GigE Vision - CPU Load and Latency

GigE Vision is becoming a more powerful and reliable interface in the machine vision market. In addition to IEEE 1394a/b (FireWire) and Camera Link, GigE Vision is now found in many applications, for example, in traffic control, PCB inspection, and medical care. The GigE Vision standard makes a persuasive argument especially regarding bandwidth, transmission mechanism, reliability, and cable length. GigE Vision is based on the topology of well-established Gigabit Ethernet technology. Gigabit Ethernet has the advantage of wide penetration in both the industrial and consumer markets and serves as the basis for an easy-to-use, low-cost interface for a wide variety of applications.

GigE Vision CPU Load

In the machine vision market, there are still some open questions regarding GigE Vision. Whenever Gigabit Ethernet is discussed in connection with machine vision, CPU usage and latency times in the PC are among the most interesting and controversial topics. Different players in the machine vision industry still must be convinced that the CPU load generated by the use of a GigE transmission mechanism is within acceptable limits. The main difference between GigE and FireWire or Camera Link is that with GigE there is no CPU-independent incoming data management. This means that every incoming packet must be handled when arriving in the PC and copied afterwards. This process always requires CPU involvement. To keep the CPU load as low as possible, Basler offers two different GigE Vision drivers: a filter driver and a performance driver.

The Basler Filter Driver

The Basler filter driver quickly separates incoming packets and transfers them directly to the application. The filter driver can be used with all network interface cards available on the market. The CPU load associated with the filter driver is generally attributable to the fact that packets must normally be copied two times.

The Basler Performance Driver

The Basler performance driver also separates incoming packets and transfers them directly to the application. The CPU load associated with the performance driver is generally attributable to the need to make at least one copy of the incoming data. The main advantage of the performance driver is that it lowers the CPU load needed to service the network traffic between the PC and the camera(s) even more significantly than the filter driver.

The Basler performance driver is a hardware specific GigE Vision network driver and is compatible with network interface cards that use specific Intel chipsets.

Filter and Performance Driver CPU Load Comparison

Due to the different architectures of the drivers and to different hardware platforms, the CPU load of the filter and the performance drivers varies. There are other parameters that also contribute to this effect including: network vendors, network topology, packet size of the data transfer stream, bandwidth, and the network class (avoids packet resends).

As described above, the filter driver and the performance driver use the CPU with different intensities. The graphs in Figures 1 and 2 reflect typical results for a medium performance PC and a bandwidth situation comparable to IEEE 1394b. The PC is equipped with a Pentium dual core 2.8 GHz processor, an Intel Pro 1000 GT network card, and has the Basler pylon SDK installed. The measurements were performed with a Basler piA640-210gm. Network workloads were between 15 and 61 Mbytes/s and the packet size was either 500 or 4000 bytes.

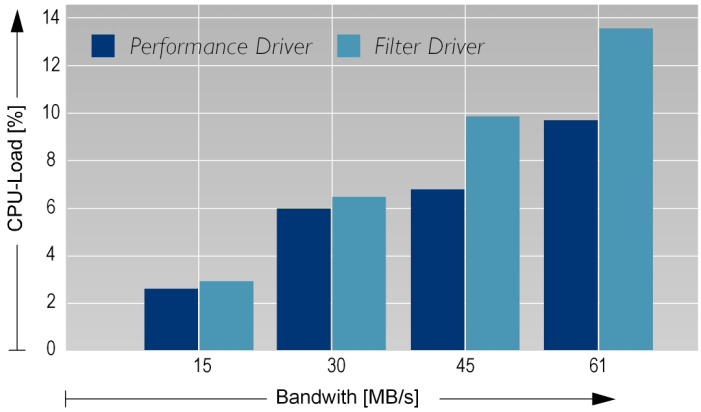

Figure 1: CPU load with a packet size of 500 Byte

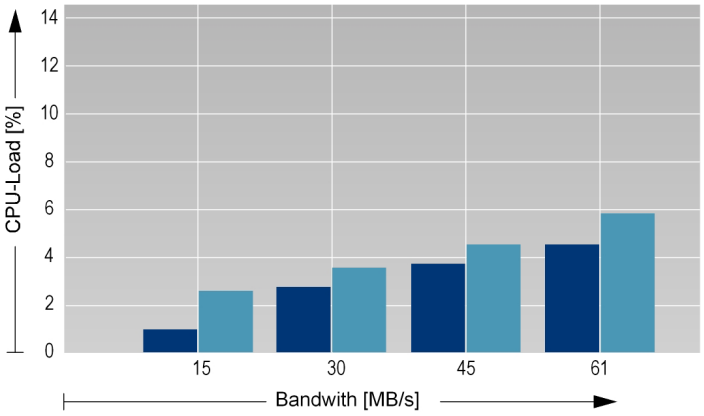

Figure 2: CPU load with a packet size of 4000 Byte

As you can see, the performance driver has a significantly lower CPU load than the filter driver. In addition, you can see that a larger packet size has a positive impact on the CPU load. A data stream based on small packets includes more overhead/protocol data than a data stream with large packets. A typical CPU load imposed by a GigE based application is 5 %

For high performance applications, we suggest using the pylon performance driver and an Intel network card that supports jumbo frames (i.e., a 6 Kbyte packet size).

To make the evaluation of measurement data more transparent, Basler is providing a Java script that can be used to determine the current CPU load on a machine vision system's host PC. This script uses the same Windows operating system functions as the Windows Task Manager, and the results are comparable.


```
// CPU Load.
//
// This script samples the systemwide CPU load at regular intervals.
//
// Run by using "cscript /nologo cpuload.js" within a console.
// Stop by pressing Ctrl-C.
//
// Note: Because the Win32_PerfFormattedData_PerfOS_Processor formatted
// data class is not available on earlier Windows versions,
// this script only runs on Windows XP and above.

var wbemFlagReturnImmediately = 0x10;
var wbemFlagForwardOnly = 0x20;

var strComputer = ".";
var objWMIService = GetObject("winmgmts:\\\\" + strComputer + "\\root
\\CIMV2");

var total = objWMIService.Get("Win32_PerfFormattedData_PerfOS_
Processor.Name='_Total'")

for (;;)
{
          total.Refresh_();
          WScript.Echo("\rProcessor Time:
" + total.PercentProcessorTime + "%  ");
          WScript.Sleep( 1000 );
}
```

Java script to determine CPU load

For the workload evaluation process, a sample program from the pylon SDK called 'AcquireContinuous.cpp' is used. This application continuously acquires images and stores them in the application's buffer. This program does not perform any additional user interface activities such as displaying the image or analyzing the image content.

```
[…]

        // Grab c_ImagesToGrab times
        for ( int n = 0; n < c_ImagesToGrab; n++)
        {
            // Wait for the grabbed image with a timeout of 3 seconds
            if ( StreamGrabber.GetWaitObject().Wait( 3000 ))
            {
                // Get the grab result from the grabber's result queue
                GrabResult Result;
                StreamGrabber.RetrieveResult( Result );

                if ( Grabbed == Result.Status() )
                {
                   // Grabbing was successful, process image
                    cout << „Image #" << n << „ acquired!" << endl;
                    cout << „Size: „ << Result.GetSizeX() << „ x „
                         << Result.GetSizeY() << endl;

                    // Get the pointer to the image buffer
                    const uint8_t *pImageBuffer = (uint8_t *) Result.Buffer();
                    cout << „Gray value of first pixel:
                                „ << (uint32_t) pImageBuffer[0]
                         << endl << endl;

                    // Reuse the buffer for grabbing the next image
                    if ( n < c_ImagesToGrab - c_nBuffers )
                        StreamGrabber.QueueBuffer( Result.Handle(), NULL );
                }
                else if ( Failed == Result.Status() )
                {
                    // Error Handling
                    cerr << „No image acquired!" << endl;
                    cerr << „Error code : 0x" << hex
                         << Result.GetErrorCode() << endl;
                    cerr <<"Error description : „
                         << Result.GetErrorDescription() << endl;

                    // Reuse the buffer for grabbing the next image
                    if ( n < c_ImagesToGrab - c_nBuffers )
                        StreamGrabber.QueueBuffer( Result.Handle(), NULL );
                }
            }
            else {
                // Timeout
                cerr << „Timeout occurred!" << endl;

                // Get the pending buffer back (It is not allowed to deregister
                // buffers when they are still queued)
                StreamGrabber.CancelGrab();

            // Get all buffers back
            for ( GrabResult r; StreamGrabber.RetrieveResult( r ););

                // Cancel loop
                break;
            }
        }

        // Stop acquisition
        Camera.AcquisitionStop.Execute();

           […]
```

Part of the 'AcquireContinuous.cpp' program used to create a CPU load

GigE Vision Latency Time

The latency time of the GigE Vision interface is frequently discussed in the machine vision market. There are many parameters that have an impact on the latency time of a machine vision application.

In machine vision applications, there are usually two scenarios. In the first scenario, image acquisition is started by a program (a software trigger). In the second scenario, an external electrical trigger (a hardware trigger) is used.

With the software trigger scenario, the application creates an exposure start command using the Basler pylon API. The pylon API passes the command through the IP stack. Next, the command is transmitted over the physical network layer to the camera. When the command is received, the camera starts exposure of the sensor.

With the hardware trigger scenario, passage of a command through the IP stack and over the network layer is not necessary. The electrical signal is applied directly to the camera and immediately starts sensor exposure.

The receipt of the data stream from the camera by the host PC is identical in both scenarios. The camera transmits the streaming data across the network layer to the GigE vision driver (performance or filter driver) via the Pylon API to the image buffer of the application.

Figure 3 shows the logical path of the software trigger (green) from the application to the camera and the data stream path from the camera to the application (blue).

To determine the latency times of scenario one and two, different time stamps are defined:

- Software trigger start
- Exposure start command received by the camera
- Hardware trigger start
- Start of exposure
- End of exposure
- Images received by the camera

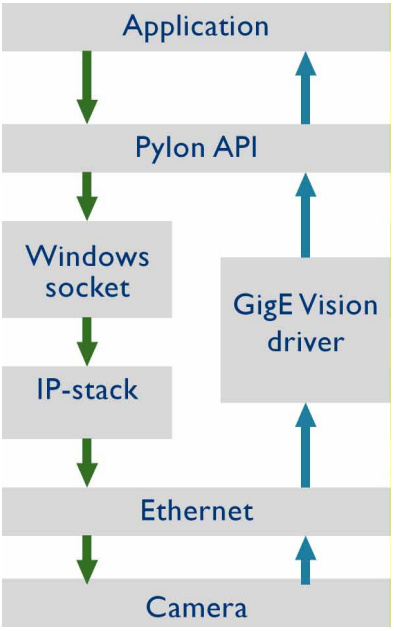

Figure 3: Trigger logical path

Figures 4 and 5 show timelines for the trigger signals and their jitter. The measurements were made for GigE and IEEE 1394 to be able to compare both technologies. The measurements were performed with Basler scA640-70fm/gm cameras.

As Figure 4 shows, the timeline of the software trigger is very short in relation to the total timeline, but the variation is significant. This variation is caused by the IP stack of the operating system. There is no significant influence from the GigE Vision driver. The latency time of GigE and FireWire are very similar and are both in the range of 30%.

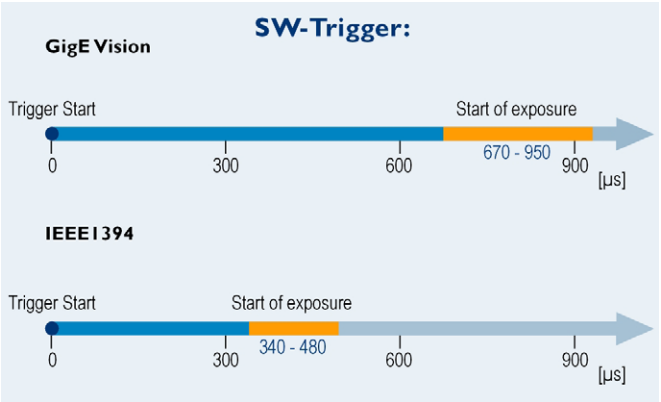

Figure 4: Latency time of software trigger for GigE vs. IEEE 1394

The timeline from the hardware trigger to the start of exposure depends on the camera model and is in the area of 50µs (please refer to the user's manual for your specific camera model).

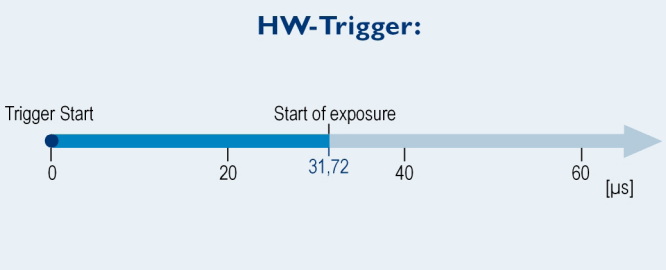

Figure 5: Exposure start delay for the scA 640-70

The timeline for the data stream from the camera to the image buffer of the application depends on several different parameters including: performance of the host PC, image size, packet size, bandwidth of the network, etc.

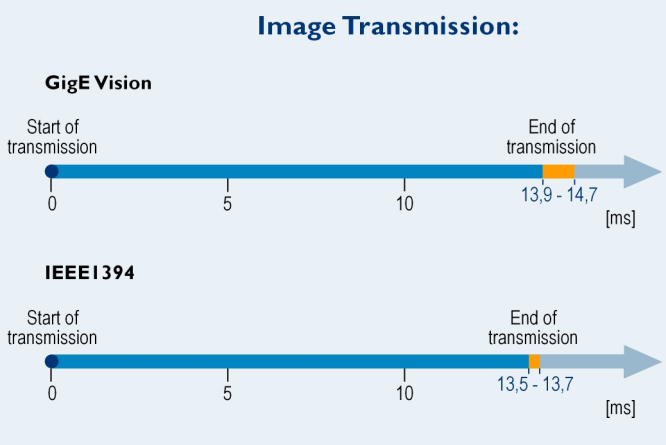

Figure 6: Latency time of image transmission for GigE vs. IEEE 1394

The jitter of the timeline is very small and is approximately 5% of the transmission time of the data stream. This is true because the architecture of the GigE driver is very efficient.

Evaluations have also shown that the process priority has no significant impact on latency time. The largest fraction of the processing time is used in the IP stack or in the GigE Vision driver, which are independent of the process priority.

In addition to the latency time of the camera and the operating system, there are other components that can add additional delays and jitter times. A network switch, for example, can cause this effect. The extent of the delay depends on the hardware vendor. The hardware vendor's data sheet can provide more detailed information.

To create a machine vision application with near to realtime conditions, we suggest using hardware triggered image acquisition and a network topology that does not include a network switch.

Measurement results indicate that GigE is a powerful, easy-to-use technique comparable to IEEE 1394. The CPU load is slightly higher than with FireWire, but can be compensated for with a more powerful hardware architecture. The jitter behavior is directly comparable to FireWire and there are no disadvantages in system architecture.

Basler AG
Germany, Headquarters
Tel. +49 4102 463 500
Fax +49 4102 463 599
bc.sales.europe@baslerweb.com
www.baslerweb.com

USA
Tel. +1 610 280 0171
Fax +1 610 280 7608
bc.sales.usa@baslerweb.com

Asia
Tel. +65 6425 0472
Fax +65 6425 0473
bc.sales.asia@baslerweb.com