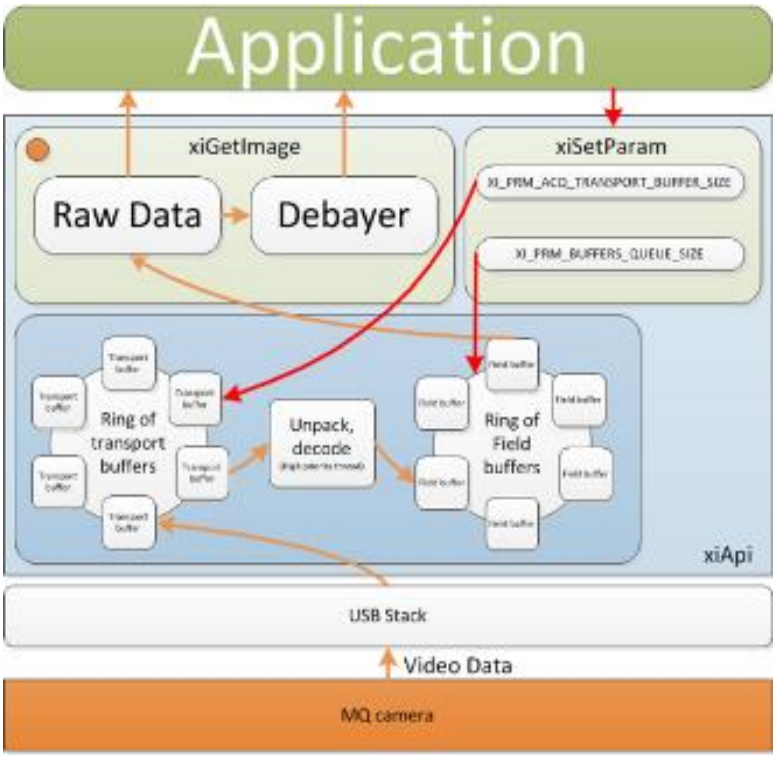
https://www.ximea.com/support/wiki/usb3/Buffer_-_lost_frames




# ロストフレーム 及び どのくらいの量のデータがカメラ内部にバッファーする?



カメラ内部に大容量のメモリ バッファがあり、複数の画像フレームを保存できると、アプリケーションでフレームが失われることはないというのが一般的な考えです。
このアプリケーション ノートでは、フレームの損失、その原因と対策について詳しく説明します。
この懸案には、CameraLink、FireWire、USB、GigE、CoaXPress、PCI Express、Thunderbolt など、関連するカメラ インターフェイスの観点から対処します。

# 理由 原因

フレームが失われる原因は何でしょうか。それはすべて、**帯域幅**、**レイテンシ**、**バッファ処理**に関するものです。

帯域幅とレイテンシは相互に依存していますが、常に 1 つ以上の根本的な原因があります。これらは、ハードウェア関連とソフトウェア関連に分けることができます。

**帯域幅**と**ハードウェア**側:

## 1. 帯域幅リミット

インターフェイスでカメラに使用できる帯域幅が配信されるデータの量よりも低い場合、そのような設定では常にフレームが失われます。

## 2. 遅延　レイテンシ

使用可能な帯域幅が要求された帯域幅と等しい場合、ハードウェア インターフェイスからの要求を処理する際に遅延を許容できないため、このようなシステムはフレームを欠損します。

## 3. 信頼性マージン

使用可能な帯域幅に要求された帯域幅を超えるマージンがある場合、このようなシステムはフレームを欠損することなく確実に動作します。このマージンがどのくらいの大きさであるべきか、またそれがカメラ バッファ サイズに関係するかどうかについては、以下で説明します。

# ハードウェア

帯域幅は PC ホスト側から提供され、カメラによって使用されます。**帯域幅**は、次の**ハードウェア**上の理由によって制限される可能性があります:

## 1. 通信リンク

エラー率が高い、または通信リンクが信頼できないために、トランスポート パケット* が失われます。

この問題は、トランスポート パケットが失われた理由をトラブルシューティングすることで解決する必要があります。GigE カメラで使用される UDP などのコネクションレス プロトコルでは、すべてのパケットが到着したかどうかをアプリケーション レベルで追跡し、パケットが一定時間内に到着しなかった場合はパケットまたはフレーム全体の再送信を要求する必要があります。

**USB バルク転送**などのトランザクション ベースのプロトコルでは、**パケットが失われることはありません**。USB 3.0 インターフェイスの設計により、パケット配信の **100% 信頼できるプロトコル**が存在します。

このプロトコルは、USB 3.0 トランザクション レイヤーで実装されています。数回の再試行後にパケットの配信が失敗した場合、ホストとカメラの両方がその失敗を認識し、このパケットの再送信を試みたり、API にエラーを報告したりすることがあります。

パケットが送信されない理由としては、SuperSpeed 信号の信号整合性、ファームウェアの問題、電磁耐性、ボード レイアウトなどのハードウェア設計の堅牢性、適切な電源レールの分離、高速信号インピーダンスなどが考えられます。

これらの側面はカメラ側とホスト側の両方に当てはまりますが、最後に重要なのがケーブルです。ケーブルのパフォーマンスは、高速信号を減衰させ、カメラに十分な電力を供給することで、通信の信頼性に大きく影響します。

## 2. その他のデバイス

同じバス上に他のデバイスがあり、帯域幅やその他の共有リソースをめぐって競合しています。これは、各カメラを個別の xHCI コントローラーに配置することで解決できます。

### 3. ファームウェア

カメラのファームウェアは、ホストがサポートする最大速度でデータをストリーミングできないため、トランスポート パケットのコミット時に遅延が発生し、帯域幅が減少します。

PC ハードウェア自体は遅延を引き起こしません。

## ソフトウェア

**ソフトウェア**は、新しいカメラ データを受信するために空のバッファーを提供するという要求を処理するための**レイテンシ**に影響します。PC ソフトウェア (USB プロトコル スタックやカメラ ドライバー、API レベルのバッファー処理ルーチンなど) のこれらのレイテンシーにより、片側で使用可能な帯域幅が実質的に減少します。

### 1. インターフェイス

ハードウェア インターフェイス カードからの割り込みを処理するためのレイテンシー。
FireWire、USB、GigE などのカメラ インターフェイスは、それぞれのホスト アダプターを使用しています。これらのアダプターには、割り込みを処理するための最大レイテンシの要件が異なります。その中で最も重要で時間的に重要なのが、新しいデータを配信するための記述子を取得する要求です。レイテンシは、コンピューター、オペレーティング システム、ソフトウェア ドライバー、プロトコル、ソフトウェア スタックのパフォーマンスによって異なります。

PCI Express や Thunderbolt などのカメラ インターフェイスは、中間バッファーやバス プロトコルを使用していません。これら 2 つのインターフェイスには、大幅に少ないソフトウェア コンポーネントがあり、設計上、レイテンシがはるかに短くなっています。

### 2. アプリケーション

アプリケーションのレイテンシ　- アプリケーションが各フレームからのデータ処理のループにあり、必要なタイムスロット内にこのタスクを完了できない場合。これは、処理アルゴリズムが設計上、時間決定論的でない場合に発生する可能性があります。別の理由としては、基盤となるオペレーティング システムが処理タイムスロット内に必要な量の CPU リソースを提供できないことが考えられます。

## バッファ処理

結局のところ、ソフトウェアの側面はバッファ処理方法に集約されます。
バッファ処理には、2 つの一般的な実装があります。

### 1. 割り当て

アプリケーションは、コンピュータ メモリに画像バッファを割り当て、これらのバッファをカメラ キューにコミットし、バッファが画像データとともにキューから出るのを待ちます。次に、アプリケーションはデータを処理して、「空の」バッファを再度コミットする必要があります。アプリケーションが「低速」で、すべての受信フレームを時間内に処理できない場合に問題が発生します。
アプリケーションがバーストでデータを取得して処理する必要がある場合、アプリケーションは 2 回 (またはそれ以上) の連続バーストに必要な数のバッファを割り当て、それらすべてをカメラにコミットします。これにより、すべてのデータが配信され、バースト間に余裕のある時間を確保して処理を完了し、処理されたすべてのバッファを再度コミットできます。

## 2. API

カメラ API は、カメラから送られてくる画像データを順番に埋めていくバッファの循環キューを割り当てます。アプリケーションが新しい画像データを取得したい場合、最初のバッファが画像データでいっぱいになるまで待機するか、すでに画像データが入っているバッファがある場合はすぐに戻ります。API は、どの画像がアプリケーションに最後に配信されたか、どの画像が現在画像データでいっぱいになっているかを追跡します。アプリケーションが遅すぎて循環キュー全体がすでにデータでいっぱいになっている場合、API は「オーバーラン」状況を検出します。

# まとめ

これまでのすべての懸案事項をまとめると、次のようになります。

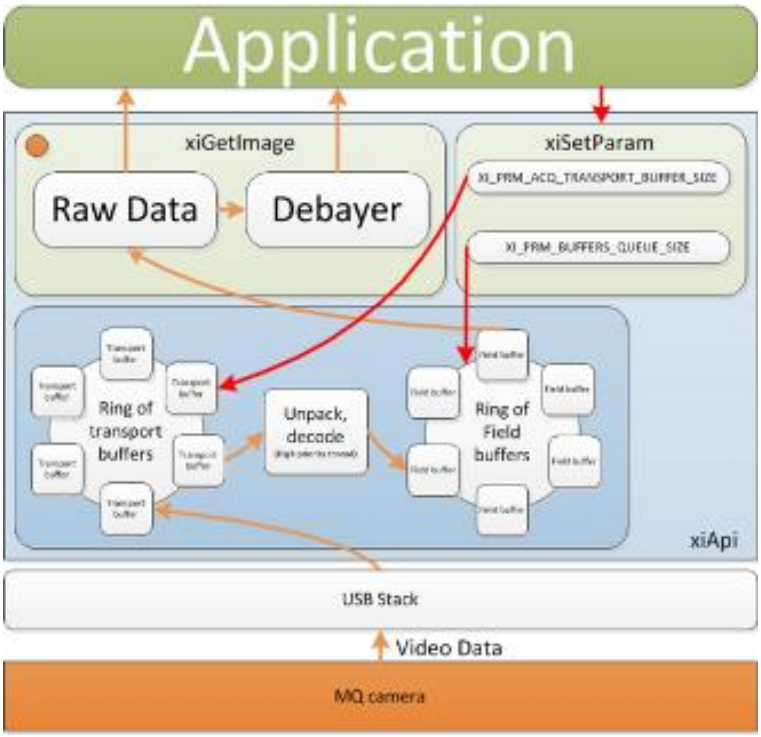
**1. 接続のないプロトコル** (GigE の UDP) の場合、パケットまたはフレームが失われた場合に再送信できるように、カメラ内に**大きな画像バッファを用意することが重要**です。

**2. USB 3.0** などのトランザクション レイヤーでパケットの再送信を保証するプロトコルの場合、カメラ内に**大きな画像バッファがあってもメリットはありません。**必要なのは、バスの可用性のジッターを許容する FIFO です。

**3.** 信頼性の高い高速アプリケーションを構築するには、**帯域幅とレイテンシの両方を**分析してトラブルシューティングする必要があります。
**これらが分析されていない場合、内部に大きな画像バッファがあるかどうかに関係なく、遅かれ早かれカメラはフレームを失います。**
**バッファ処理方法も、アプリケーションの要件に合わせて調整する必要があります。**

XIMEA API、ドライバー、カメラファームウェアは、サポートされているホストコントローラーからのすべての帯域幅を利用し、レイテンシを可能な限り低くするように調整されています。

https://www.ximea.com/support/wiki/usb3/Buffer_-_lost_frames




# Lost Frames and How Much Data to Buffer Inside the Camera?



There is a common belief that a bigger memory buffer located inside the camera and capable of storing several image frames ensures that application will never lose frames. This application note elaborates on the subject of lost frames, its reasons and remedy. We will address this question from the perspective of relevant camera interface, namely CameraLink, FireWire, USB, GigE, CoaXPress, PCI Express and Thunderbolt.

# Reasons

What are the reasons for the lost frames? It is all about **bandwidth, latencies** and **buffer handling**.
Bandwidth and latencies are mutually dependent, but always have one or more underlying root causes. These can be divided into hardware and software related.
On the **bandwidth** and **hardware** side:

## 1. Bandwidth limit

If the available to the camera bandwidth on the interface is lower than the amount of data to be delivered, such setup will always lose frames.

## 2. Latency

If available bandwidth is equal to the required, such system again will lose frames since it cannot tolerate any latency in servicing request from the hardware interface.

## 3. Reliability margin

If available bandwidth has a margin above the required one, such system can reliably work without lost frames. The question how big shall be this margin and whether it is related to the camera buffer size will be addressed below.

# Hardware

The **bandwidth** is provided by the PC host side and utilized by the camera. Bandwidth can be limited by the following **hardware** reasons:

## 1. Communication link

Loss of transport packets* because of the high error rate or unreliable communication link.
This problem must be addressed by troubleshooting the reason for the loss of transport packets. Connectionless protocols like UDP used by GigE cameras require application level to track whether all packets have arrived and request resending of a packet, or a whole frame, if a packet did not arrive within certain time.
For transaction based protocols, like **USB bulk transfer, packet cannot be lost**. There is **100% reliable protocol** of packet delivery by design of the USB 3.0 interface.
This protocol is implemented at the USB 3.0 transaction layer. If delivery of a packed failed after several retries, then both host and camera are aware about that failure and may attempt to resend this packet, and/or report an error to the API.
Reasons for not delivered packet can be signal integrity of SuperSpeed signals, firmware issues, EM immunity, hardware design robustness like board layout, proper power rails decoupling, high speed signal impedance, etc.
These aspects are valid both for the camera and host sides, and last but by far not least is the cable. Cable performance significantly influences reliability of the communication by attenuating high speed signals and delivering sufficient power to the camera.

## 2. Other devices

There are other devices sitting on the same bus and thus competing for the bandwidth and other shared resources. This can be resolved by placing each camera on individual xHCI controller.

### 3. Firmware

Camera firmware is not capable of streaming data at the maximum speed supported by the host, and thus reduces bandwidth by introducing latencies in committing transport packets.
PC hardware by itself is not imposing latencies.

## Software

**Software** contributes to **latencies** to handle request of supplying empty buffers to receive new camera data. These latencies in the PC software (like USB protocol stack and camera drivers, the buffer handling routines at API level, etc.) are effectively reducing the available bandwidth on one side.

### 1. Interfaces

Latency of servicing interrupts from hardware interface cards.
Camera interfaces, like FireWire, USB, GigE, are using respective host adapters. These adapters have different requirements for maximum latency to service interrupts, from which the most important and time critical is request to get descriptors for delivery of new data. The latency depends from the performance of the computer, operating system, software drivers, protocols and software stacks.
Camera interfaces like PCI Express or Thunderbolt are not using intermediate buffers or bus protocols. These two interfaces have substantially lesser software components and by design have much shorter latencies.

### 2. Applications

Application latencies – if application is in the loop of processing data from each frame and cannot complete this task within required timeslot. This can happen if algorithm for processing is not time deterministic by design. Another reason can be that underlying operating system cannot provide required amount of CPU resources within processing timeslot.

## Buffer handling

At the end of the day the software aspect converges to the buffer handling method. There are two typical implementations of buffer handling:

### 1. Allocation

Application allocates image buffers in computer memory, commits these buffers to camera queue, and waits when a buffer will get out of the queue with image data. Then application must process data and again commit “empty” buffer. The problem happens if application is “slow” and cannot process all incoming frames in time.
If application needs to acquire and process data in bursts, then application allocates as many buffers as needed for 2 (or more) consecutive bursts and commits all of them to a camera. This will ensure that all data are delivered and application has spare time between bursts to complete processing, and then commit again all processed buffers.

## 2. API

Camera API allocates a circular queue of buffers sequentially filling it with image data coming from a camera. When application wants to get new image data, it is either put on hold until first buffer is filled with image data, or returns immediately if there is already a buffer with image data. API keeps track of which image was last delivered to the application and which image is currently filling with image data. API detects “overrun” situation when application is too slow and whole circular queue is already filled with data.

# Summary

So, to wrap-up all previous considerations:

**1. For connection less protocols** (UDP in GigE) **it is vital to have large image buffer** inside the camera to be able to resend the packet or frame if it is lost.

**2.** For protocols which ensure packet resending on transaction layer like **USB 3.0, large image buffer inside the camera is not providing any benefit**. What is required is a FIFO to tolerate jitter on bus availability.

**3.** To build a reliable high speed application **both bandwidth and latencies** must be analyzed and troubleshoot.
**If they are not, then sooner or later camera will lose frames, regardless whether it has or has not a large image buffer inside.**
**Buffer handling method also must be aligned with the application requirements.**

XIMEA API, drivers and camera firmware are tuned to utilize all bandwidth from supported host controllers and have lowest possible latencies.