RESEARCH PAPER

USB LATENCY REQUIREMENTS AND
THE EFFECT OF VIDEO ADAPTER PCI
RETRY CONDITION ON MAINTAINING
USB STREAMING PIPELINES

by

Alberto Martinez

Dr. Sures Vadhva

California State University, Sacramento

Department of Electrical and Electronic Engineering

March 20, 1997

California State University, Sacramento

Abstract

USB LATENCY REQUIREMENTS AND
THE EFFECT OF VIDEO ADAPTER PCI
RETRY CONDITION ON
MAINTAINING USB STREAMING
PIPELINES

by

Alberto Martinez

Dr. Sures Vadhva

The Universal Serial Bus (USB) in a PC system has strong latency requirements to maintain the data stream for any active device. These requirements are especially true in the case of isochronous pipes, when large data transfers must be executed in a timely manner. In a typical PC architecture the USB controller is located in the PCI bus South Bridge; thus, USB traffic and latency are susceptible to PCI utilization and arbitration. This paper provides an analysis of the effect of the PCI retry mechanism on USB latency. This document also provides suggestions for an architectural redefinition that addresses this issue.

TABLE OF CONTENTS

Background 6
Problem description 6
Debug process 7
Issues workaround 10
Assumptions 11
USB Interface Analysis 11
    USB Software Solution 12
    USB Hardware Controller Solution 13
PCI Interface Analysis. 13
    USB Controller memory Access 14
    PCI Retry Mechanism 15
    Determining Buffer Requirements 16
    Driver and Graphic Pipeline Control 16
About a balanced design 18
Further work 18

LIST OF FIGURES

Number *Page*
Figure 1: USB-Enhanced System Architecture 7
Figure 2: Placement of Isochronous Transactions in USB Schedule Frames 9
Figure 3: Detail of Isochronous Packet Overlapping into Start of Next Frame 9
Figure 4 Producer-Consumer Model and Water Marks Usage 17

ACKNOWLEDGMENTS

The authors wish to acknowledge the contribution by the following individuals and institutions to the preparation of this document:

John Trelford (Intel Corporation, Intel Platform Support Labs)
Bill Wager (Intel Corporation, Intel Platform Support Labs)

Chapter 1

CASE STUDY: USB VIDEO CAMERA AND PCI GRAPHICS ACCELERATORS

Background

To provide the required frame for this document, we have chosen to first describe the initial problem that started the investigation of USB latency in PCI. This chapter will describe the facts of the investigation without making any assumptions about the cause of, or possible resolution to the problem.

Problem description

A USB camera was installed in a Pentium® processor 82430HX/PIIX3 based system (see figure No. 1). The camera was used as a video input device in a "Video Telephone" configuration, which was bundled as a part of a PC system, and included a PCI Video graphic accelerator card. The system displayed the video input from the camera in a small window on the screen. The application software included a "video mute" function that blocked the camera image from been displayed. To accomplish this function, a bitmap of a closed curtain was superimposed over the actual video display window. A possible application of this feature is to block the video feed for privacy, allowing voice only communication.

It was while executing the video mute function that the telephone application "stopped responding" to user inputs. This anomaly is usually called application "Hang-up". Notice that in the majority of occasions the actual OS (Windows* 95 Intel* USB Stack Supplement) was still active allowing launch and usage of different applications. To recover video telephone functionality, the system had to be re-initialized. A soft reset was sufficient to recover.

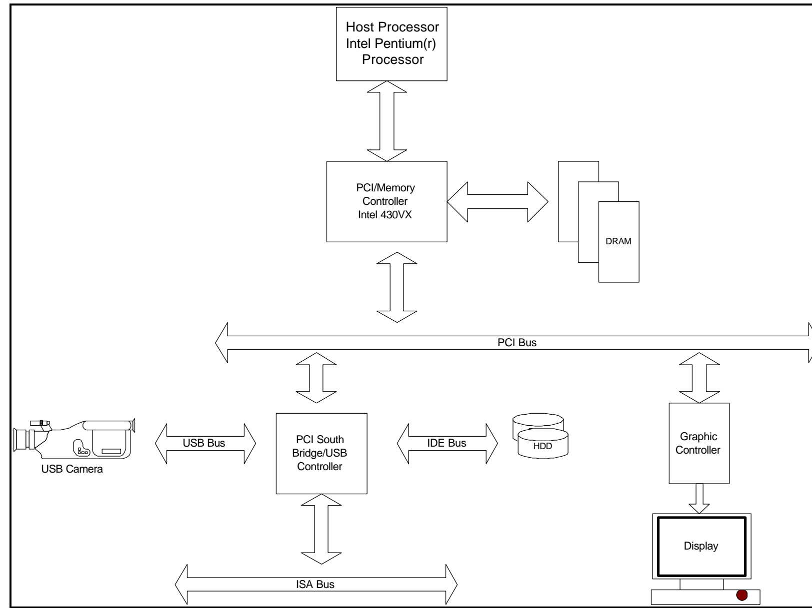

Figure 1: USB-Enhanced System Architecture

Debug process

A debug process followed; our intention was to determine the root cause of the problem. Our major concern at the time was to rule out a hardware malfunction. Given that the USB camera uses two isochronous pipes to provide the actual video stream, our first task was to ensure all master latency timers (MLT) in bus master devices where configured as recommended by performance analysis[1] on USB functionality.

The performance analysis indicated master latency should be set for a maximum of 1 ms for any PCI Bus Master and 2 ms minimum for the PIIX3 Function 2 (USB controller), to allow isochronous transactions under a defined maximum load. These time parameters translate to MLT values of 20h for all Bus Master and 40h for the USB controller.

Changing the MLT values to these recommended setting did not correct the issue; consequently, it was decided to used a more intrusive approach to identify the nature of the failure. A USB protocol checker (designed by Intel/IPSL) was connected to snoop the USB cable during the failure. Proceeding accordingly, it was found that during the failure

a "babble" condition was present at the end of an isochronous package. This condition caused the USB controller to shut down the active port, which eventually was reflected as an application failure.

The next step was to understand the nature of the babble condition. For this purpose, a Tektronix® DAS 9200 was used to probe the PCI Bus by means of a signal interface on the Intel/IPSL PCI Protocol Checker. Concurrently, the USB bus was monitored by means of a Intel/IPSL USB Monitor. After the analysis was completed, it was found that the PIIX3 was requesting the PCI bus on behalf of the USB Controller. The intention of the controller was to upload 32 Bytes of data to memory. This data was a data block from the current USB isochronous packet. Simultaneously, (but starting when the bitmap displayed on the video telephone window), the graphic accelerator was retrying command transactions intended to the video controller pipeline. The PCI arbiter was repeatedly granting the bus to the north bridge (82430HX) but not to the USB controller, preventing the USB controller from reading the next transfer descriptor from memory as well as blocking data package transfers to memory. The retry process usually exceeded 2 ms, eventually causing two independent failure scenarios:

1.- USB FIFO overrun. Data for the next video frame is dumped (lost), reducing the total frame per second count. This loss eventually degrades the video camera display performance and ultimately the user experience.

2.- Port "babble" condition. By delaying the IN token from the next transfer a port "Babble" condition occurs, causing the USB port to be automatically disabled, resulting in an application error. FIFO overrun is undesirable; however, port babble causes a system failure, making the anomaly critical.

Figure 2 below shows how isochronous transfers fit into a USB schedule. Note they appear first in a 1ms USB frame. Figure 3 below shows in greater detail the structure of a single frame, indicating the region of frame (***n+1***) over which the isochronous transfer from frame ***n*** overlaps. Expanding the time used for the isochronous transfer beyond the ***n***th frame's boundary violates the frame structure, resulting in port babble.

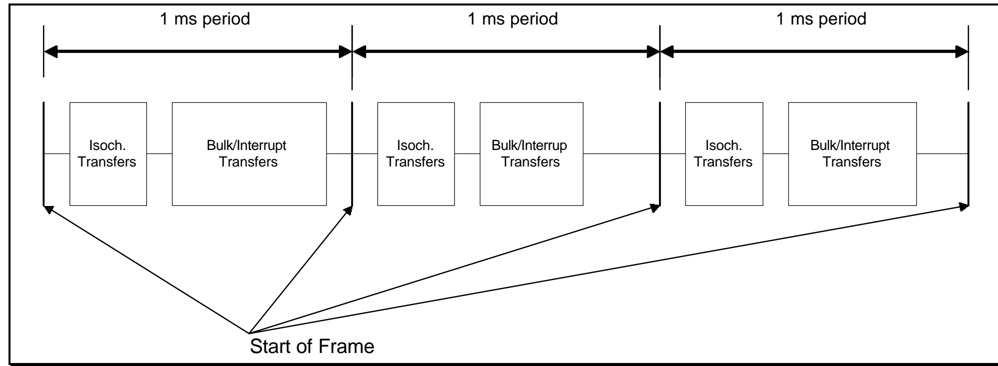

Figure 2: Placement of Isochronous Transactions in USB Schedule Frames

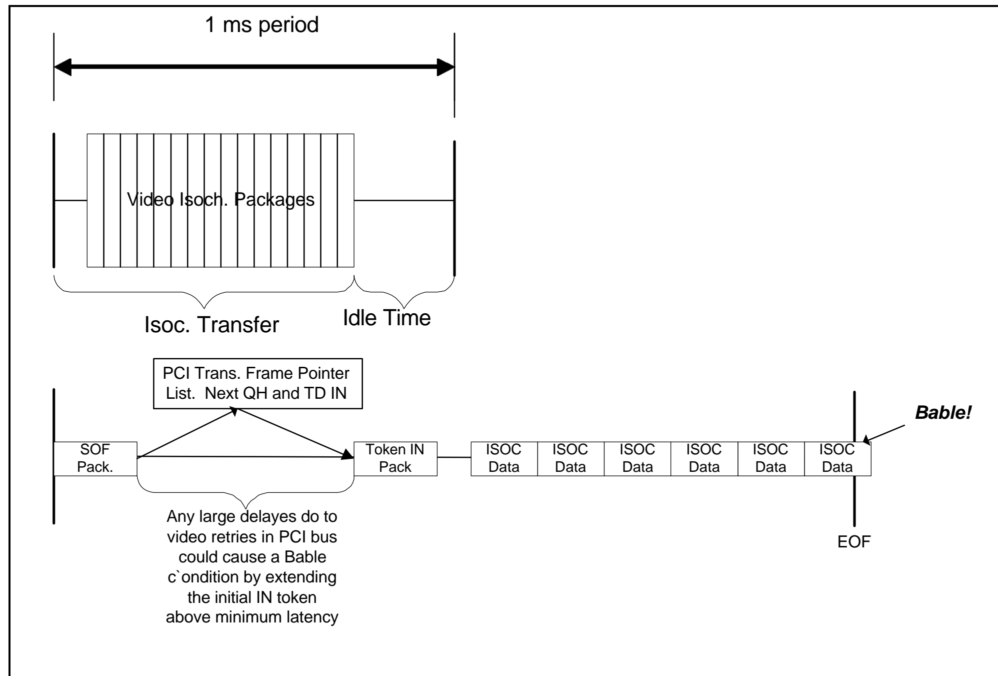

Figure 3: Detail of Isochronous Packet Overlapping into Start of Next Frame

These issues were reproduced using three different hardware-accelerated video adapters. These tests corroborated the common nature of the problem.

Issues workaround

After consulting with the graphic card manufacturers it was found that to obtain the maximum possible video display rate, as measured by a popular video benchmark software, the video driver continuously writes commands to the Graphic Accelerator command pipeline. Eventually the command pipeline is full, and no commands can be received:. the graphic controller retries additional incoming commands. At this point, if a USB-to-PCI transaction is initiated, the host controller arbiter will not grant the PCI bus to the USB controller until all North bridge internal posting buffers are flushed. Given that the video commands are still in these posting buffers, effectively blocking USB controller access to memory, the USB transaction will eventually fail to complete within the required time frame.

One possible workaround is to disable Windows 95 hardware acceleration capabilities. This fix is made in the Windows95 *System Properties* menu, under the *Performance* tab, using the *Graphics* button. It is possible to select a setting in between ***Full Hardware Acceleration*** *and* ***No Hardware Acceleration*** that will suffice as a work around this issue. However, using ***No Hardware Acceleration*** will visibly impact the video adapter's performance.

A second possible workaround is to enable the video drivers to throttle the commands send to the graphic controller. All the video adapters tested allowed some level of control by adding an initialization entry to the ***system.ini*** file. When throttling was enabled, the video drivers avoid overrunning the video adapter command pipeline. Consequently, this throttling reduces arbitration turnaround, allowing USB traffic to properly access memory. This option will also impact the video performance; however, the performance reduction is not clearly appreciated unless a benchmark program is used.

Chapter 2

ARCHITECTURAL ANALYSIS

Assumptions

For the purpose of this analysis, we will concentrate on the second failure mechanism, *babble condition*. The sequence of events that result in the application error after a port babble also cause frame drop condition. Therefore, the architectural analysis and possible resolution of the port babble will also largely apply to the frame drop condition. It will be indicated in the text below when this underlying assumption is no longer valid.

The system failure is based on the system's sensitivity to graphic accelerator responsiveness (or lack thereof) to pending device driver requests.

The PCI Specification[2] does not limit the number of retries a PCI target may execute, and does not indicate any maximum latency before a target must complete a retry condition. Thus, the retry behavior of the graphic accelerator in this case is not in conflict with the current PCI specification.

To try to understand the true nature of the issue, we will break down the failure mechanism into its minimal components, starting with the USB physical layer, then moving to the graphic accelerator driver. It is hoped that with this analysis, an economical and viable solution could be identified.

USB Interface Analysis.

At the instant of a failure, the USB device is transferring an isochronous data package. This data package, over 300 bytes, is large but still within the parameters indicated by the USB Specification[3]. It is true that if the data package were smaller, the probability of occurrence of the failure would be proportionally reduced. However, the same failure

mechanism could be reproduced with a large number of smaller packages. Thus, it does not look like constraining the package size and/or the total allocable USB bandwidth would remove the failure mechanism. Practically speaking, any architectural redefinition in this area should encompass a larger bandwidth availability to allow more appealing results in video conferencing-like applications.

If the above assumptions are true, then different approaches should be considered:

1.- Is it possible to recover from the babble condition in a way that will not result in a port shut down and application error?

2.- Is it possible to introduce a more effective fail safe mechanism to prevent the port babble condition?

USB Software Solution

The underlying assumption behind a port shut down in USB is a device malfunctioning. Data transmission after the EOF is assumed to be caused by a defective device, therefore unrecoverable in nature, except by unplugging the device from the port. But in this case, the device is functioning properly; a system latency issue results in the babble condition. Here the failure goes beyond the original fail tolerance introduced in the USB specification.

Without introducing changes to the USB specification, there is a limited solution at the USB peripheral driver and USB Stack interface. The Universal Host Controller Driver (UHCD) to device driver interface could be modified to allow a ***fly-by*** re-initialization of the port and device. The application would then pause while the port and device re-initialize, and the system would not hang. Permanent port shutdown would not be required. This scenario assumes the underlying PCI retry mechanism on the graphics accelerator is a statistical anomaly, occurring infrequently.

If the babble condition repeats at a predetermine rate, a final shut-down could be executed, in this case under the assumption a device or critical system failure has occurred.

USB Hardware Controller Solution

Answering the second question above is possibly more complex, given that a solution requires a larger modification to the USB Specification and UHCI interface. Such a proposal is probably material for a more detailed analysis, which is beyond the scope of this document. Nevertheless, we will try to summarize a mechanism that should be sufficient to prevent latency-based babble conditions.

It is possible to introduce a timer/counter mechanism based on current location of execution in the frame and the timing of the reception of the transfer descriptor (TD) for the next IN/OUT token. The current TD contains the expected package length and using the value in the newly-defined timer, comparing it to the estimated execution time for the current packet, an estimated likelihood of a babble condition occurring could be calculated. If babble is likely, then the IN/OUT token would not be submitted, and the transaction would be treated as a time-out.

Note that the USB controller hardware requirements, gate count and the like, should be part of any feasibility study for this solution. Also note that this mechanism would not be able to resolve the less-critical frame drop condition. The frame drop condition has dependencies on the transmission of data already received from the USB controller and sent to memory. The timer and TD solution described above does not address the inter-package latencies typical of this failure.

PCI Interface Analysis.

It is in the PCI side of the system architecture where a more complete solution could be identified. At the same time it is on the PCI bus were more complex interactions take place, thus making any possible solution more involved not only technically, but also from the industry interactions required.

Trying to understand the PCI side of the issue, two questions could be posted:

1.- Why is the PCI North bridge not granting memory access to the USB controller?

2.- Why is the PCI graphics accelerator using the retry mechanism carelessly, negatively affecting PCI latencies?

USB Controller memory Access

The USB controller (see fig. 1) resides as function no. 2 inside the PCI South (compatibility) bridge. This location places the USB controller as a member of the PCI to ISA bridge family of devices, together with the proper ISA bridge and IDE controller inside the Intel PIIX3 device. There are a number of advantages to designing such a location for USB; however, this location in the current design forces all PCI arbitration to have the same architectural constraints as does ISA bridge arbitration. When the South bridge requests the bus on behalf of an ISA device, the PCI arbiter must ensure that all pending PCI transactions are flushed (completed) before the PCI bus is granted to the requester. This behavior is required because of numerous limitations on the ISA bus, including:

1.- Absence of a retry mechanism for ISA masters.

2.- Absence of deterministic latencies for ISA masters.

3.- Low speed that translates into large PCI utilization.

The USB interface does not necessarily suffer from the same limitations. However, the USB and ISA controllers share the same PCI device arbitration, so the PCI arbiter has no means to differentiate between a request on behalf of an ISA master or the USB controller. Therefore, when the USB controller arbitrates for memory access and a retry from the graphic accelerator has occurred, the PCI bus will not be granted to the USB controller until the pending (retried) transaction is completed.

To remove this limitation, it is possible to redefine the architecture of the South bridge to accommodate the USB controller as a second PCI device. This change adds an additional South bridge GNT#/REQ# signal pair to de-couple USB PCI arbitration from any possible ISA master in the system. Again, the details of such mechanism are not within the scope of this document. In summary, it appears that de-coupled arbitration for the USB controller should allow the initial USB request to result in USB access to the memory controller in the PCI north bridge.

This de-coupled arbitration would probably resolve the frame drop situation (USB controller doing memory writes and not reads); however, it is not sufficient to prevent the babble condition caused by a delayed IN token. Even when a grant is given to the USB controller to access memory space, PCI strong ordering rules could still prevent the transaction completion. To maintain event synchronization, PCI does not allow a memory write posted in the North bridge (command retried by the graphic accelerator) to be passed/crossed by a memory read command from PCI (USB controller trying to read the next TD from the list.) Below is an excerpt from the PCI specification, appendix C, where these rules are described:

"A read transaction must push ahead of it through the bridge any posted writes originating on the *same* side of the bridge and posted *before* the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the *opposite* side and were posted *before* the read command completes on the read-destination bus."

Therefore, a complete solution to grant the USB controller reasonably quick access to memory requires addressing the original indiscriminate retries by the graphic accelerator.

PCI Retry Mechanism

The PCI retry mechanism allows a PCI target to tell the master to initiate the transaction again later. A retry is usually caused by a device being internally busy or incapable of returning data requested, within the limits of the PCI transaction latency requirements. In the case of this analysis, the retry condition occurred as a result of a command buffer (FIFO) in the graphic accelerator being full. The graphic accelerator needs to process a number of previously queued commands before it is capable of accepting additional commands. Evidently, the number of consecutive retries that can occur will be a function of how fast the commands are retired from the graphics controller command pipeline, and the size of the FIFO . USB frames have a duration of 1 ms; any USB controller access to memory delayed by more than a significant fraction or larger of this value (as is caused by the graphic adapter retries) will disrupt USB functionality as described in previous pages.

The PCI specification does not restrict the number of retries nor the maximum time period during which the retry command is to be accepted by the target. However, it could be interpreted from the specification a relevant *spirit of the specification,* an underlying *"good neighbor policy"* that should be maintained by all PCI devices. What follows is a

possible solution for the graphic accelerator vendors to apply to reduce the impact on PCI latencies, and hence minimize or eliminate the USB babble condition.

Determining Buffer Requirements

Each PCI device that interfaces to the bus needs buffering to match the rate the device produces or consumes data. In the case of the graphic accelerator and this analysis, buffering implies the *rate* commands are send to the graphics pipeline. Therefore, buffer capabilities are a function of the specific graphic accelerator efficiency in processing the commands. A detailed analysis could be performed to determine the specific needs of each one of the adapters evaluated. A tuned buffer will prevent the misuse of PCI resources and eliminate the latency limitation of USB transfers. However, the usefulness of any recommendations from any buffer analysis will be limited by the assumptions made regarding processor performance and maximum bandwidth availability. As indicated in this document, the graphic accelerator driver is overwhelming the accelerator command buffer;, this *performance miss-match* is possible given the high performance processor used during the test and the capabilities in the Host to PCI bridge. It is possible that previous generations of a system configuration would not overrun the command FIFO, given a slower performance rating. In short, a buffer estimation made based in current technology may fail in future system configurations. To avoid this problem a second layer of control between the driver and the graphic accelerator could be implemented.

Driver and Graphic Pipeline Control

As described earlier in this document, some of the graphic adapters evaluated allowed a configuration that prevented the USB failure. The basic functionality that was enabled in such a configuration was a second layer of communication between the driver (command producer) and the graphic accelerator (command consumer.) Under the Producer-Consumer operational model reflected it this case, it is possible to establish a finer control between the driver interface and the command pipeline in the graphic accelerator. The driver and command pipeline could communicate between each other via a flag and a status element.

The graphic accelerator could indicate a “ready” status by setting a flag when the command pipeline is ready to receive further commands. Likewise, just before the command

pipeline has reached an unacceptable level, the same flag could be reset. This *ready flag* handshake model could be further enhanced by implementing dynamic flow control in the command pipeline. The flow control could be based on a pair high/low water mark set. When the command pipeline reaches below a preset "lower mark" the graphic accelerator would use the bus mastering capabilities to set the flag in memory. To throttle down the flow of commands, when command pipeline reached above the "high mark", the graphic accelerator would access the read flag again resetting it to not-ready status.

In a well-tuned PCI peripheral, the use of the dynamic control described above would be as a second-level support for correctly-defined buffer capabilities of the command pipeline. Figure 4 depicts the Producer-Consumer model and the use of water marks for buffer control.

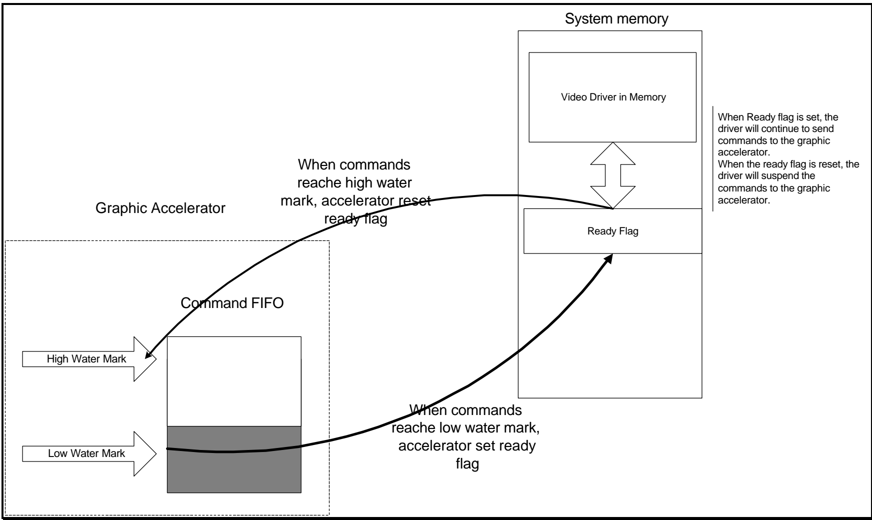

Figure 4 Producer-Consumer Model and Water Marks Usage

Chapter 3

CONCLUSIONS AND CONSIDERATIONS

About a balanced design

PCI latencies and PCI peripherals' behavior are definitely factors to consider in a USB-capable system. When USB peripherals are used that support large isochronous packages, heavy USB workloads are present. The system designer must consider the use of *well behaved* PCI devices, especially PCI graphic accelerators, which must not overuse the PCI retry mechanism, affecting USB pipes.

There are a number of possible architectural changes that could be applied to improve the current chip-set design and system specifications. These changes alone could greatly improve the robustness of the system, but will not be sufficient to protect against an unbalanced PCI device. It is even possible to define more drastic steps to enforce a "good neighbor" policy; however, these steps increase restrictions, possibly curtailing designers' freedom when choosing more cost-effective peripheral implementations.

Finally, it is for us evident that the best solution still resides on the hands of the peripheral designers. A correctly balanced design should still be capable of performing while allowing other devices and busses to properly operated in the system.

Further work

At the time the authors are completing this document new and exiting technologies are emerging. The Accelerated Graphic Port or AGP, is becoming the graphic accelerator standard. The AGP bus already de-couples the graphic accelerator from the PCI bus and adds specific functionality to support pipeline mode of operation. At the present time the authors have not engaged in a study to determine the susceptibility or lack thereof of this

new architecture to USB latencies. Such investigation is certainly material of much work, like the present.

Also, the IEEE 1394-1995 standard is furthering streaming capabilities in the PC architecture. As in the case of USB, 1394 is susceptible to nuances of PCI peripherals' operation. While finishing the draft of this document, we were pleased to observe that techniques similar to this document recommendations are been used in the definition of the 1394 link interface in future generations of south bridge designs.

BIBLIOGRAPHY

[1] Smith, Mike. *"10 ways to improve USB performance"*, Training presentation, Folsom, CA: 1996.

[2] PCI Special Interest Group. PCI Specification Rev. 2.1, Portland, OR. 1995

[3] USB Special Interest Group. USB Specification Rev. 1.0, Portland, OR. 1996