Time Synchronization Accuracy With PCI And USB Devices
# Table of Contents

Time Synchronization Accuracy With PCI And USB Devices ................................................. 1  

PCI Access Times ............................................................................................................. 2  

DMA Approach By 3rd Party Manufacturers ................................................................. 4  

Meinberg's Approach ....................................................................................................... 4  

Getting Accurate time Stamps At A Very High Rate ..................................................... 4
Time Synchronization Accuracy With PCI And USB Devices

The accuracy of time you can yield with a PCI card as reference time source depends on many preconditions, e.g. the operating system type (Windows, Linux, ...) and particular version of that OS, the CPU type (good TSC support, or not), the chipset on the mainboard, and the quality of the oscillator on the mainboard which determines the stability of the system time. In addition there are hardware limitations introduced by the concept of the system busses, e.g. PCI or USB.

Linux, FreeBSD, etc. provide a good programming interface to read the current system time accurately, and to apply adjustments to the system time properly. Windows is much worse than this.

There are CPU types providing reliable TSC counters which can be used to measure and thus at least partially compensate some latencies.

The frequency of the oscillator on the particular mainboard has a mean deviation from its nominal frequency, and in addition the frequency varies more or less with changes of the ambient temperature inside or outside the PC housing. This can be due to changes in the CPU load causing more or less power consumption and associated heat inside the PC housing, an air condition kicking in and out in a server room, or just to daily temperature variations between day and night.

Since the time drift depends on the frequency variations, the control loop of a good time synchronization software tries to determine and compensate the frequency of the oscillator, and steers the oscillator so that a time offset doesn't even develop.
The picture above shows a graph generated from a loopstats file by ntpd 4.2.8p7 running on a Linux kernel 3.16, using a Meinberg GPS180PEX PCI card as time reference. Data has been recorded over about 4 days. The mean frequency offset is about -9.2 ppm and you can clearly see the frequency variations due to the daily changes of the ambient temperature, which ntpd tries to compensate.

**PCI Access Times**

Whenever a the CPU accesses some peripheral the access time depends on how the peripheral is connected to the CPU. For example, reading the CPU's internal TSC counter register just takes a few nanoseconds, but reading data from a device in the chipset is already in the range 1.5 microseconds, or 1500 ns. Reading some data from a PCI Express card is even worse:

- The read command including register address needs to be serialized by the chipset on the mainboard
- The serial message needs to be passed on to the card, traversing some bridges
- The card needs to convert this back to parallel data and access the on-board register
- The data read from the register needs to be serialized
- The serial message needs to be passed back to the mainboard, again traversing some bridges
- On the mainboard this needs to be converted back to parallel data
- Parallel data is finally read by the CPU

If a whole stream of data is to be transferred then some of the steps can be pipelined, i.e. while one message is sent the next message can already be prepared/serialized, so the overhead of the
different steps above can often be neglected. However, if an application just reads a time stamp from the PCI card at a random point in time then each read access requires the full overhead as explained above. In addition, each of the message passing actions can be delayed if a part of the bus is busy. This is unnoticeable for the driver, and thus a big disadvantage for timing applications in general.

A 64 bit time stamp from the card is usually read as 2 x 32 bit data. How long this takes depends also on the chipset and the architecture of the mainboard and can be in the range of 4 to 8 microseconds, or even worse. For example we've seen a mainboard where it takes 4 microseconds to read the time stamp from a card in the first slot, and 8 microseconds to read the same type of time stamp from the same card in a different slot.

The PCI subsystem is basically similar to a computer network, where the CPU and every PCI slot are individual network nodes. Each PCI bridge on the mainboard between the CPU and a particular PCI slot is similar to a network router or switch. This means that data access to a device can be delayed by a PCI bridge if a different CPU core accesses a different device via partially the same route between bridges, or if there's an ongoing DMA transfer e.g. from a hard disk controller. This is basically the same as if you do time synchronization across the network, where some packets may be queued/delayed, and others are not.

![Diagram of PCI subsystem](image)

In the graph above, if there's an ongoing DMA transfer from the disk controller to memory (red line) then obviously the first part of the PCI bus close to the CPU is busy, and if an application running on the CPU is tries to read some data from the GPS card (green line) then access is delayed until the DMA transfer has finished.

This causes a latency which can't be easily detected by the application, and when it comes to
accuracy requirements in the microsecond or sub-microsecond range things get really tricky.

**DMA Approach By 3rd Party Manufacturers**

There are folks out there telling you the best solution is to let a PCI card itself write the current time continuously to a specific memory location, using DMA mechanisms, so applications just have to read the time stamp from that memory location.

However, if the card automatically updates some memory location once every few microseconds then there is no way to find out if the DMA data packet sent from the card to the memory location has been delayed, or not. If it is delayed then the memory location is updated later than expected, and an application reading the time from that memory location retrieves a previous time stamp, i.e. an earlier time, which is wrong.

This is similar to receiving broadcast packets from an NTP server, where the receiver is unable to find out if and how long the packet it has just received has already been travelling on the network.

Interesting questions would be what happens if configuration data is read from or written to the card by an application while the card is sending time stamps via DMA, or if concurrent access occurs, e.g. an application tries to read the memory location while it is just updated.

So at a closer look at the DMA approach has also quite a few limitations.

**Meinberg's Approach**

The most important thing for synchronization of the PC's system time is to **read the time from a card and the current operating system time as close as possible** after each other.

Meinberg's approach to do this under Linux or *BSD is to let the kernel driver read both time stamps, plus the TSC count before and after this has been completed. The time synchronization software can then check **how long it really took** to read these time stamps. When it took longer than “usually” on a specific machine there is a good chance that the program execution has been interrupted, the PCI access has been delayed due to bus arbitration, or something similar, and so the time synchronization software can just discard these time stamp pair and get a pair of new ones.

This is basically what also ntpd does when it sends a query to an NTP server across the network, then checks the overall propagation delay after the reply from the server has been received, and compensates the latencies as good as possible.

**Getting Accurate time Stamps At A Very High Rate**

Some folks out there want to get time stamps from a PCI card very quickly, at a very high rate. The best approach we have found for this scenario is to write a multi-threaded application where one thread reads and updates a time stamp / TSC pair in periodic intervals, and other threads just use the
current TSC (which can be read very fast) and the last time stamp / TSC pair to extrapolate the current time.

The `mbgxhrtime` example program which ships with the driver packages for Linux, FreeBSD, etc. implements this feature.

See also the whitepaper at https://www.meinbergglobal.com/english/info/#whitepaper

Specifically:
Meinberg Driver and API Concepts

— Martin Burnicki 2017-05-29 15:15