この関数はまた、fifo->position を設定して、FIFO の現在の書き込み位置を 0 から size-1 までの値で示します。ここで、`size` は fifo_init() に与えられた値です。ゼロ以外の fifo->slept は、呼び出し時に FIFO がいっぱいだったことを示します。

この関数は、書き込み可能なバイト数を返します (info->taken と同じ)。しかし、関数 fifo_done() が呼び出された場合、FIFO がいっぱいでなくても、fifo_request_write() は 0 を返します (決して読み取られない FIFO にデータを書き込む意味はありません)。

4.6.6 fifo_wrote()

fifo_wrote(struct xillyfifo *fifo, unsigned int req_bytes) – この関数は、req_bytes バイトの挿入を反映するように FIFO の状態を変更します。 FIFO が空だったために fifo_request_drain() がスリープ状態だった場合は、ウェイクアップされます。

重要:

req_bytes には **健全性チェックはありません**。 *req_bytes* が、*fifo_request_write()* への最後の関数呼び出しによって返された *info->bytes* よりも大きくないことを確認するのは、ユーザー アプリケーションの責任です。

この関数は void 型です (したがって、何も返されません)。

4.6.7 fifo_done()

fifo_done(struct xillyfifo *fifo) – この機能はオプションで使用でき、threads (読み取りまたは書き込み) のいずれかが終了した場合に、アプリケーションを正常に終了させるのに役立ちます。 FIFO の構造にフラグを設定するだけで、両方の threads がスリープしていた場合はそれらを起動します。そうすることで、FIFO が空の場合、fifo_request_drain() はスリープ状態ではなくゼロを返し、fifo_request_write() は関係なくゼロを返します。

このようにして、これらの関数の呼び出し元は、FIFO がもう使用されていないことを認識し、thread の実行を停止する可能性が最も高い、必要に応じて動作する可能性があります。

pipe に供給しているデータ ソースが終了したとき (たとえば EOF に達したとき)、またはデータ コンシューマーがもはや受け入れられなくなったとき (たとえば broken pipe) に、この関数を呼び出します。

この関数は void 型です (したがって、何も返されません)。

4.6.8 FIFO_BACKOFF define variable

FIFO が最後のバイトまでいっぱいになるのは望ましくない場合があります。それを回避する明確な理由はありませんが、データの書き込み先と読み取り元の間に小さなギャップを維持することが望ましい場合があります。

たとえば、FIFO_BACKOFF を 8 に設定できるため、FIFO に書き込まれた最後のバイトは、読み取り用の最初の有効なバイトと 64 ビット ワードを共有しません。これはかなり大げさな予防措置ですが、8 バイトのメモリという低価格で提供されます。

Xillybus または XillyUSB を使用する場合、この機能は必要ありません。

5

サイクリック frame buffers

5.1 序章

一部のアプリケーション、特にビデオ画像のリアルタイム処理では、多くの場合、buffers の数を維持して、各 buffer が固定サイズになるようにすることが望まれます。ビデオ処理アプリケーションでは、このような buffer ごとに 1 つの frame が含まれます。これにより、必要に応じて frames をスキップしたり、複数回再生したりできます。

frame grabber アプリケーションでは、buffer が空になるまで 1 つまたは複数の frames をスキップすることで、overflow 状態を処理できます。たとえば、ライブビュー アプリケーションでは、このような overflow 状態は、表示ウィンドウが移動またはサイズ変更されたときに発生する可能性があります。このように frames をドロップすると、小さな latency を維持しながら、ビデオ ソースからの継続的なデータ フローの中断を防ぐことができます。

frame replay アプリケーション (たとえば、 live output を表示する画面) では、表示する新しい frame がない場合、出力イメージが繰り返されます。これにより、ソース (ディスクなど) が一時的に停止し、表示された画像が短時間フリーズする状況が解決されます。完全に優雅というわけではありませんが、stream が同期しなくなるよりはましです。多くの場合、イメージの繰り返しメカニズムは、やや粗雑ですが、特に出力の frame rate が入力の frame rate よりもかなり高い場合 (たとえば、30 fps から 60 fps)、frame rates の違いを克服するためにうまく機能します。

このセクションでは、4.4 の段落で紹介した FIFO デモ アプリケーションを変更して、この種の buffers のセットを管理する方法について説明します。

5.2 FIFO サンプルコードの適応

frame buffers と FIFO の循環セットの維持には類似点があります。実際、FIFO の各バイトが frame buffer を表す場合、FIFO で特定のバイトを読み書きできる状態は、frame buffer 全体を読み書きできる状態と同じです。

たとえば、受信した画像データを格納するために 4 つの frame buffers が割り当てられている frame grabber アプリケーションを想定します。さらに、これら 4 つの frame buffers を次のように管理するために、4 バイトの FIFO が設定されているとします。

データを受信する thread は、最初の frame buffer から開始し、サイクリックに次の frame buffer に続きます。新しい frame buffer への書き込みを開始する前に、この thread は 4 バイトの FIFO がいっぱいでないことを確認します。 frame buffer を完了した後、FIFO にバイトを書き込み、FIFO がいっぱいでない場合は次のバイトに進みます。

画像データを消費する thread は、同じ順序で frame buffer を循環します。新しい frame buffer から読み取ろうとする前に、4 バイトの FIFO が空でないことを確認します。 frame buffer で終了し、次に進む準備ができたら、FIFO から 1 バイトを読み取ります。

この規則に従うことで、データを受信する thread が消費されていない frame buffer をオーバーランすることはなく、消費する thread が無効なデータを含む frame buffer から読み取ろうとしないことが保証されます。実際のところ、FIFO のバイト数は、セット内の有効な frame buffers の数を表しています。

書き込まれたバイトと読み取られたバイトの値は違いがないことに注意してください。したがって、これら 4 バイトのメモリを割り当ててデータを格納する必要は実際にはありません。 FIFO のハンドシェイク メカニズムだけが役割を果たします。

したがって、段落 4.6 で概説されている FIFO API はそのまま採用できます。

- size パラメーターを frame buffers の数として関数 fifo_init() を呼び出します (`size` は任意の integer になる可能性があることを思い出してください)。 fifo_init() は、決して使用されない FIFO にメモリを割り当ててロックします (各バイトは単に frame buffer を象徴するため)。このメモリの浪費は無視できますが、将来の混乱を避けるために、コード内の関連部分を削除できます。
- 関数 fifo_request_drain() を呼び出して、読み取る frame buffer を取得します。 info->position には、使用する frame buffer へのインデックスが含まれます (番号は 0 から始まります)。 frame buffer の準備ができていない場合、fifo_request_drain() は準備が整うまでスリープします。

- buffer から読み取った後、bytes_req=1 で関数 fifo_drained() を呼び出します。
- 関数 fifo_request_write() と fifo_wrote() は、frame buffers に書き込む thread によって同じ方法で呼び出されます。
- FIFO_BACKOFF はゼロに設定する必要があります。 frame buffers でこの機能を使用する意味はありません。

5.3 ドロップと frames の繰り返し

overflow の状態に決して到達してはならない image frames の継続的なソースの場合を考えてみましょう。

これは、データ ソースから frame buffers にデータを転送する thread でのブロックを防ぐためです。これを実現するには、着信 frame ごとに次のシーケンスをループする必要があります。

- 関数 fifo_request_write() を呼び出して、どの frame buffer に書き込むかを調べます。
- info->position が指す frame buffer への書き込み
- 書き込みが終了したら、関数 fifo_request_write() を再度呼び出します。前回の呼び出し以降、buffer への書き込みが報告されていないため、この関数呼び出しは確実にスリープ (block) しません。
- fifo_request_write() が 1 より大きい値を返した場合は、関数 fifo_wrote() を呼び出します (もちろん、req_bytes=1 を使用します)。 fifo_request_write() への後続の関数呼び出しは確実にスリープしません (block)。これは、予備の buffer が複数あり、消費されたのは 1 つだけであるためです。実際、fifo_request_write() への次の関数呼び出しは、次の frame buffer を選択するだけで置き換えることができます。
- 一方、fifo_request_write() が 1 だけを返す場合は、関数 fifo_wrote() を呼び出さないでください。代わりに、着信データを受け入れるために実行される次のループで現在の buffer を再度使用するか、frame 全体をデータ ソースから特定の宛先に排出しません。

この使用法によりブロッキングが防止されるため、fifo_request_write() の実装で while() ループが呼び出されないため、これを削除することができます。関連する semaphore とその初期化および破棄コードを削除することで、さらにコードを削減

できます。それらをコードに残しても影響は最小限であるため、この最適化は主にコードを読みやすくすることの問題です。

同様の方法で、FIFO からの thread 書き込みで frames を繰り返すことができます。関数 fifo_drained() を呼び出す直前に関数 fifo_request_drain() を再度呼び出し、現在の frame が 2 未満を返す場合はそれを繰り返します。

6

特定のプログラミング手法

6.1 Seekable streams

同期 Xillybus stream は、seekable として構成できます。 stream の位置は FPGA 内の application logic にアドレスとして個別のワイヤで示されるため、demo bundle とサンプル コードに示すように、FPGA 内のメモリ アレイまたは registers とのインターフェイスは簡単です。

この機能は、FPGA で control registers をセットアップする場合に特に便利です。stream の同期の性質により、低レベルの I/O 関数が戻る前に FPGA 内の register が設定されます。

次のコード スニペットは、メモリ内の `address` または FPGA 内の register space のアドレスに `len` バイトのデータを書き込む方法を示しています。これらの 2 つの変数は事前に設定されているものとします。

```
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` は、ファイルが書き込みまたは読み取り/書き込み用に開かれた open() への関数呼び出しから返された値であると想定され、`buf` は、書き込まれるデータを含む buffer を指しています。

この例は、段落 3.3 に示されている例の拡張です。

このコードで唯一特別なのは、アドレスを設定する lseek() への関数呼び出しです。lseek() 関数を呼び出すときは、SEEK_SET オプションのみを 3 番目の引数として使用する必要があります。

後続の関数呼び出しは、I/O stream の位置に従ってアドレスを更新するため、関数 lseek() を呼び出した後に複数の連続書き込みを行うことに制限はありません。

FPGA で 16 ビットまたは 32 ビット ワードとしてアクセスされる streams の場合、lseek() に指定されるアドレスは、それぞれ 2 または 4 の倍数でなければなりません。 FPGA で application logic に提示されるアドレスは、stream の I/O 位置 (最初は lseek() に与えられる) をそれぞれ 2 または 4 で割った値として常に維持されます。より広い単語については、同じ対数規則が適用されます。

tell() 関数は、stream 内の正しい位置 (つまり、現在のアドレス) を返すことがありますが、この情報の信頼できる情報源ではありません。疑わしい場合は、関数 lseek() を再度呼び出します。

lseek() は、データの読み取りに同じ方法で使用できます。デモ アプリケーション バンドルの memwrite.c と memread.c (および Getting started with Xillybus on a Linux hostの説明) を参照してください。

6.2 streams の両方向の同期

特定のアプリケーションでは、複数の streams をおそらく反対方向に同期させる必要があります。たとえば、無線伝送システムを host に実装し、RF 受信機に接続された A/D converter からデジタル サンプルを受信することができます。同様に、RF 送信機に接続された D/A converter にデジタル サンプルを送信している可能性があります。この種のシナリオでは、多くの場合、受信したサンプルとの関係で送信時間がわかるように、送信用のデジタル サンプルを生成する必要があります。受信信号の正確な時刻を知ることも重要です。

幸いなことに、単純な FPGA logic で実装できます。そのような解決策の 1 つは、送信用の最初のサンプルが FPGA に到着するまで、受信したデジタル サンプルを無視することです。

host は、FPGA からサンプルを読み取るために stream を開くことから始まります。 FPGA が受信サンプルをドロップするため、この stream はこの段階ではアイドル状態です。次に、host は、FPGA に送信するサンプルを書き込むために stream を開き、FPGA へのデータの書き込みを開始します。最初のサンプルが FPGA に到着すると、受信サンプルの無視を停止し、host への送信を開始します。

その結果、FPGA から読み取られる最初のサンプルは、FPGA に書き込まれる最初のサンプルと一致します。したがって、host 上のアプリケーションは、それぞれの stream 内の位置を一致させるだけで、送信用の任意のサンプルの timing を受信した任意のサンプルと一致させることができます。 FPGA の latency と A/D および D/A の遅延を補正するには、わずかな補正が必要になる場合がありますが、そのような latency は一定であり、既知です。

streams は常に連続している必要があります。これを実現する方法については、セクション 4 で説明しました。

送信と受信の間の相対的な時間関係を維持するだけで十分な場合は、このソリューションで十分です。サンプルを外部イベントまたは別の時間基準と同期させる必要がある場合は、サンプルをスキップするという同じ原則を必要に応じて適用して、目的の結果を得ることができます。

任意の時点で driver の buffers に保持されているデータ量の監視については、Xillybus FPGA designer's guideの “Monitoring the amount of buffered data” というセクションで説明されています。

6.3 パケット通信

一部のアプリケーションでは、data stream をさまざまな長さのパケットに分割する必要があります。推奨される解決策では、2 つの別個の streams を使用し、データの送信者がチャネルを通じてパケット自体を送信し始めるときに、パケットの長さを知る必要はありません。

既知の固定長のパケットの些細なケースは、単一の stream でパケットを次々と送信するだけで解決されます。反対側の受信機は、各パケットの固定数のワードを読み取るだけです。これは、video frame grabber または video replay アプリケーションの典型的なソリューションです。

さまざまな長さのパケットの場合、FPGA がバイトのパケットを host に送信する upstream アプリケーションを見てみましょう。 FPGA がパケットの長さを知っているのは、最後のバイトが到着したときだけだと仮定しましょう。

FPGA 側 (送信側) の実装は次のとおりです。

- FPGA は、パケット内のすべてのバイトを最初の Xillybus stream に書き込みます。
- FPGA は、パケットの最初のバイトを書き込むときにバイト カウンターをリセットし、追加のバイトを書き込むたびにインクリメントします。
- パケットの最後のバイトが書き込まれると、FPGA は 2 番目の Xillybus stream でカウンターの値を送信します。パケットの長さ (マイナス 1) が含まれます。

このソリューションの重要な属性は、FPGA が送信前にパケット全体を保存する必要がないことです。到着したデータをそのまま渡すだけです。

host のユーザー アプリケーションは、次のようにループを実行します。

- 次のパケットのバイト数を含む、2 番目の stream から 1 ワードを読み取ります。
- 必要に応じて、要求されたサイズの buffer にメモリを割り当てます。
- 最初の stream からパケット専用の buffer に指定されたバイト数を読み込みます。

host はデータにアクセスする前に読み取るバイト数をフェッチしますが、FPGA はこれらを逆の順序で streams に書き込みます。別の Xillybus streams を使用すると、この逆転が可能になります。

パケットが host から FPGA に送信される場合も、同様の配置が適用されます。データ用に 1 つ、バイト カウント用に 1 つ、2 つの streams を使用するという原則はそのままです。 FPGA の application logic は、もう一方の stream からデータを取得する前に、1 つの stream からバイト数を読み取ることができるようになりました。

この配置は、非データ stream で他のメタデータを渡すように拡張することもできます。たとえば、パケットの宛先やネットワークでのルーティングなどです (最初のバイトが到着したときに不明な場合があります)。

6.4 hardware interrupts のエミュレート

小規模なマイクロコントローラー プロジェクトでは、hardware interrupts を使用して、何かが発生したこと、およびソフトウェアが何らかのアクションを実行する必要があることをソフトウェアに警告するのが一般的です。ソフトウェアが Linux で userspace プロセスとして実行される場合、hardware interrupts は問題外であり、software interrupts でさえ、他の非同期イベントと同様に、それほど快適に処理することはできません。

Xillybus ベースのシステムで推奨される解決策は、メッセージを運ぶために特別な stream を割り当てることです。最も単純な形式では、hardware interrupt は専用の stream で 1 バイトを送信することによってエミュレートされます。

host 側では、userspace application が stream からデータを読み取ろうとします。その結果、“interrupt” が通知されない場合、アプリケーションはバイトが到着してウェイクアップするまでスリープ (blocking) します。アプリケーションはイベントを処理し、専用の stream から別のバイトを読み取ろうとするため、必要に応じて再びスリープ状態になります。

メイン アプリケーションと interrupt routine の間の適切な対話を実現するために、この専用の stream を別の software thread またはプロセスで読み取ることができます。この配置により、専用メッセージ stream から読み取る thread に関係なくメイン コードが流れ、送信されたメッセージに応じて後者がスリープおよびウェイクアップします。

このメソッドの変形では、送信されたバイトの値を使用して、エミュレートされた interrupt の性質に関する情報を渡します。また、実装で意味がある場合、各メッセージは 1 バイトより長くてもかまいません。

この方法は logic のリソースを浪費しているように見えるかもしれませんが、Xillybus は元々、このようなソリューションを合理的にするために、stream を追加するたびに logic をあまり消費しないように設計されていました。

6.5 Timeout

特定のアプリケーションでは、I/O 操作が blocking 状態のままになる時間を制限したい場合があります。特に、データ フローが停止する原因となる何らかのハードウェア障害が発生する可能性がある場合です。

Xillybus 自体は、データがこのように停止される理由ではないことを確認するために広範囲にテストされていますが、データ ソースとデータ コンシューマーはさまざまな理由で停止する可能性があります。

これに対処するあまり好ましくない方法は、select() または pselect() 関数を使用することです。これらは、複数の file descriptors を待機する必要がある場合に意図されていますが、timeout 機能も備えています。これらの関数を使用することはお勧めできません。それらの非自明なインターフェイスがバグの原因となる可能性があるためです。特に、timeout がキャッチする特殊なケースではそうです。

より自然な方法は、Linux の alarm 機能を使用することです。これはプロセスごとの timeout メカニズムであり、有効期限が切れたときに signal (software interrupt) をプロセスに送信します。 signal は、スリープ状態の read() または write() 関数呼び出しに強制的に制御を返すことを思い出してください (段落 3.2 および 3.3 を参照してください)。これらの関数は、負の値と errno を EINTR に設定して返します。前の例では、そのような割り込みは単なる障害でしたが、それでも timeout の実装には役立ちます。

どのプロセスも、その機能に関係のない複数の signals を受け取ることができます。 signal の受信自体は、timeout の状態を示すものではありません。見分ける方法はいくつかありますが、最も安全な方法は、その質問にまったく依存しないことです。I/O 操作に一定時間以上かかった場合、それは timeout です。したがって、最も簡単な戦略は、次に示す例のように時間を測定することです。これは、段落 3.2 から関数 read() を呼び出すものに基づいています。

この例の include files の一般的なリストは少し長くなります。

```
#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>
```

この例に固有の、次の宣言が必要です。

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

データを読み取るための while ループは、次のように開始されます。

```
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);
  }
```

時間は、clock_gettime() で関数 read() を呼び出す前後で測定されます。これは、単調な時間測定にアクセスできるため (システムユーティリティによって変更される system clock とは対照的に)、時間差を測定するのに適した関数です。この関数は、gcc の引数に `-lrt` フラグを追加する必要がある場合があるため、必要なライブラリをロードすることに注意してください。

alarm() への関数呼び出しは、2 秒後に signal を要求します (引数は秒数です)。各プロセスには alarm timer が 1 つしかないため、同じ timer の別の使用をオーバーライドしないように注意する必要があります。たとえば、一部の Linux 実装では sleep() です。

このコードは次のとおりです。

```
  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);
   }
```

時差が計算され、`elapsed` に格納されます。この単純な例では、語長の移植性の問題を回避するために double-precision floating point 変数を使用しています。ただ

し、これは integer でも実行できます。

条件は簡単で、時間計測の間隔が 2 秒以上経過していれば timeout です。 read() が返された理由は調べられていません。それは signal かもしれませんし、データが最終的に到着したのかもしれませんが、遅すぎます。どちらにしてもerrorです。

alarm() への関数呼び出しは、最初の測定が行われた後に行われたため、timeout は少なくとも 2 秒の長さの時間差を作ることが保証されていることに注意してください。

while ループは前と同じように続きます。

```
  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);
  }
}
```

上記のように、signals は引き続き無視されます。 timer がプロセスを起動した場合、時間差によって timeout の状態が明らかになり、終了します。

timeout を実装するこの方法は、multi-threaded 環境では複雑な問題になる UNIX signal に基づいていることに注意してください。複数の threads が展開されている場合は、そのうちの 1 つを他の watchdog にするのが最も簡単です。

また、上記の例では、timeout によってプロセスが終了されることに注意してください。これは、この操作を実行する signal handler で実装する方が簡単です。上記の方法は、実行中のプロセス内で修正応答が実行される場合に適しています。

timeout 間隔の精度を高めるには、代わりに setitimer() の使用を検討してください。

6.6 Coprocessing/ Hardware acceleration

Coprocessing (*hardware acceleration* とも呼ばれます) は、アプリケーションが logic fabric の柔軟性を利用して特定の操作をより高速に、より安価に、より少ないエネルギー消費で、または特定の processor よりも効率的に実行できるようにする

手法です。動機が何であれ、coprocessing を適切なソリューションにするためには、効率的なデータ転送フローが不可欠です。

coprocessing ベースのアプリケーションのデータ フローは、一般的なプログラミング データ フローとは根本的に異なることを認識することが重要です。この違いを説明するために、たとえば、floating point 表現で数値の平方根を計算する必要があるコンピューター プログラムを考えてみましょう。

プログラマーの簡単な方法は、数値を引数として sqrt() に渡し、それを呼び出し、関数が戻るまで待機することです。

代わりに、FPGA の logic fabric で平方根を計算したいとします。よくある間違いは、sqrt() を、計算用の値を FPGA に送信し、計算が完了するのを待ってから結果を返す特別な関数に置き換えることです。これは確かに sqrt() の単純なドロップイン代替品ですが、元の sqrt() よりも遅くなり、効率が低下する可能性が最も高くなります。データが bus を双方向に移動するのにかかる時間とFPGA が計算を行うのにかかる時間は、sqrt() が必要とする processor cycles よりもかなり長い可能性があります。そうは言っても、データフローが正しく設計されていれば、FPGA での平方根の計算ははるかに高速になります。

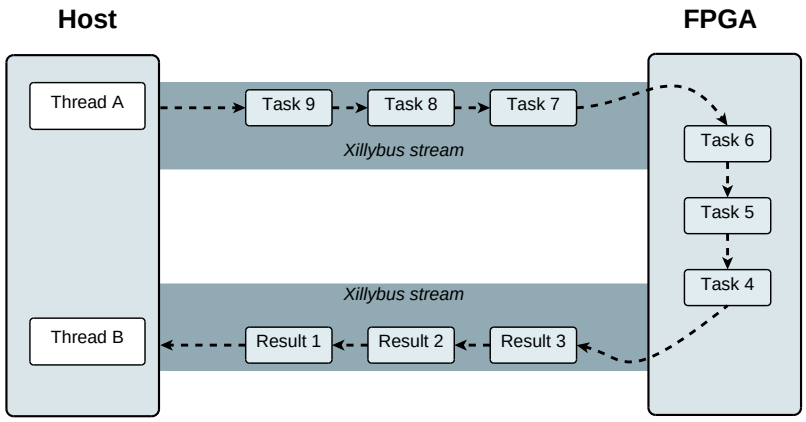
bus と FPGA の logic によって課せられた latencies を克服するには、ソフトウェアを再編成する必要があります。特に、単一の thread を持つプログラム内のタスクは、2 つ以上の threads (またはプロセス) に分割する必要があります。複数の threads が不可能または望ましくない場合は、multi-threading の動作を模倣するために他のプログラミング手法を利用できますが、それでもプログラミング パラダイムは multi-threaded です。

sqrt() の例に戻ると、この関数の呼び出しは 2 つの threads に分割されます。最初の thread は、平方根計算用のデータをハードウェア (または操作の要求を表す他の形式のデータ構造) に送信します。 2 番目の thread は、ハードウェアから結果を受け取り、アルゴリズムのその時点から処理を続行します。

これは、単一のデータを見るとあまり意味がないように思えますが、coprocessing の動機は、処理するデータ項目が多数あることを意味します。したがって、最初の thread は計算用のデータフローを送信し、2 番目の thread は結果のフローを受信します。

pipelining のこの手法は、ハードウェアの latency の影響を最小限に抑えます。これは、threads のいずれも、この latency の時間を効果的に待機していないためです。代わりに、latency は 2 つの threads 間の処理項目の量に影響を与えますが、スループットは 2 つの threads と FPGA logic の処理能力にのみ依存します。

次の概念図は、アイデアをまとめたものです。



sqrt() の加速計算は比較的単純な例ですが、coprocessing を利用する際の課題の多くをカバーしています。ほとんどの場合、コンピューター プログラムの大部分を書き直して、すべてが pipeline のデータ フローによって駆動されるようにする必要があります。

注意すべきもう 1 つの問題は、Xillybus は read() および write() で動作するため、FPGA に向けて stream に書き込む前に、計算のためにいくつかのデータ項目をグループ化することが有益である可能性があることです。同様に、各 read() 呼び出しで複数の結果アイテムを読み取ろうとすると、パフォーマンスが向上する場合があります。この背後にある理論的根拠は、read() と write() が特定のオーバーヘッドを持つ system calls であるということです。データ要素が小さく、高速で送信される場合、これらの system call のオーバーヘッドは相当なものになる可能性があります。 sqrt() の場合は、この良い例です。double float は通常、8 バイトの長さです。この長さの I/O system call は非常に非効率的であるため、単一の system call に対して複数の double float 要素を連結すると違いが生じます。

また、すべてのアプリケーションが一定の長さのデータ チャンクを含むわけではないことにも注意してください。たとえば、任意の文字列の hashes (たとえば SHA1) を計算するために coprocessing を使用すると、さまざまな長さで処理するためのデータ要素が含まれる可能性があります。セクション 6.3 では、これに対する解決策を提案しています。

A

内部: streams の実装方法

A.1 序章

Xillybus を使用するためにその実装の詳細を理解する必要はありませんが、一部の設計者は、好奇心や特定のソリューションの適格性を検証するために、内部で何が起こっているかを知りたいと考えています。

このセクションでは、DMA buffers に基づいて継続的な streams を作成するために実装された主な手法について概説します。 PCIe / AXI の場合は Xillybus に適用されますが、別のメカニズムを使用する XillyUSB には適用されません。

Xillybus の設計方法の目標は、基礎となるメカニズムをユーザーに対して透過的にすることであり、大部分はそれらを意識する理由はありません。 Xillybus を IP core として使用するために必要ではない可能性が非常に高いため、以下の技術的な詳細に進むときは、この点に留意してください。この部分では、それがどのように機能するかについて説明し、ユーザーが知る必要があることについては説明しません。

以下に、upstream フロー用と downstream 用の 2 つの主要なセクションがあります。同様の手法が両方向で使用されるため、一方のセクションの多くは他方のセクションの繰り返しです。

説明を簡単にするために、特に断りのない限り、説明は非同期 streams に焦点を当てています。 end-of-file 信号と non-blocking I/O のオプションについては、ここでは説明しません。

A.2 “Classic” DMA 対 Xillybus

従来、ハードウェアとソフトウェア間のデータ転送は、固定サイズの buffers の数の形式をとっていました。データは固定長の buffers に編成されます。 buffer の準

備が整うたびに、何らかの信号が反対側に送信されます。たとえば、ハードウェアが buffer への書き込みを終了した場合、interrupt を processor に送信して、データを処理する準備ができていることをソフトウェアに通知できます。ソフトウェアはデータを消費し、buffer に再び書き込むことができることをハードウェアに通知します。通常は、メモリ マップされた register に書き込むことによって行われます。通常、両側が round-robin 方式で buffers にアクセスします。

Xillybus は、FPGA 側とソフトウェア側の両方で、ユーザー インターフェイスに継続的な stream トランスポートを提供します。内部的には、Xillybus は従来の round-robin パラダイムと DMA buffers のセットを使用しています。

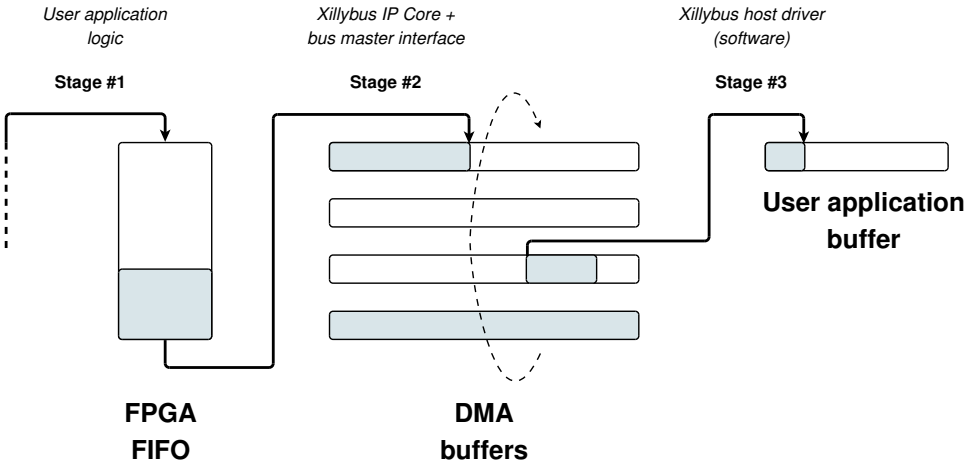
ただし、以下で説明する手法は、連続した stream の錯覚を作成するために使用されるため、ユーザーは基礎となるデータ トランスポートの存在を無視できます。特に、アプリケーションが固定サイズのチャンクでデータを送信する場合でも、以下で説明するように、DMA buffers のサイズをアプリケーション データに一致させる必要はありません。

A.3 FPGA host (upstream)

A.3.1 概要

以下の図は、FPGA から host (upstream) 方向への流れを示しています。影付きの領域は、それぞれのストレージ要素でまだ消費されていないデータを表します。

この例では、4 つの DMA buffers が示されていますが、これらの数は IP Core Factory で構成できます。



次に詳しく説明するように、データは 3 段階で host に向かって流れます。

A.3.2 ステージ #1: Application logic から中間 FIFO

FPGA の user application logic は、user application logic と Xillybus IP core を接続する FIFO にデータをプッシュします。 overflow を回避するために FIFO の “full” 信号を尊重することを除いて、いつ、またはどのくらいのデータがプッシュされるかについての要件はありません。

A.3.3 ステージ #2: 中間 FIFO から DMA buffer

この段階で、Xillybus IP core は FIFO から host の RAM の DMA buffer にデータをコピーします。これを達成するために、core は bus master インターフェイス (PCIe、AXI4 など) を使用して、host の processor の介入なしに、host のメモリに直接データを書き込みます。

host の RAM には、DMA buffers のプールが割り当てられます。各 DMA buffer のライフサイクルは、多くの同様の設定と似ています。当初、すべての DMA buffers は空であり、概念的にハードウェアに属しています。ハードウェアは、round-robin 方式で buffers にデータを書き込みます。特定の buffer への書き込みが完了すると、buffer が使用可能になったことを host に通知し (いわば、buffer は host に引き渡されます)、その後、次の buffer で引き続き書き込みます。次に、host は、渡された buffer のデータを消費することができます。その後、buffer に再び書き込むことができることをハードウェアに通知します (host は buffer をハードウェアに返します)。

ステージ #2 のデータ フローは、FIFO の “empty” 信号と、DMA buffers のプール内のスペースの可用性によって制御されます。 Xillybus IP core が FIFO からの低 “empty” 信号を感知し、一部の DMA buffer にスペースが残っている場合、FIFO からデータをフェッチし、それを DMA buffer に書き込みます。 FIFO が再び空になるか、DMA buffer にスペースがなくなると、IP core の内部ステート マシンはデータのフェッチを一時的に停止し、DMA buffer で中断したところから続行します。

データ フローが停滞している間、IP core は、他の stream に代わってデータをコピーする (つまり、別の中間 FIFO をドレインする) など、他のアクティビティでビジー状態になる可能性があります。その結果、FIFO が “empty” 信号を Low に変更してからデータのフェッチを再開するまでの間に、ランダムな遅延が生じる可能性があります。この遅延はさまざまですが、全体として、IP core は、512 ワードの FIFO が overflow の状態に達しないことを保証します (平均レートが制限内にある限り)。

各 DMA buffer は、host に渡す前に完全に記入するか、部分的に記入して host に提出することができます。部分的に満たされた buffer を引き渡す条件については、ソフトウェアの動作をある程度理解する必要があるため、後で詳しく説明します (セクション A.3.5)。

同期 streams の場合は、Xillybus IP core が中間の FIFO からデータをフェッチする前に、一定量のデータに対する明示的な要求を待機することを除いて、非常に似ています。

A.3.4 ステージ #3: DMA buffer からユーザー ソフトウェア アプリケーションへ

このステージは、read() system calls (または Microsoft Windows の対応する IRP) に応答することにより、host 上の Xillybus の driver に実装されます。確立された API によると、read() 要求には、user application によって提供される buffer と、読み取る最大バイト数でもある buffer のサイズが含まれます。関数呼び出しは、最大バイト数 (完全なフルフィルメント) またはそれ以下を読み取った後に戻る場合があります。

driver は、read() 要求の完全な履行を可能にするために、DMA buffers で消費するのに十分なデータがあるかどうかを判断するために、渡された DMA buffers をチェックすることから始めます。その場合、データをユーザーの buffer にコピーし、おそらく DMA buffers をハードウェアに戻し、system call から戻ります。

それ以外の場合、read() 関数呼び出しの標準 API では、driver が要求されたバイト数未満で戻るか、任意の期間待機 (スリープ) することができます。 driver は、少ないデータであまり頻繁に返らないように設計されています (これにより、それぞれ少ないデータで多くの read() 関数呼び出しが発生し、CPU サイクルが浪費される可能性があります) が、不要な latency も回避します。ジレンマは、read() 関数呼び出しで必要とされるよりも DMA buffers 内のデータが少ない場合に何をすべきかということです。部分的な履行または待機で戻る (および待機する量)。

選択された戦略は、最大 10 ms までさらにデータを待機し、利用可能なデータを返すことです (または、標準の API が要求するように、データが利用できない場合は無期限に待機します)。これにより、かなり応答性の高い戻り時間になりますが、read() 関数の呼び出し元が常に利用可能なデータよりも多くのデータを要求する場合、オーバーヘッドは 1 秒あたり 100 read() 関数呼び出しに制限されます。

これは、read() 関数呼び出しが必ず 10 ms の latency を持っていると言っているわけではありません。user space application が準備できるバイト数を事前に知っている場合、その数を超えて要求することはできません。そうすることで、マイクロ秒単位の latency が保証されます。

ただし、注意が必要な部分があります。host は、引き渡された DMA buffers を認識していますが、host が認識していない部分的に満たされた DMA buffer が存在する可能性があります。そのため、部分的に満たされた DMA buffer を考慮すると、read() 関数呼び出しを完全に実行するのに十分なデータが実際に存在する可能性があります。

このケースを適切に処理するために、driver は、不足しているバイト数が部分的に満たされた buffer に収まるかどうかをチェックします。これが実際に当てはまる場合は、どれだけのデータがあれば十分かをハードウェアに通知します。次に、driver は 10 ms 待機を開始します。これにより、read() 関数呼び出しを完全に完了することが実際に許可されている場合、ハードウェアは部分的に満たされた buffer をすぐに送信する機会が与えられます。

部分的に満たされた buffer が必要な量に達した場合 (おそらくすぐに)、ハードウェアはそれを host に渡し、read() 関数呼び出しをすぐに完了します。

10 ms の期間が終了すると、driver は利用可能なデータをすべて持って戻ります。データがまったくない場合、driver はハードウェアに要求を送信して、部分的に満たされた buffer を渡します。 10 msの期間はすでに終わっているので、データがあればすぐに返すのが目的です。

どのような状況でも、DMA buffer が完全に消費されると、driver はそれをハードウェアに返します (つまり、再度書き込み可能であることをハードウェアに通知します)。

同期 streams について一言: read() 関数呼び出しが呼び出されたときにデータが DMA buffers で使用できないことを除いて、フローは原則として同じです。これは、ハードウェアが指示された場合にのみ、FPGA の FIFO からデータをコピーできるためです。したがって、同期 streams の read() 関数呼び出しには、コピーする必要があるデータの量をハードウェアに通知することが含まれます。待機メカニズムは同じままです。最初に 10 ms、次に部分的に満たされた buffer を要求します。

A.3.5 部分的に満たされた buffers の引き渡し条件

部分的に満たされた buffers を引き渡すケースは、上記から推測でき、便宜上ここにリストされています。

一般的なルールは、ハードウェアがそのような早期のサブミットによって read() 関数呼び出しがすぐに返されることを通知されている場合、部分的な buffer が host に引き渡されることです。これは、次の 3 つの条件のいずれかで発生します。

- host は現在、read() 関数呼び出しを処理しています。これは、現在の部分的に満たされた buffer が引き渡されたときに完全に実行されます。

- read() 関数呼び出しは 0 バイトで、制限時間 (つまり 10 ms) に達しました。
- 同期 streams のみ: ハードウェアが host によって要求された量のフェッチを完了したとき。

FIFO が空になった場合、それ自体は DMA buffer 提出の理由にはならないことに注意してください。

A.3.6 例

8 ビット非同期 stream の次の単純なケースを考えてみましょう。 stream がデータを含まない状態で始まり、その後 FIFO が 1 つの要素 (つまり、1 バイト) で埋められるとします。次に、host のアプリケーション プログラムが read() 関数を呼び出し、1 バイトを要求します。これは可能な一連のイベントです。

- Xillybus IP core は低い “empty” 信号を検出するため、FIFO から 1 バイトをフェッチした後、再び空になります。
- バイトは、DMA で、DMA buffer の最初の位置に書き込まれます。 buffer がいっぱいではないため、host には通知されません。
- read() 関数呼び出しが host で呼び出され、1 バイトが要求されます。
- driver にはデータを取得する DMA buffer がありません。データ (1 バイト) を含む DMA buffer のみがハードウェアに認識されます。
- driver は、必要なデータの量が DMA buffer のサイズよりも少ないことを検出し、そのため、少なくとも 1 バイトがあれば、部分的に満たされた buffer を渡すようにハードウェアに指示します。
- driver は 10 ms スリープを開始し、何かが起こるのを待ちます。
- ハードウェアは、部分的に満たされた buffer を host に渡すことで即座に応答します。
- driver はすぐに起動し、要求された 1 バイトを read() 関数呼び出しで提供された buffer にコピーして戻ります。

この単純な例は、データのサイズが DMA buffer よりも大幅に小さいにもかかわらず、read() 関数呼び出しが事実上即座に返される方法を示しています。

例をもう一度見てみましょう。小さな違いが 1 つあります。FIFO には 1 バイトだけが書き込まれますが、read() 関数呼び出しは 2 バイトを要求します。シーケンスは次のとおりです。

- Xillybus IP core は低い “empty” 信号を検出するため、FIFO から 1 バイトをフェッチした後、再び空になります。
- バイトは、DMA で、DMA buffer の最初の位置に書き込まれます。 buffer がいっぱいではないため、host には通知されません。
- read() 関数呼び出しが host で呼び出され、 *2*バイトが要求されます。
- driver にはデータを取得する DMA buffer がありません。データ (1 バイト) を含む DMA buffer のみがハードウェアに認識されます。
- driver は、必要なデータ量が DMA buffer のサイズよりも少ないことを検出し、そのため、少なくとも*2* バイトがあれば、部分的に満たされた buffer を引き渡すようにハードウェアに指示します。
- driver は 10 ms スリープを開始し、何かが起こるのを待ちます。
- DMA buffer には 1 バイトしかないため、ハードウェアは何もしませんが、2 バイトが要求されました。
- driver は 10 ms の後に起動し、何もありません。 buffer が空でない限り、ハードウェアに要求を送信して、部分的に満たされた buffer をできるだけ早く引き渡します。
- ハードウェアは、部分的に満たされた buffer を host に渡すことで即座に応答します。
- driver はすぐに復帰し、要求された 1 バイトを関数呼び出し元の buffer にコピーして、戻ります。

この 2 番目の例は、実際には 1 バイトしかないときに 2 バイトを要求した結果を示しています。関数呼び出しは、10 ms の後でのみ 1 バイトで戻ります。ただし、この遅延はほとんどの実際のシナリオでは気付かれないことに注意してください。

A.3.7 実際的な結論

- アプリケーション レベルのデータが常に N バイトのチャンクで構成されている場合でも、DMA buffer のサイズを変更する必要はありません。 user application software は、read() 関数呼び出しが必要なデータ量を正確に要求するようにするだけでよく、部分的な buffer メカニズムは、データが FPGA の FIFO にプッシュされたときに関数呼び出しが返されることを確認し、latency は非常に低くなります。

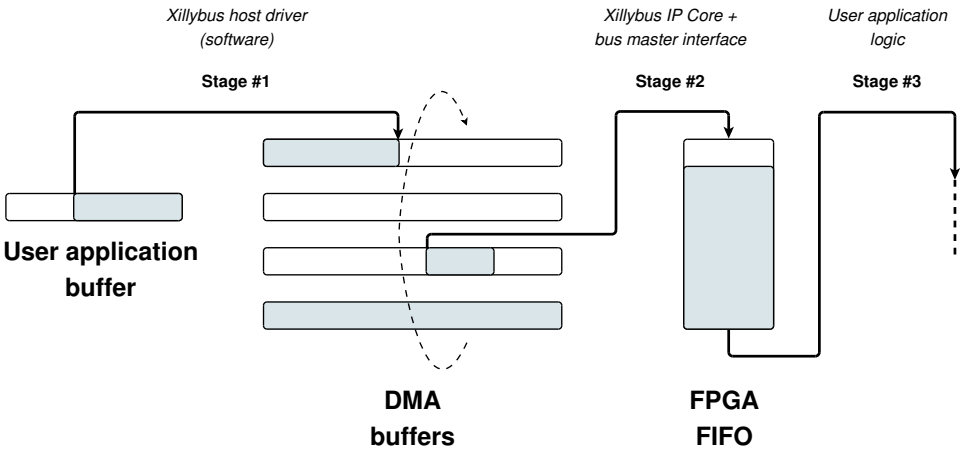
- データの継続的な streams の場合でも、latency は小さな buffers で read() 関数呼び出しを行うことで削減できますが、追加のオペレーティング システムのオーバーヘッドが犠牲になります。 DMA buffers のサイズに関係なく、latency は read() 関数呼び出しで要求されるデータ レートとバイト数のみに依存します。 read() 関数呼び出しを完全に実行できない場合、read() 関数呼び出しは 10 ms まで待機し続けるため、DMA buffer サイズを小さくしても役に立ちません。
- 10 ms が受け入れ可能な latency である場合、返すデータがまったくない場合を除き、read() 関数呼び出しはこの期間の後に戻ることが保証されているため、最適化しても意味がありません。

A.4 Host FPGA (downstream)

A.4.1 概要

次の図は、host から FPGA (downstream) 方向のデータ フローを示しています。影付きの領域は、それぞれのストレージ要素でまだ消費されていないデータを表します。

この例では、4 つの DMA buffers が示されていますが、これらの数は IP Core Factory で構成できます。



前述のように、データは host から FPGA に 3 段階で流れます。詳細は次のとおりです。

A.4.2 Stage #1: DMA buffer へのユーザー ソフトウェア アプリケーション

このステージは、write() system calls (または Microsoft Windows の対応する IRP) に応答することにより、Xillybus の driver (host) に実装されます。確立された API によると、write() 関数呼び出し要求には、ユーザー アプリケーションによって提供される buffer と、書き込む最大バイト数でもある buffer のサイズが含まれます。関数呼び出しは、最大バイト数 (完全なフルフィルメント) 以下を書き込んだ後に戻る場合があります。

DMA buffers のプールは、host の RAM メモリに割り当てられます。各 DMA buffer のライフサイクルは、多くの同様の設定と似ています。最初は、すべての DMA buffers は空であり、概念的には host に属しています。 host は、round-robin 方式で buffers にデータを書き込みます。特定の buffer への書き込みが完了すると、buffer が使用可能になったことをハードウェアに通知し (いわば、buffer がハードウェアに引き渡されます)、その後、次の buffer で引き続き書き込みます。ハードウェアは buffer のデータを消費し、その後、buffer に再び書き込むことができることを host に通知します (ハードウェアは buffer を host に返します)。

Xillybus の driver は、できるだけ多くのデータを DMA buffers にコピーしようとすることで、write() 関数呼び出しに応答します。 DMA buffer が完全にいっぱいになると、ハードウェアに引き渡されます。つまり、host は、buffer を消費できることをハードウェアに通知し、ハードウェアが buffer を host に戻す前に再度書き込みを行わないことを保証します。

driver が DMA buffer スペースを使い果たす前に少なくとも 1 バイトを書き込むことができた場合、write() 関数呼び出しは書き込まれたバイト数を返します。それ以外の場合は、DMA buffer が書き込み可能になるまで無期限に (スリープ、つまり “blocking” によって) 待機し、可能な限り多くのデータを DMA buffer に書き込み、戻ります。

DMA buffer が部分的に満たされている場合、write() 関数呼び出しの最後にハードウェアに渡されないため、1 つの DMA buffer にハードウェアが認識していないデータが存在する可能性があることに注意してください。 “flush” オペレーションは、部分的に満たされた buffer を引き渡します。これは、次の 4 つのケースのいずれかで行われます。

- 明示的な flush。書き込むバイトがゼロの write() 関数呼び出しが原因です。この write() 関数呼び出しはすぐに戻ります (つまり、データが FPGA によって消費されるのを待ちません)。
- 自動 flush は、最後の write() 関数呼び出しの後に 10 ms で開始されます。
- ファイルが閉じられると、flush が発生します。このシナリオでは、close() 関

数呼び出しは、戻る前にデータが FPGA によって完全に消費されるまで最大 1 秒間待機します。

- 同期 streams では、write() へのすべての関数呼び出しは flush() で終了し、データが FPGA によって完全に消費されるまで無期限に待機します。

長さゼロの buffer を使用した write() 関数呼び出しは、明示的な flush を強制し、書き込まれたすべてのデータが FPGA で利用できるようにすることに注意してください。ただし、データが FPGA によっていつ消費されるかをアプリケーション ソフトウェアに示すことはありません。このような同期が必要な場合は、同期 stream を使用する必要があります。

A.4.3 ステージ #2: DMA buffer から中間 FIFO

この段階で、Xillybus IP core は host の RAM の DMA buffers から FPGA の FIFO にデータをコピーします。これを実現するために、core は、host の processor の介入なしに、host のメモリから直接データを読み取るために、いくつかの bus master インターフェイス (PCIe、AXI4 など) を使用します。

ステージ #2 のデータ フローは、FIFO の “full” 信号と、FPGA に属する DMA buffers のプール内のデータの可用性によって制御されます。 Xillybus IP core が FIFO からの低 “full” 信号を感知し、一部の DMA buffer にデータの準備ができている場合、DMA buffer からデータをフェッチし、FIFO に書き込みます。 FIFO が再びいっぱいになるか、DMA buffers が空になると、IP core の内部ステート マシンはデータのフェッチを一時的に停止し、DMA buffer プールで中断したところから続行します。

データ フローが停滞している間、IP core は、他の stream に代わってデータをコピーする (つまり、別の中間 FIFO を満たす) など、他のアクティビティでビジー状態になる場合があります。その結果、FIFO が “full” 信号を Low に変更してからデータ コピーが再開されるまでの間に、ランダムな遅延が生じる可能性があります。この遅延はさまざまですが、全体として、IP core は 512 ワードの FIFO が十分に深いことを保証します。

もちろん、ハードウェアは部分的に満たされた DMA buffers を認識しており、それぞれに含まれるデータの量を追跡します。

A.4.4 ステージ #3: 中間 FIFO から application logic

FPGA 内の user application logic は、user application logic と Xillybus IP core を接続する FIFO からデータを取得します。 underflow を回避するために FIFO の “empty”

信号を尊重することを除いて、いつ、またはどれだけのデータがフェッチされるかについての要件はありません。

A.4.5 例

次の 8 ビット非同期 stream の単純なケースを考えてみましょう。 stream がデータを含まない状態で開始し、その後 host のアプリケーションが device file に 1 バイトを書き込むとします。

イベントのシーケンスは次のとおりです。

- driver の write() 関数呼び出しは、1 バイトの書き込み要求で呼び出されます。
- stream にはデータが含まれていないため、明らかに DMA buffers にはスペースがあります。 したがって、driver はバイトを最初の DMA buffer にコピーして戻ります。
- 10 ms の間は何も起こりません。
- autoflush メカニズムは 10 ms の後にトリガーされ、driver が DMA buffer に 1 バイトが含まれているという情報をハードウェアに渡します。
- Xillybus IP core は DMA buffer からバイトを読み取り、それを中間の FIFO に書き込みます。
- application logic は FIFO から自由にバイトを読み取ることができます。

A.4.6 実際的な結論

- アプリケーション レベルのデータが常に N バイトのチャンクで構成されている場合でも、DMA buffer のサイズを変更する必要はありません。 user application software は、データの flush を要求するだけでよく、各チャンクの最後にゼロバイトを要求する write() 関数呼び出しが必要です。 マイクロ秒単位の latency は、この方法で実現されます。
- データの連続 streams の場合でも、write() 関数呼び出しを小さな buffers で行い、その後に flush (0 バイトでの write() 関数呼び出し) を行うことで latency を減らすことができますが、追加のオペレーティング システムのオーバーヘッドが犠牲になります。 DMA buffers のサイズに関係なく、latency はデータ レートと flush 要求間のデータ量のみに依存します。

- flush が常に特定のデータ チャンクの後に発生し、DMA buffer が特定のレベルを超えていっぱいになることがないことが事前にわかっている場合は、DMA buffer のサイズを小さくすることは理にかなっています。ただし、そうすることの唯一の利点は、host で RAM の量を節約できることです。これは重要ではありません。
- 10 ms が許容可能な latency である場合、autoflushing メカニズムは 10 ms のアクティビティがなくなった後に作動するため、最適化しても意味がありません。