Real-Time Development System for Image Processing Engines

Sergio Goma, Radu Gheorghe, Milivoje Aleksic,
Advanced Micro Devices, 75 Tiverton Court, Unionville, ON, Canada L3R 9S3

ABSTRACT

Certain feedback loop based algorithms contained in an image processing engine, such as auto white balance, auto exposure or auto focus, are best designed and evaluated within a real-time framework due to strong requirements of close study of the dynamics present. Furthermore, the development process entails the usual flexibility associated with any software module implementation, such as the ability to dump debugging information or placement of break points in the code. In addition, the end deployment platform is not usually available during the design process, while tuning of the above mentioned algorithms must encompass particularities of each individual target sensor. We explore in this paper a real-time hardware-software solution that addresses all the requirements mentioned before and functions on a non-real time operating system (Windows). Moreover we exemplify and quantify the hard deadlines required by such a feedback control loop algorithm and illustrate how they are supported in our implementation.

Keywords: real time image processing, development system, IEEE1394 (Firewire bus), real time feedback based algorithms, cell phone camera.

1. INTRODUCTION

In the past years the consumer market for cellular phones, digital cameras and camcorders, has enjoyed considerable growth. Current technology trends point towards the migration of the current PC multimedia experience towards similar support on personal portable handheld devices such as cellular phones. It is true that cellular phones with digital cameras and video recording capabilities have become ubiquitous today, and we expect the trend to continue with the introduction of high definition video and increase in sensor resolution. End product expectations on behalf of users also grow as the technology progresses, placing increased weight on improving the quality and reliability of image processing algorithms. In addition, the development of new sensors in the pursuit of increased resolution, process or overall quality improvement, requires careful benchmarking and tuning of existing image processing algorithms for the new parameters and changes introduced. Nonetheless, the dynamic nature of image processing algorithms such as auto exposure, auto white balance or auto focus requires a real-time closed-loop implementation, and is thus something that raises significant challenges in testing and development. A big argument is towards developing this type of algorithms in a real-time workspace, as close as possible to the real life situations, but usually a real-time environment is not development friendly and usually adds additional constraints to the control process, making a possible solution to be dependant of the developing environment.

To address the points illustrated previously, i.e. new sensors rolling out, the constant requirement for algorithms improvement along with their dynamics, and also the unavailability of end target deployment platform during algorithm development, we designed and implemented a real-time hardware-software camera development system that implements the required algorithm control loops on an ordinary PC, using WindowsXP as an operating system.

Our first problem was to select a commercially available data transfer protocol that would meet both our high bandwidth requirements needed by concurrent video streaming and separate real-time camera control channel. The camera control channel requires real-time low-latency in order to implement the above mentioned type of algorithms. Growing sensor resolutions imply a data bandwidth generation that can surpass 400Mbit/s; therefore we required capabilities beyond those currently provided by USB. In addition to high bandwidth, our solution required video streaming with as little intervention from the host CPU as possible. Our choice of interface to satisfy our requirements was the

Real-Time Image and Video Processing 2009, edited by Nasser Kehtarnavaz, Matthias F. Carlsohn, Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 7244, 724409 · © 2009 SPIE-IS&T · CCC code: 0277-786X/09/$18 · doi: 10.1117/12.812372

Firewire/IEEE1394b[7] bus, which offers bandwidths up to 800Mbit/s and direct memory transfers. Paskins in[1] and Steiberg in[2] offer a very good introduction to the IEEE1394 bus concepts.

Also, the availability of custom IEEE1394 link layers already optimized for camera interface and control played an important role in our selection. We also considered Ethernet as a candidate, but we found more complex to achieve what is already native in IEEE1394 (Mbaye et.al. offer in[4] a detailed discussion of the subject). As it turns out, the IEEE1394 bus offers a lot of attributes that made 1394 a preferred choice in test and debug of many real-time systems, from industry[5], automated test equipment[3] and military applications[6].

This paper is organized as follows. In section 2, we illustrate the system architecture, emphasizing the hardware components. Here, we used an off-the shelf, commercially available, IEEE1394 camera development system, that allowed us the flexibility needed by our implementation. We continue in section 3 with our software architecture and a concrete example of a feedback control loop algorithm in section 4, concluding in section 5 with our results and measurements.

2. SYSTEM ARCHITECTURE

To test and develop an image processing engine, real time interaction is a key factor. To achieve this, we chose the following system architecture, available as off-the shelf IEEE1394 camera development system:

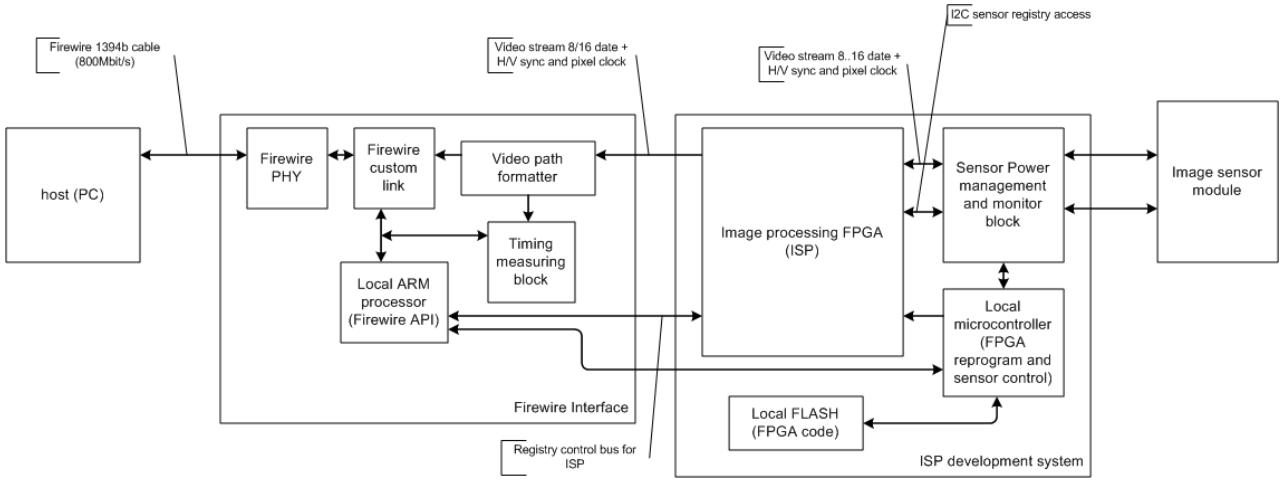


Figure 1: System architecture - real time test and debug of an image processing engine

Currently, operating at 800Mbit/s, IEEE1394b[7] provides two major ways to exchange data: isochronous which offers real-time guaranteed bandwidth performances and asynchronous which offers guaranteed data integrity. Intricate details regarding IEEE1394 operation are beyond the scope of this article however, we will mention that our implementation uses a custom IEEE1394 link layer and only one source node to guarantee real-time performance for asynchronous transactions as well. The custom IEEE1394 link layer does not implement a retry mechanism for asynchronous transactions, but rather sets a status flag that may be polled should an error occur. In this configuration, as the isochronous transfer uses up to 80% of bus bandwidth[8], the control channel implemented using asynchronous transactions offers real-time performance with bandwidths of up to 20% of the bus bandwidth. Because of this, any asynchronous transaction is guaranteed to have a very low latency resulting in isochronous like performance. This situation is illustrated in Figure 2.

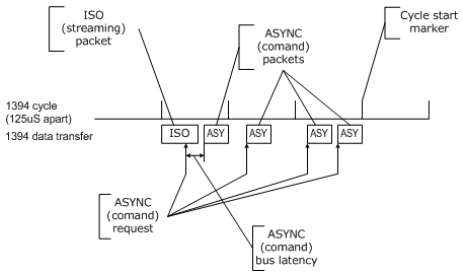


Figure 2: IEEE1394 bus cycle offers real-time latency on asynchronous transactions if one node is present

As the IEEE1394 cycle time is 125us, and each cycle up to 80% of time can be used for isochronous traffic[8], in the worst case, asynchronous traffic latency will be less than 100us, situation illustrated in Figure 2. Of course, this hard requirement can be met with certainty if only one node other then the bus manager (host - PC) is present.

Once that we tailored the use of the IEEE1394b protocol to our needs, and we defined the system architecture the next step was to approach the software architecture in a way that would maintain our performances. As expected this required a custom approach as well which we detail in the next section.

3. SOFTWARE ARCHITECTURE

To achieve real time like capabilities and meet hard deadlines under a non real time operating system, such as Windows, the software architecture requires careful partitioning. In this section we illustrate and discuss the different components of the proposed architecture. We will then exemplify the software operation and its ability to meet certain hard deadlines using a timing diagram. Formulas for maximum processing time allowed will be derived and presented. Figure 3 below, depicts the various layers in the software stack along with the data flow.

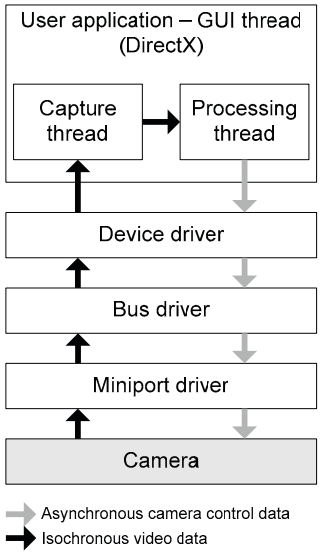


Figure 3: Layered architecture and data flow

We employed standard, out of the box miniport and bus drivers that are shipped with the operating system. A detailed description of these components is beyond the scope of this article, however we will mention that the miniport driver is responsible for interfacing to the open host controller registers (OHCI) on the IEEE1394 capture card and programming DMA transfers in the controller chip on behalf of the device driver, while the bus driver is responsible for device and resource management as well as providing a glue interface layer between the device driver and the low level miniport driver. For more information the reader is referred to the implementation of Lim[3] et.al. Our GUI thread is implemented using DirectX primitives for fast rendering abilities. Although not applicable to the case study in this paper, our software implementation has been tested with the maximum of 63 devices concurrently streaming video. A relatively similar implementation, but with 2 devices only, was attempted under Linux by Mikawa[9].

3.1 DEVICE DRIVER

We adapted a commercially available device driver for controlling our camera development system with a structure similar to that described by Lin et all in[10]. For asynchronous camera control transfers, the driver performs overlapped I/O, allowing a calling thread not to block until the transfer is completed. For isochronous video data transfers, the driver shares a circular buffer I/O queue with the capture thread. In addition to providing backing store memory for DMA operations, the buffers in the buffer queue include header descriptors containing notification events on which the capture thread waits. To allow for fast copy operations, the size of the backing store memory in each buffer is set to be the size of one video frame rounded up to the nearest integer multiple of a memory page size. The size of the buffer before rounding is calculated according to the formula below:

$$Sz_{buff} = (B_{ppack} + 8) * H * L_{ppack} \quad (1)$$

Where,

B_{ppack} is the number of bytes per IEEE1394 isochronous packet* calculated according to (2),

L_{ppack} is the number of sensor/ROI lines per IEEE1394 isochronous packet,

H is the height in lines (total number of sensor/ROI lines) .

The total number of bytes per isochronous packet is calculated as:

$$B_{ppack} = W * B_{ppix} * L_{ppack} \quad (2)$$

Where,

W is the width in pixels (total number of sensor/ROI pixels) ,

B_{ppix} is the number of bytes per pixel.

3.2 CAPTURE THREAD

We implemented the capture thread to be a thread running at an elevated priority in order to achieve the goal of providing a constant flow of buffers for video DMA transfers as specified below:

* The maximum size of an IEEE1394 isochronous packet in this implementation is 8192 bytes; there is an additional requirement that this number be divisible by 4[11].

$$Rate_{buffer_attach} \geq Rate_{camera_frame} \quad (3)$$

As mentioned above, the capture thread shares a circular buffer I/O queue for video transfers with the driver stack. In addition it shares a second queue, the processing queue, with the processing thread.

The capture thread operates according to a simple state machine with three sequential states namely *initialize*, *play* and *stop*. In the *initialize* state, the resource allocation is performed in terms of IEEE1394 bandwidth and transfer channel. Upon successful initialization, the *play* state is entered. Initially, in this state, all the buffers in the DMA transfer circular I/O queue are checked out from the queue and submitted to the driver below. As the buffers are checked out, each buffer descriptor is inserted into a descriptor only circular queue for sequencing purposes. Ultimately, after all the buffers are attached for DMA, the video capture can start. The DMA transfers finish according to the order in the sequencing queue. As each transfer is completed, the capture thread verifies and strips IEEE1394 isochronous packet headers from the received data, checks out a buffer from the circular buffer queue shared with the processing thread and performs a fast copy of the data into this buffer. It then signals the processing thread with a data ready event and checks the I/O buffer back into the DMA queue. The *stop* state can be entered asynchronously as a result of the user exiting the application or stopping the capture, or autonomously as a result of an error in the *play* state. Memory de-allocation and resource cleanup is performed as a last step in this state.

3.3 PROCESSING THREAD

Similarly to the capture thread, the processing thread runs at the same priority level. As mentioned above, it shares a circular buffer queue with the capture thread. The latter signals and awakes the former through a notification event upon buffer transfer completion. All processing steps act upon the data contained in the latest received buffer. As a last processing step, a fast copy is performed into the user application display buffer for visualization and a message is posted to the main user application for rendering at which point the shared buffer is checked back into the shared queue for subsequent reutilization.

3.4 TIMING

The figure below illustrates a typical timing example of the capture, processing threads and device driver interaction. The I/O capture buffer queue is comprised of two buffers that are alternated.

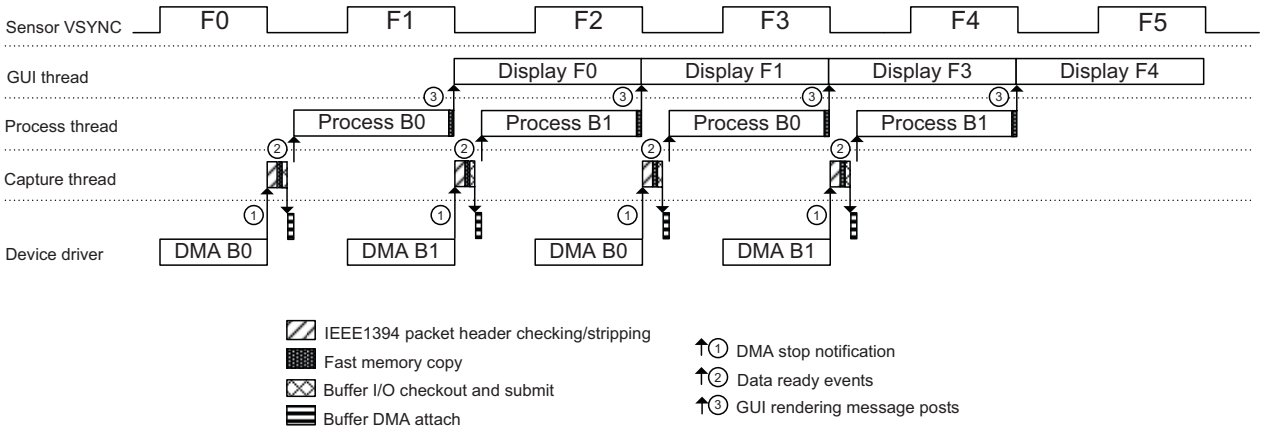


Figure 4: Typical timing

Notice in Figure 4 that some processing time is taken up by housekeeping tasks such as memory copying and inter thread signaling. In the results section we will quantify and analyze this latency. The capture and process thread are implicitly synchronous to the sensor vertical sync signal, as the state machine in the capture thread waits upon the buffer ready event signaled by the end of the DMA transfer. The process thread sleeps until the capture thread has data ready for processing. Another reason to have the processing in a separate thread, is that sometimes, evaluation of an algorithm may require processing times longer than a frame, and as capture thread does not wait for the process thread to finish, all frames are guaranteed to be captured. As rendering of the data is implemented with DirectX making the GUI thread synchronous to the display refresh, the synchronization to the sensor vertical sync signal is compromised. The connection between the process and GUI threads is achieved through a Windows messaging system based message queue. The queue level on the GUI thread side is monitored such that the GUI thread is not stalled, should a race condition occur.

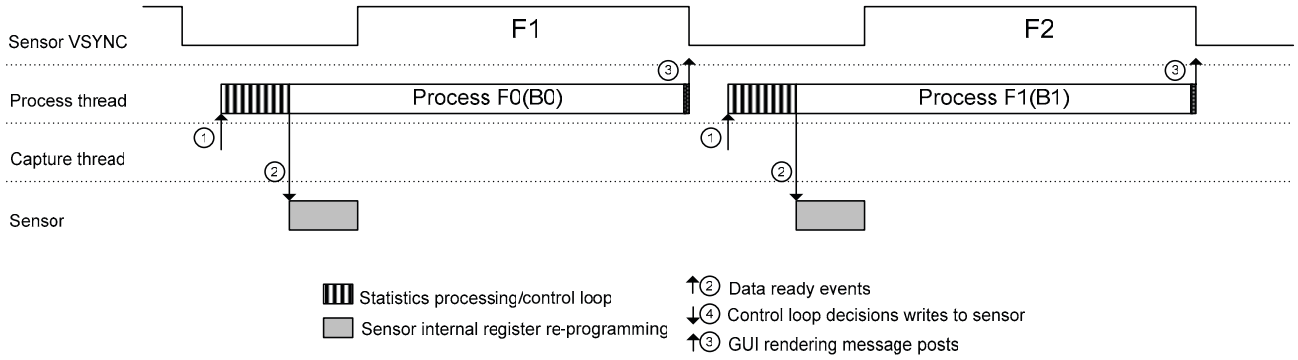


Figure 5: Sample control loop algorithm timing

From a timing perspective, for the purpose of this paper, we consider two cases of interest. The first case is illustrated in Figure 4 above in which the only hard deadline to be met is to finish processing frame N by the time frame $N+1$ has finished DMA transfer into the capture buffer queue.

This allows for a maximum processing time given by equation 4 below:

$$T_{proc_{\max}} = T_{frame} + T_{blank} - (T_{strip} + T_{copy} + T_{submit} + T_{attach}) \quad (4)$$ where,

T_{frame} is the sensor frame time

T_{blank} is the sensor blanking time

T_{strip} is the time required for packet header stripping and/or checking

T_{copy} is the time required for copying the data into to the processing buffer

T_{submit} is the time required for the capture thread to re queue the buffer

T_{attach} is the time required for the device driver to re attach the buffer

The second case, illustrated by in Figure 5 is similar to the first but has an additional hard deadline. If the processing thread is running an algorithm consisting of a feedback based control loop such as auto exposure or auto white balance, it is often the case that control decisions must be written back to the sensor within the vertical blanking time in order to be applied to the next frame. As such, for processing of decisions that must be applied during sensor vertical blanking before the start of the next frame, we observe the following maximum processing time:

$$T_{Vproc_{\max}} = T_{blank} - (T_{strip} + T_{copy} + T_{attach}) \quad (5)$$

4. IMAGE CONTROL LOOP

As an example, the auto exposure algorithm is used to analyze latencies involved on control loop on host (PC) side. From the software perspective, the algorithm needs to analyze a captured frame and change sensor timing parameters. The example analyzes a CMOS image sensor that has a rolling shutter to implement exposure.

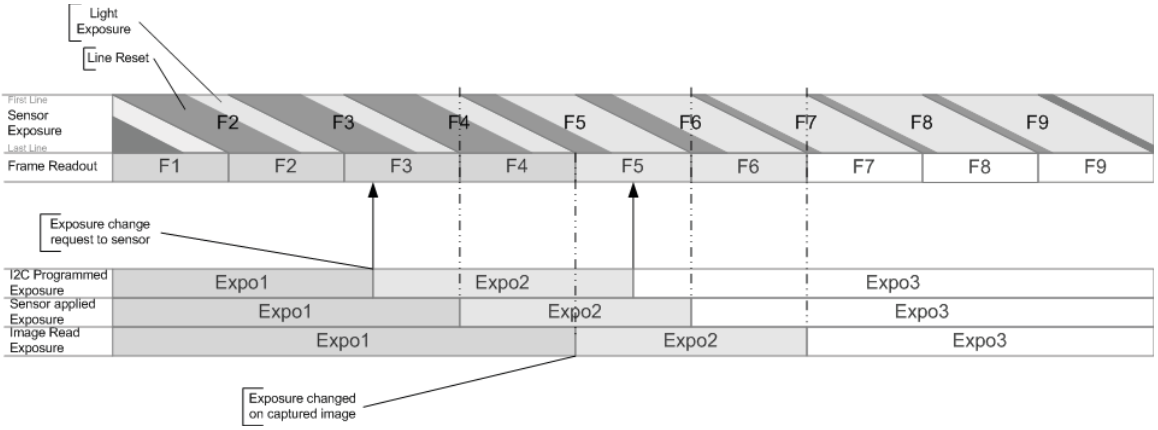


Figure 6 CMOS sensor rolling shutter exposure and timing implications

A CMOS camera sensor that uses a rolling-shutter type of exposure needs additional consideration, as any change in exposure will be propagated in the second captured frame after the moment of exposure change (see Figure 6). Because of that, it is critical that a control loop that implements an auto-exposure algorithm fits entirely in the vertical blanking interval. This requirement is based on the following assumed structure:

- captured image is analyzed and statistics relevant to the implemented control loop algorithm are collected. The earliest time this can be accomplished is after the full frame was captured on host side.

- the algorithm analyzes statistics data and decides an update of timing/exposure registry of the sensor (usually through I2C interface that connects the image sensor to the ISP development system - see

Figure 1).

All processing has to be executed during vertical blanking interval in order to meet rolling-shutter requirements of the CMOS image sensor. If those conditions are not met, the new exposure will be delayed one more frame. To illustrate this situation we refer to the following pictures that show a successful update of the sensor during vertical blanking.

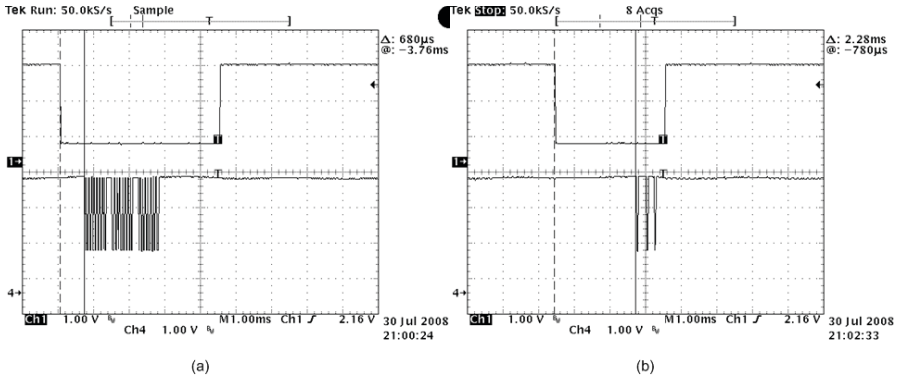


Figure 7 Control loop interaction during vertical blanking: (a) using hardware generated statistics (b) using software generated statistics

5. RESULTS

We implemented the solution and ran latency experiments under Windows XP on a Pentium 4 machine with hyper-threading running at 3.2GHz, with 1.00GB of RAM. Below, we present system level latencies and bandwidth measurements that were determined by using the system level approach depicted in Figure 1.

On one of the corner case tested, the streaming parameters used were:

Pixel clock = 98.304MHz (1394 bus clock)

Horizontal active data = 1024 pixels

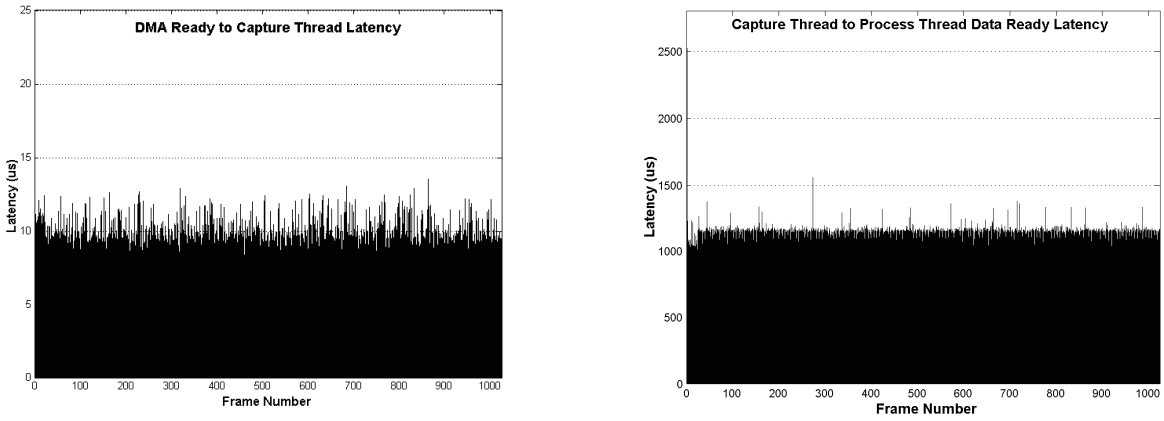
Horizontal line = 1536 clocks or 15.63us

Vertical active data = 1024 lines

Frame time = 1028 lines or 16.06ms, which yields to 62.26 frames/second

Pixel size = 8bits/pixel

In this case, we achieved the theoretical maximum isochronous data bandwidth capability of the bus (8192 bytes isochronous payload – S800[7]) at 62.5Mbytes/s, or 500mbit/s. We added asynchronous camera control traffic to this bandwidth and measured the latency. The results are depicted in figures 7 through 10.



(a) (b)

Figure 8: Isochronous latency results - (a) DMA ready to capture thread; (b) capture thread from point measured in (a) to process thread awakening (includes IEEE1394 packet header check/strip and memory copy)

The isochronous results refer to the points in Figure 4 of the timing diagram.

Statistic	DMA to Capture (us)	Capture to Process (us)
Average	10.205	1154.792
Minimum	8.390	1011.545
Maximum	13.522	2780.255
Standard Deviation	0.900	77.974

Figure 9: Isochronous latency statistics

To reach this bandwidth, the video transmission parameters were 8192 bytes/cycle, or 8 sensor lines per isochronous packet.

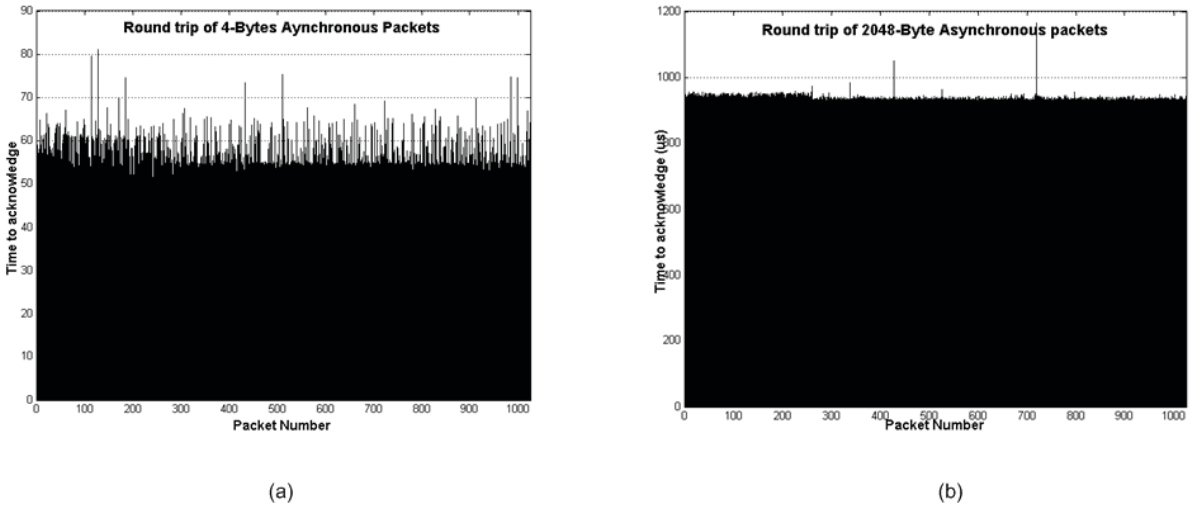


Figure 10 Asynchronous round trip latency - (a) 4 byte packet; (b) 2048 bytes packet; in both cases data is copied to the host (ARM) before an acknowledge is generated

The following table shows full round-trip latencies: a control command is send from the process thread, is received over the IEEE1394 bus, the local processor (ARM) reads the packet and writes the result to the image processing FPGA (see Figure 1). The local registry bus exchange between the local processor and the image processing FPGA is parallel, synchronous and write transactions are pipelined, so no delay can occur on this connection.

Statistic	2048 bytes (us)	4 bytes (us)
Average	940.421	58.711
Minimum	1164.783	51.560
Maximum	930.139	81.696
Standard Deviation	11.278	4.223

Figure 11 Asynchronous latency statistics

As the smallest asynchronous transaction available on IEEE1394 is quadlet based (4 bytes), the right column in Figure 11 shows the associated latencies for that case, whereas the left column shows the largest asynchronous packet allowable by the SP1 implementation[13] of the Windows bus driver. This is consistent with the real time RTLinux results Sarker[12] et al., proving that with a careful planned software architecture, a non-native real-time operating system like windows, can offer similar performances to a real-time operating system.

6. CONCLUSIONS

Designing and testing an image processing engine is an involved process with requirements that imply real-time development capabilities. We presented a system level architecture that meets the demanding requirements of a real-time imaging system, while making the interface transparent, thus allowing architecture teams to focus on the task at hand. Furthermore we implemented our real-time solution on a PC computer running a non real-time operating system (WindowsXP) and using off-the shelf hardware, rather than the end product platform on which the image processing engine would be deployed. We presented the architecture of the system with emphasis on the interface and software side

on host (PC) and we showed as results, latencies and bandwidths that were measured in order to prove that the proposed solution can meet the requirements of a real-time imaging system.

REFERENCES

[1] Paskins, A. "The IEEE 1394 bus" New High Capacity Digital Media and Their Applications (Digest No: 1997/114), IEE Half-Day Colloquium on Volume , Issue , 12 May 1997 Page(s):4/1 - 4/6

[2] Steinberg, D., Birk, Y., "An Empirical Analysis of the IEEE-1394 Serial Bus Protocol," IEEE Micro, vol. 20, no. 1, pp. 58-65, Jan/Feb, 2000

[3] Purcell, A., "IEEE 1394: a growing computer connectivity standard for automated test " AUTOTESTCON Proceedings, 2000 IEEE Volume , Issue , 2000 Page(s):398 – 402

[4] Mbaye, M.M., Tohio, B., Savaria, Y., Pierre, S., "Performance of a firewire-Ethernet protocols conversion on an ARM7 embedded processor" Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on Volume 2, Issue , 4-7 May 2003 Page(s): 1267 - 1270 vol.2

[5] Frommhagen, K., Nauber, P., Schelinski, U., Scholles, M. "1394AP: a protocol for deterministic industrial communication via IEEE 1394" Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on Volume 2, Issue , 19-22 Sept. 2005 Page(s): 8 pp. –

[6] Bai, H., "Analysis of a SAE AS5643 Mil-1394b Based High-Speed Avionics Network Architecture for Space and Defense Applications", Aerospace Conference, 2007 IEEE Volume , Issue , 3-10 March 2007 Page(s):61 – 70

[7] IEEE Std. 1394b - 2002 IEEE Standard for a High-Performance Serial Bus - Amendment 2

[8] D. Anderson, FireWire System Architecture, Addison Wesley, Reading, Mass., 1999

[9] Mikawa, M., "A programming interface for IEEE 1394 video device in robot vision" SICE 2003 Annual Conference Volume 1, Issue , 4-6 Aug. 2003 Page(s): 766 - 771 Vol.1

[10] Lim, H-S., Park, D-H., Kang, S-J., Oh, P-J., "Priority queue-based IEEE 1394 device driver for supporting real-time characteristics" Consumer Electronics, 2000. ICCE. 2000 Digest of Technical Papers. International Conference on Volume , Issue , 2000 Page(s):198 – 199

[11] IEEE Standard 1394-1995, Standard for a High Performance Serial Bus, IEEE Press, Piscataway, N.J., Aug. 1996

[12] Sarker, M.O.F., Kim, CH., Cho, J-S., You, B-J., "Development of a Network-based Real-Time Robot Control System over IEEE 1394: Using Open Source Software Platform" Mechatronics, 2006 IEEE International Conference on Volume , Issue , July 2006 Page(s):563 – 568

[13] MSDN Developer Guide, Windows Driver Kit: Buses, http://msdn.microsoft.com/en-us/library/ms789373.aspx