Time Synchronization Accuracy with PCI and USB Devices
Table of Contents

Time Synchronization Accuracy with PCI and USB Devices ...................................................... 1
  PCI Access Times .................................................................................................................. 2
  Access Times Depend on the PCI Slot and Chipset ............................................................. 4
  USB Access Times .............................................................................................................. 5
  PCI DMA Approach by 3rd Party Manufacturers ................................................................. 5
  Meinberg's Approach .......................................................................................................... 6
  Getting Accurate Time Stamps at a Very High Rate .............................................................. 6
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 CPU accesses some peripheral device, the access time depends on how the device is connected to the CPU. For example, reading a counter value from the CPU's internal TSC counter register just takes a few nanoseconds, but reading data from a counter register in the chipset may already take about 1.5 microseconds, or 1500 ns. Reading some data from a register on 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's chipset 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 an entire stream of data is to be transferred, some of the steps can be pipelined, i.e. when 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, 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 not noticeable to the driver and is therefore a major 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, see below.

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 forwarded immediately, without delay.

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

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

**Access Times Depend on the PCI Slot and Chipset**

The time required to read a 64 bit timestamp from a PCI card depends strongly on the characteristics of the PCI bus on a particular mainboard. This includes the type of chip set installed on a mainboard, and the number of PCI bridges between the CPU and the slot into which the PCI card has been installed. Thus, access times may even vary between different slots on the same mainboard.

You can use the `mbgfasttstamp` utility which is part of the `mbgtools' command line tools to determine the latency.

For example, the following command reads a number of time stamps in a loop as fast as possible, and then displays the results and time differences:

```
mbgfasttstamp -n 40 -b
```

With a particular type of PCI card in a specific Linux workstation you may get these results:

<table>
<thead>
<tr>
<th>HR time</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-01-07 08:36:57.1937446 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937478 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937511 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937543 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937577 (+3.4 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937609 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937642 (+3.3 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937674 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:36:57.1937706 (+3.2 us), latency: 0.2 us</td>
<td></td>
</tr>
</tbody>
</table>

The latency numbers in the output indicate how long it takes from entering the API function until the PCI card is actually accessed, and the time differences indicate how long it totally took to read a 64 bit timestamp.

This means it takes ~3.3 us here to read a 64 bit timestamp from a card, using the Meinberg API calls which even use locking to make sure the results are not messed up in a multitasking environment.

However, if you run the same command on the same computer, against the same PCI card only installed in a different PCI slot, you may get different results, e.g.:

<table>
<thead>
<tr>
<th>HR time</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-01-07 08:47:33.9481913 (+7.0 us), latency: 0.3 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:47:33.9481971 (+5.9 us), latency: 0.3 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:47:33.9482032 (+6.0 us), latency: 0.2 us</td>
<td></td>
</tr>
<tr>
<td>2019-01-07 08:47:33.9482083 (+5.1 us), latency: 0.2 us</td>
<td></td>
</tr>
</tbody>
</table>
The access times are nearly twice as long here, just because the card is installed in a different PCI slot.

There's an 17.2 us outlier here because program execution has been interrupted by a hardware IRQ, or was delayed because the PCI bus was busy.

Please note there are also systems where the mean access time is about 12 us instead of about 3.3 us.

These limitations are due to the PCI (Express) bus, and they also apply to PCI cards from other manufacturers, even if they don't mention this in their data sheets.

**USB Access Times**

USB access basically works the same way as described above for PCI Express. Instead of PCI bridges there are USB hubs, and parts of the USB connection can also be shared between several devices.

In addition, a driver can't access a USB device directly. It has to do this via the USB subsystem provided by the operating system, which is responsible for writes packets to or reading packets from a device on behalf of a driver or application.

The transfer rate and packet delay depend on the USB version, but the access time is generally much worse than via PCI Express, so it's much harder to read a precise time stamp via USB than via the PCI Express bus.

**PCI 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 has been delayed, the memory location is updated later than expected, and an application reading the time from that memory location retrieves a time stamp that has been updated previously, 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 traveling 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 in case concurrent access occurs, e.g. an application tries to read the memory location while it is just being 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 “usual” 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. 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 applications need 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 read the current TSC (which can be done very fast) and use the last time stamp / TSC pair to extrapolate the current time.

The `mbgxhrtme` 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](https://www.meinbergglobal.com/english/info/#whitepaper)

Specifically:

— Martin Burnicki 2020-04-17