Spring 5-31-1999

Improved SU320 NIR camera image processing system

Weimin Shi
New Jersey Institute of Technology

ABSTRACT

IMPROVED SU320 NIR CAMERA IMAGE PROCESSING SYSTEM

by
Weimin Shi

This paper describes an improved image processing system which is applied in solar observation. In order to get the higher performance in frame grabbing rate, the system is using PCI Master Bus with on-board memory frame-grabber. From software side, the application program has been modified to reduce the grabbing time, which speeds up the whole process significantly. The paper described them in detail.

IMPROVED SU320 NIR CAMERA IMAGE PROCESSING SYSTEM

by
Weimin Shi

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

May 1999

APPROVAL PAGE

IMPROVED SU320 NIR CAMERA IMAGE PROCESSING SYSTEM

Weimin Shi

Dr. Dao-chuan Hung, Thesis Advisor — Date
Associate Professor of Computer Information Science, NJIT

Dr. Haimin Wang, Thesis Advisor — Date
Associate Professor of Physics Science, NJIT

Dr. Pengcheng Shi, Committee Member — Date
Associate Professor of Computer Information Science, NJIT

BIOGRAPHICAL SKETCH

Author:	Weimin Shi
Degree:	Master of Science in Engineering Science
Date:	May, 1999

Undergraduate and Graduate Education:

- Master of Science in Computer Science
  New Jersey Institute Technology, Newark, NJ, 1999
- Master of Science in Materials Science & Engineering
  New Jersey Institute Technology, Newark, NJ, 1998
- Bachelor of Engineering in Electronic Engineering
  Xi'an Jiaotong University , Xi'an, P. R. China, 1992

Major: Computer Science

Dedicated to my parents and my husband Weili

ACKNOWLEDGMENT

The author wishes to express her sincere gratitude to her advisors Professor Haimin Wang and Dao-chuan Hung for their guidance, inspiration and support throughout this thesis.

Special thanks to Professor Pengcheng Shi for serving as a member of the committee.

The author appreciates the timely help and suggestions from Hangjun Chen and other members in Solar Observation Research Group.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Problem, Application and Challenge

Image processing is used for two somewhat different purposes. First, improving true visual appearance of images to human viewers. Second, preparing images for measurement of the features and structures present.

SU320 NIR Camera Image Processing System is intent to allow images to be captured from the digital output of the SU320 NIR camera and process the images. The system proposes a method demonstrating how to grab multiple frames in PC memory and process them once all the image data are in PC memory. According to the real procedure we use liquid crystal to get retarded images as we grab in image data from camera to PC memory. Our target is to speed up the frame-grabbing process to have enough time to control liquid crystal reversing.

We bought camera, frame-grabber board and software developing tools from the three companies. The task is to set up the system and configure it to fit our own requirements. The further solar research on filament detection will be based on the images we captured.

1.2 Keywords and Abstract Phrases

PCI: Peripheral component interconnect.

IC-PCI: Image capture, *PCI*-bus.

ISA: ISA was standard PC bus for imaging before *PCI.*

AM: Acquisition module.

AM-DIG: *AM*, digital acquisition module.

AOI: Area of interest.

DMA: Direct memory access.

CHAPTER 2

SYSTEM OVERVIEW

2.1 SU320-1.7RT Camera

Room-Temperature, High Resolution, Near-Infrared Area Camera

Features:

- RS170 and CCIR Frame Rates
- 320 x 240 pixels

2.2 AM-DIG Module

AM-DIG is a plug-in module that provides an interface to digitizing or digital output cameras and video sensors. The module receive pixel data based on timing signals (clock, line enable, frame enable) provided by the digitizing camera and synchronizes this data to the mother board timing. The *AM-DIG* can provide clock and mode control signals to the digitizing camera.

The *AM-DIG* module controls camera selection and image acquisition. Figure below is a block diagram of the *AM-DIG*.

2.3 IC-PCI Specifications

Memory

- Image Memory: 4MB linear memory
- Supports *AM* trigger modes
- Acquires 8 bit, 16 bit, and 24 bit image data at clock rates up to 40 MHz

Host access

- *PCI* Bus interface to video memory , control registers, and AM

- Image memory: occupies 4 MB of memory
- Registers: mapped into three different I/O spaces by the *PCI* host
- *AM*: occupies 8KB of memory address space
- Image data: 32-bit access only
- Control registers and *AM*: 160bit word access only

Acquisition

- Frequency: Up to 40 MHz read access to FIFO on *AM*
- Supports *AM* trigger modes
- Supports all *AM* family acquisition modules

Environmental

The *IC-PCI* compiles with the *PCI* short card mechanical spec, allowing installation in any slot of a *PCI* based machine.

- Board Size: *PCI* short card, 6.88 by 4.205 inches
- Operating Temperature: 10 to 60 degree Celsius
- Relative Humidity: 0 to 90% non-condensing
- Power Requirements: typical 0.50 Amperes at +5 Volts

2.4 System Requirements

OS: Window NT/98/95

Memory: minimum 32M RAM

Video card: *IC-PCI*

CHAPTER 3

SYSTEM ANALYSIS AND IMPROVEMENTS

3.1 SU320-1.7RT Camera

3.1.1 Features and Functions

This room temperature-operated, solid state camera is compact, lightweight and easy-to-use. It weighs approximately 2.6 lbs. and it s dimensions are 4" x4"x6". Offering both video and digital output, the SU320-1.7RT provides a reliable method of capturing light from 900nm to 1700nm, with peak responsively at 1550nm. The SU320 InGaAs focal plane array offers a high quantum efficiency exceeding 70% over its entire wavelength band without the need for external cooling.

The 320 x 240 pixel area camera enables the user to reach those goals that in the past seemed unattainable. These leading instruments are utilized in various scientific and industrial applications including: Astronomy, Environmental Control and Emission, Enhanced Vision and Imaging, Semiconductor and Failure Analysis, Non-Destructive Testing, Chemical Analysis, and Tele-communications. While the video SU320 is operated as easily as a standard camcorder, the digital SU320 is designed for more specific applications where variable integration times and external triggering are required. The digital model has an integration time range of 127 microseconds to 16.3 milliseconds (in 8 steps). In addition, the RS170 models are compatible with standard analog and digital frame-grabber boards (included) for convenient computer interfacing.

The RS170 allows users to capture images in the near IR region with unprecedented sensitivity. With a full well capacity of 10 million electrons per pixel and electronic read-out noise of less than 2,000 electrons per pixel, the user will experience a full 12-bit dynamic range.

The SU320-1.7RT has both a video output for direct interface to standard monitors and VCRs and a digital output for integration into a customer's computer. The camera is superior to any other solid-state or vacuum tube in the NIR spectrum. It is lightweight, easily transportable, requires no cooling, and does not exhibit defects such as image lag, persistence, blooming, low damage threshold, or tube wear-out.

3.1.2 The Linearity Memory of the SU320-1.7T

The InGaAs focal plane arrays in Sensors Unlimited it's room temperature near infrared cameras are "active pixel" devices; the photo current from each pixel is amplified before it is integrated on the charge storage capacitors. The FPAs are linear, i.e. the integrated charge which appears as a voltage at the output of the FPA is proportional to the optical power.

There are restrictions, however, on the degree of linearity. Specifically: The degree of linearity is greater than 98% for signals up to half of the saturation charge. This means that when the FPA signal vs. optical power is fitted to a straight line with a zero intercept, the measured response is within 2% of the calculated response. The degree of linearity is greater than 95% for signals up to 80% of the saturation charge. If the FPA (or portion of the FPA) is exposed to excessive light, the internal gain will automatically decrease. This is the source of the anti-blooming feature of Sensors Unlimited FPAs. This automatic gain reduction is based on the total photo current from all of the pixels and the gain reduction affects all of the pixels. If a very intense spot of light (such as from a laser) strikes a portion of the array, it will reduce the global gain of the entire array, which defeats the linearity.

3.2 Frame Grabber

3.2.1 Select the Right Frame Grabber

So many frame grabbers are available in the current market, it can be a tough task to decide which one is best suited for a particular image vision application. Determining what is required for a particular application becomes a critical factor and can determine the success or failure of a image vision project.

The frame grabber features required for image vision differ significantly from multimedia, medical, or military applications. One of the most common vision tasks requires that when a certain event occurs, an image is acquired into host computer memory where it can be analyzed and a decision made. For example, filaments are moving in the solar, an image is taken and a computer-based algorithm determines the quality of the object.

Inspection rate, algorithm details, and physical constraints will often determine the speed, resolution and type of camera and optics used. While RS170 cameras are common, the need for more detail and faster acquisition rates are rapidly driving image vision in the direction of higher resolution, progressive scan cameras. Couple this requirement with the need for triggered acquisition, frame reset, strobe control, and some digital I/O , So, what's required?

- Acquisition, Camera Selection

Most image vision solutions are implemented using analog cameras, while digital output cameras are available and may give better signal to noise.

- Camera Control

A minimum requirement for image vision is accurate A/D circuitry and either a very good PLL or the ability to provide a clock to the camera (drive the camera timing). Without this, the fidelity of the acquired image is in question and both quantitative and qualitative image vision applications are compromised. Beyond this, camera control becomes important. Most image vision applications require that an image be acquired as a result of an external event, a trigger. While many frame grabbers can provide this capability, many cannot issue a strobe signal as a result of this external trigger. If they do, there may not be a way to adjust or delay the strobe pulse relative to the trigger event. The ability to properly time the strobe to the trigger pulse can vastly improve lighting on the subject.

Additionally in many image vision applications a frame reset capable camera is required. A frame-reset camera allows the camera's timing to be interrupted, and "reset" at any point, guaranteeing instant return to the top of frame. This results in acquisition of the image immediately upon receipt of the external event. Without this ability, a delay occurs between the issuance of the trigger signal, and when the camera begins to acquire and provide data to the frame grabber. Even worse, the delay will be variable because the trigger signal and camera timing are not synchronized. For interlaced RS170 signals this

delay can vary from almost 0 to an entire frame (33 ms). Input signal conditioning such as the ability to control gain and offset are important to minimize effects from camera variability or lighting fluctuations, and to provide the best camera signal to the A/D. Also, a lookup table is frequently used at the input to modify the data for easier signal processing or better display capabilities.

- Processing the Data

Many image vision applications require that the host computer control several devices and manage several events, the trend has been toward multitasking interrupt driven software. Thus, a frame grabber and software those are capable of operating in an operating system like Microsoft Windows NT using interrupts and multiple threads is desirable. Often to lower manufacturing costs, frame grabber companies will not put much, if any memory on the frame grabber itself.

Instead these frame grabbers without memory rely on the speed of the *PCI* bus to perform line by line transfers of the acquired data to the host memory. While this method works, and can in some cases achieve real time or close to real time capture rates, the drawback to the memoryless frame grabbers does not become apparent until viewed in the context of a image vision system.

As mentioned, image vision systems are not just acquiring an image for display. Instead these systems are multitasking. They are displaying images, reacting to interrupts, processing I/O, and analyzing acquired data. Frame grabbers without memory show the real "cost" in the increased load to the CPU. The host CPU is simply busy handling the line by line data transfer and may not be available for the required processing.

- Software

When considering software, the image vision developer first has to assess how much of the software integration task he is willing to assume. The choice of whether to write a custom algorithm or use an off-the-shelf package will depend upon the uniqueness of the application and the time constraints of the project.

In general, using software packages written specifically for image vision not only speed the completion of the project, but usually result in a faster, more reliable solution. One of the most common bottlenecks in this process is the integration of the host-based software with the frame grabber. Because of this, strong consideration should be taken to the maturity of the interface between the frame grabber's software and that of the image-processing package being considered. Many frame grabber manufactures software is simplistic and requires that common image vision functions be written by the end user.

Basic frame grabber software functionality should at a minimum include, methods for achieving triggering, strobing, frame reset, data transfer to host, image display, and a way to dynamically tweak the camera interface. These are invaluable when first setting up the system to acquire an image.

Once the desired image is acquired and delivered to the host memory, the task of actually solving the image vision problem begins. Here again, access to a powerful image-processing library that is designed for image vision (vs. general purpose image analysis), can greatly improve performance, decrease development costs and improve success rates. Before designing the system, there are several questions to be considered:

1. Review the camera requirements - Can the frame grabber handle it? Are you going to have to develop the camera interface, or does the frame grabber manufacturer deliver a ready-to-go plug and play interface to the camera you need?

2. Consider the quality of the digitization. Will noise obscure or make the measurement unreliable? "Multimedia" boards typically fall into this category.

3. Are you going to need? Trigger, Strobe, Frame Reset, PLL acquisition, Timing to the camera and Input signal conditioning.

4. What kind of demands are you going to make on the CPU for processing? Is the frame grabber going to steal too many CPU cycles?

5. Look at the maturity of the software, and compatibility with other image vision products. Are the image vision algorithms you need for your application available, or will you have to write your own?

3.2.2 Advantages of Frame Grabbers with On-Board Memory

Recently, there has been quite a bit of confusion over the issue of whether a frame grabber should have on-board memory or should just transfer image data directly to the host computer. The answer, of course, depends on application requirements, but there are a number of issues when considering needs.

If your application is automated imaging or image vision, chances are you're going to need more performance, not less. This is the fundamental difference between frame grabbers with on-board memory and those without.

- Automated Imaging & Image Vision

Why are automated imaging and image vision applications different from other applications? Because they are more demanding, requiring the computer to extract information from an image, make a decision based on it, then repeat the process in a continuous fashion. The solution to automated imaging problems invariably requires compromise. These compromises involve many aspects of the problem, and, as a result, there is usually never enough performance to solve the problem at the desired cost. The universal goal is to get more performance for the same price. As we'll discuss here, this means using a frame grabber with on-board memory!

- Peak Rates vs. Sustained Rates

There are two different kinds of frame grabber memory architectures: FIFO-based and on-board memory-based. A FIFO-based design streams data into a 1 x 1K or 1 x 2K temporary buffer at the output rate of the camera. Data is sent from the FIFO to the host computer's memory after the FIFO fills. The problem occurs in that there is a data rate mismatch between the incoming data (usually 0 Hz to 40 MHz, slow scan video to very fast scan video), and the computer's *PCI* bus, which must transfer data across the bus at 132 MB/s.

The numbers can be very deceiving as manufacturers of FIFO-based boards take advantage of this situation by quoting only burst or peak transfer rates. In a *PCI* bus-based computer, this will always be 132 MB/s, as the *PCI* bus specification for burst mode requires data to be transferred at the maximum bandwidth of the bus. So, realistically, all burst, bus-mastering devices can claim 132 MB/s transfer rates. But note that these are peak mode rates. Peak rates are not the issue.

Sustained data transfer rates are what's important. If the vendor doesn't tell you the sustained transfer rate, you can determine this figure on your own by taking the total amount of data that has been sent, and dividing it by the time it took to move that data from source A to destination B. The setup times, the times to refill the FIFOs, the time to arbitrate, etc., are all taken into account. Then take a careful look at this number. If the sustained transfer rate is less than your incoming data rate, you are losing data during the transfer. If this number is greater, you have excess bus time to do other things, such as image processing analyses.

- Previous Works of Bus Designs

Why has peak vs. sustained transfer become a big issue these days? With the advantage of the *PCI* bus, (total bus capacity 132 MB/s as discussed above), it is possible to transfer video from a digitizer directly into host memory or to a display at full frame rates. For instance, RS170 video needs a transfer rate of about 10 MB/s; true color (RGB) needs triple that, or 30 MB/s, which is still well within the capacity of the *PCI* bus. Therefore, if a vendor builds the appropriate circuitry, is there a need for a frame store on the frame grabber, as was required when slower bus architectures abounded?

Take the PC at bus, for example, which had a maximum bandwidth of about 8 MB/s. If users saw sustained rates of 3 MB/s, they were looking at a good design. Obviously, this was below the required RS170 rates, so if anyone wanted to achieve respectable transfer rates, they had to use on-board memory.

- Bus Mastering

A second issue to consider is whether or not the frame grabber has bus mastering capabilities. The *PCI* bus is efficient enough that you can transfer data from the frame grabber to host memory by having the CPU control the data movement. Unfortunately,
this hogs processor time and turns your CPU into an expensive address generator. A much better alternative is to use a frame grabber that can act as a bus-master when needed.

- Application Issues

Finally, the most important considerations concern the application. Let's look at an example. Each frame is 30 ms long, we need every 10th frame, and only another 30 ms more to process the data. Since we have 270 ms of time between frames, it doesn't make a big difference if we can get the data to the CPU's memory in 1 ms or 100 ms.

On the other hand, consider the case where we need to process every frame. Data is coming in every 30 ms. The data must be moved to memory within the 30 ms, otherwise the next frame, or part of it, will be lost. Even if we can move the data at frame rates, will we have any time left over for processing the data?

The latter situation is the one faced by automated imaging and image vision applications. When parts are coming down a line and the computer must evaluate each one for defects, the CPU can always use more power! Whether you want a more robust algorithm to detect flaws or have more I/O needs, whether you need to use higher precision to calculate your results or just want to process more parts in less time, having more CPU power for the same price is an attractive solution. So, let's take a look at how on-board memory can solve the above problems.

- Why Need On-board Memory

Memory on the frame grabber, coupled with a *DMA* engine, allows data to simultaneously come in from a sensor and be transferred to the CPU's memory with very little overhead from the CPU! Let's compare the performance of a frame grabber with on-board memory and a FIFO-based frame grabber.

Assume that the frame grabber with on-board memory has enough memory for at least two frames of image data and can transfer data at 100 MB/s, sustained. The FIFO board has a 1K FIFO and can transfer data at 132 MB/s for the full 1K. Let's also assume that the interrupt latency under Windows NT is about 150 μs (a time we have measured here at ITI). Data is coming in at about 10 MB/s (640 x 480 x 30 fps = 9.3 MB/s).

As you can see in Figure 1, a memory system on the frame grabber allows the data to be sent to the host in one single burst. Since that burst rate is so high (a mere 4 ms to send the entire frame to the host), it only comprises about 10% of the total frame time, leaving the remaining 90% (29 ms) for processing.

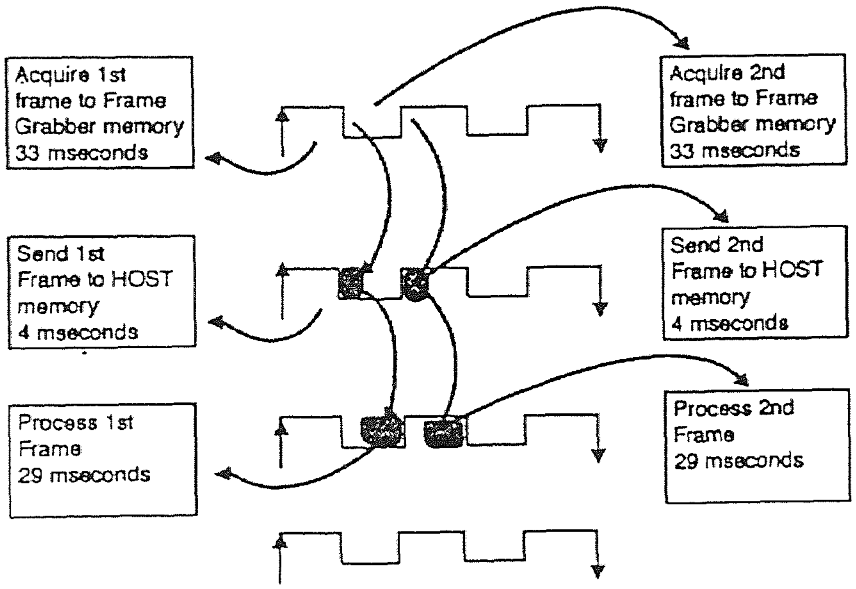
Acquire 1st frame to Frame Grabber memory 33 mseconds
Acquire 2nd frame to Frame Grabber memory 33 mseconds
Send 1st Frame to HOST memory 4 mseconds
Send 2nd Frame to HOST memory 4 mseconds
Process 1st Frame 29 mseconds
Process 2nd Frame 29 mseconds

Figure 3.1 On-Board memory allows data to be sent in a single burst

The way this is accomplished is to "Ping-Pong" the data transfer with the image acquisition. Since the memory on the frame grabber is dual-ported (the independent input and output ports can work simultaneously), one buffer on the frame grabber is set up for input of the video data. This takes 33 ms to fill. Once filled, a second buffer starts filling up with the second frame. Meanwhile, the data in the first buffer is transferred to the host memory.

The CPU does not need to be involved with the data transfer if there is a DMA engine on the frame grabber. This leaves the CPU free to process the data or whatever else is needed. As an added benefit, since there is an interrupt only on a vertical blank which occurs in the 10's of ms time frame vs. the 160µs it takes to process an interrupt under NT, the interrupt time is negligible.

On ITT's *IC-PCI* and PC Vision frame grabbers, which have DMA engines that provide transfer rates of over 100 MB/s, less than 1% of the CPU is utilized under NT 4.0 when transferring 640 x 480 images. Pretty impressive when compared to the typical FIFO-based frame grabber, which hogs the CPU over 42% of the time during data transfers.

FIFO Considerations: The main difference between FIFO and on-board memory frame grabbers is that FIFO boards cannot "ping pong" image acquisition and transfer. As mentioned earlier, FIFOs stream data out. You can never get the data to the host any faster than the input rate. You do see the data sooner, since data starts arriving in host memory after the FIFO first fills. But there is a data rate mismatch in the sense that the incoming data can be anywhere from 0 Hz to 40 MHz, while the *PCI* end of the FIFO can transfer at 100+ MB/s.

Let's look at some numbers. At RS170 rates, it takes about 50 µs to capture a line of video (actually, its 63.5 µs, but we'll use round numbers), at approximately 10 MB/s. Assuming you have a 1K FIFO working on half full lines of data that are 512 pixels long, and

PCI transfer at 100 MB/s , then the FIFO fills for 50 µs, and empties 5 µs later. This repeats for every line of video.

However, on most *PCI* systems, the *PCI* bus will not allow for data transfers that are 512 cycles long. Usually, the transfer is limited to 64 packets before it must rearbitrate. If you are running with interrupts, you can't process these fast enough under NT without missing lines of video! If you are processing without interrupts, e.g. polling, then you have the CPU in an extremely tight loop waiting for each line. You cannot get any processing done, since every 50 µs you have the next.

Display Considerations: Issues remain the same when transferring data to VGA memory. With many FIFO cards, you will not see real-time video. In fact, with some FIFO cards that we have tested, moving the mouse causes a dramatic "hiccup" in the display. This mouse movement took such a bite out of system resources that there wasn't enough left to give to the FIFO card. The end result...lost.

Since we need every image data, not every 10th or 50th frame, we need on-board memory on frame grabber. Anything else will result in lost data. It can also result in hogging of the CPU, which could be doing more productive things, like analyzing all that great image data we collected.

3.3 Advantages of PCI Master Bus

PCI Bus Mastering with a frame grabber provides a 40 times speed improvement over *ISA* bus frame grabbing while lowering frame grabber costs by more than 50%.

3.3.1 PCI vs. ISA

ISA was standard PC bus for imaging before *PCI*. Unfortunately it is a woeful under performer for imaging. A 16-bit bus provides effective bandwidth of 1-2 Mbytes/second, while video information, which requires 10 Mbytes/second minimum bandwidth for real-time operation.

PCI offers a theoretical bandwidth of 132 Mbytes/second with an upgrade path 20 264 Mbytes/second defined, but not implemented. The limiting factors today are the *PCI*-to-CPU/memory interface controller.

With *ISA* bus, there are three ways of transferring data: polled I/O, *DMA*, and Bus Mastering.

Not all *PCI* frame grabber will allow you to transfer images across the bus at the full rate. *PCI* boards can operate in two modes. In the slave mode, the board responds to bus cycles initiated elsewhere, typically by the CPU. In the Master mode,, the board initiates transfers to another location, which might be the host memory, memory on a graphics card, or any other location that occupies an address on the *PCI* bus.

- Bus Master Benefits

A Bus Master frame grabber can transfer image to memory or other peripherals at real-time video rates. This makes memory or display capability on a frame grabber redundant and provides some overall system performance and cost advantages.

- Bus Mastering to PC System RAM

Nearly all image-processing applications involve execution of some sort of processing or analysis algorithms. The total time that it takes to complete an analysis is a function of two bottlenecks: how long it takes to digitize the image and get it to a processor and how fast the processor can execute the algorithms. In the past, the primary bottleneck has been the slow

ISA bus. *PCI* Bus Mastering now shifts the burden to the PC processors. Within 1/30th of a seeking an image is digitized into PC local memory. From there, the PC processor processes it as fast as it can.

Importantly, Bus Master transfers do not involve the PC CPU at all, so it is completely free to handle other tasks like processing images. This means that you can be acquiring one image while processing another. By creating two buffers and ping-ponging between them, it is possible to do continuous image processing with the host CPU, provided that it can keep up.

In most imaging applications, users want to view images and VGA graphics together on a single monitor. To do this in real time on the *ISA* bus required that the frame grabber either have display integrated onto it or that it have a scheme like a VGA feature connector to mix the video with the graphics.

With *PCI* Bus Mastering, it is possible to send digitized video directly from a frame grabber(one *PCI* device) to the memory of a VGA card (another *PCI* device). This option allows you to choose the display card you want with no limits on the graphics display size. Given the hardware path, you must select a way to support this software. A better option is to use a software standard like the Display Control Interface(DCI) under Windows. In Windows 95, an upgraded version of DCI called DirectDraw is the standard software mechanism to support Bus Mastered display on VGA cards. Functionally, it is similar to DCI, but it provides enhanced capabilities for overlays.

- Bus Mastering to access a Board

The most common places to Bus Master images from a frame grabber are PC system memory or a VGA display card. But any device on the *PCI* bus can be the target. This includes a high-speed processor board for applications that demand the highest possible processing rates. All that is required is an address to the memory buffer could be used as an intermediary between the frame grabber and processor board if the processor board does not have onboard memory.

The *IC-PCI* captures images from a single plug-in *AM*. The *IC-PCI* provides interface to a host computer through the *PCI* and *ISA* bus. The *IC-PCI* transfers image data to the PC for display.

3.3.2 IC-PCI System

The *IC -PCI* is a low cost frame grabber with a high-speed *PCI* bus interface. It uses the high data transfer rates of the *PCI* bus to eliminate the need for on-board processing or display circuitry. Image display and processing will be handled by the host resources . The *IC-PCI* reads data from the AM-FA at 40 MHz). Image memory is 4MB in linear format, which allows acquisition from a party of standard and non-standard cameras.

- *PCI* Interface

The *IC-PCI* is capable of bus mastering data from image memory directly to a destination. within the system (system memory or another *PCI* target, such as VGA display memory). The *IC-PCI* interface also supports target access to the registers, *AM* module, and image memory, *PCI* bus interrupts may be generated based on events occurring on the *IC- PCI*. The *IC-PCI* provides configuration registers required by the *PCI* specification which allow the board to be recognized on power-up for automatic system configuration.

In Bus Master Mode the *IC-PCI* takes control of the *PCI* bus and courses data and address for image memory transfers to the host system. The *IC-PCI* allows interlaced and non-interlaced AOIs(area of interest) of the stored image to be transferred to the host. Transfer rates up to 80MB per second (or higher) are attainable based on bus traffic, block size, and capabilities of destination device.

During access to the AM address space, *IC-PCI* registers, and image frame memory the board supports *PCI* Target mode access. Target access allows Random byte read/write operations to the image memory region.

- Image Memory

The *IC-PCI* is available with 4-MB image memory. The linear format employed does not impose restrictions on the size of images acquired and stored. The beginning of acquisition storage can vs programmed on 4-KB boundaries. Multiple images can be stored, but the hardware does not have restrictions to protect one image from being overwritten by another. If an image exceeds the 4 MB limit of memory, the memory address wraps back to zero and continues writing image data at the beginning of memory. The hardware has a sequential snap mode that acquires and stores up to eight sequential images.

3.3.3 Theory of Operation

3.3.3.1 Interface : The *IC-PCI* provides direct access to the host system *PCI* bus and can operate as both Bus Master and Target. In Bus a Master mode the *IC-PCI* supplies both data and addresses to the bus, with the image frame memory being the data source. Only Bus Master write is supported, As a target, the *IC-PCI* can be accessed by other Bus Masters in the host system. The *IC-PCI* supports target access to the *IC-PCI* registers, *AM*, and image frame memory, Both read and write to operations are supported in target mode.

Memory Access: The *IC-PCI* image memory is mapped as a 4-MB region in the host system memory region. During power-up, a base address to this region is assigned by the *PCI* bus host system and stored in the Base Address Three configuration register. The *IC-PCI* uses dual ported memory, providing separate access ports for host access and image acquisition from the *AM*. Each port can be accessed simultaneously.

The *IC-PCI* has a linear memory architecture that provides flexibility for incoming images as there is no boundary for X/Y size of acquired data (within size of the image frame).

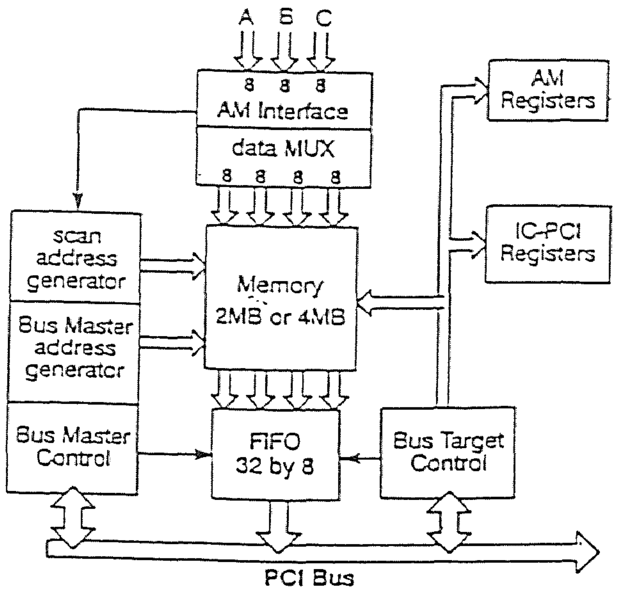

Figure 3.2 *IC-PCI* Block Diagram

AM Access: The *AM* is mapped into 8 KB of the host memory address region on power-up. The system defined a base address for the *AM* region and writes this base address into the base Address Two configuration register. Access to the *AM* space is defined as WORD (16-bit) only.

IC-PCI Register Access: The *IC-PCI* registers are organized as three groups, each having different control over the board operation. The *PCI* Configuration Registers are required by the *PCI* us specifications, and load *IC-PCI* dependent information on power-up. The *PCI* Interface registers provide control over the *PCI* bus interface. The Control Registers provide control over the *IC-PCI* memory, acquisition, and interrupts. All these sets of registers must be properly initialized for operation of *IC-PCI*.

PCI Configuration Registers: The *PCI* Configuration Registers are required by the *PCI* Specification in order to be fully compliant. These registers are loaded by a boot PROM with information about the *IC-PCI* which the *PCI* bus host uses in conjunction with information from all other *PCI* devices in the system to determine an optimum configuration.

PCI Interface Registers: The *PCI* interface registers control Bus transfer configuration, interrupt configuration, and mailbox operations. These registers require a separate I/O base address which is determined by the system on power-up and written back into Base Address Zero Register of the configuration register sets.

Control Registers: The Control Registers control Bus master transfers, interrupts, image acquisition, and *IC-PCI* initialization. These registers require a separate I/O base address which is determined by the system on power-up and written back into Base Address Zero Register of the configuration register sets.

3.3.3.2 Memory: The *IC-PCI* has a linear memory architecture that provides flexibility for incoming images as there is no boundary for X/Y size of acquired data. The upper byte of 24-bit image is always zeroed. The capacity linear memory, and interlaced addresses, must be calculated using four bytes per pixel for 24-bit full color images.

The *IC-PCI* receives interlaced or non-interlaced camera data from an *AM*. The *AM* data first passes through a multiplexed that will align the three *AM* data channels based on the pixel format. Pixel format must be programmed prior to image acquisition. Data rates from the *AM* modules are 40 MHz per 8-bit bus. Image sizes from the *AM* module can be any size (within size of frame memory and on 8-byte boundaries). Two types of image acquisition are available, External Trigger and Normal. Normal acquisition into the *IC-PCI* frame memory are based on *AM* frame timing. External trigger acquisitions are based on an eternal trigger signal from the *AM*.

Normal Acquisition: The beginning and ending of a normal acquisition is based on the framing signals FEN (frame enable). FEN is the digitized version of the Vertical Blank signals. Three modes of acquisition exist: snap, grab, and freeze. A snap command performs a single frame acquire beginning on the next FEN rising edge, A freeze command will stop a live acquire at the next FEN rising edge, A freeze command will stop a live acquire at the next failing edge of FEN.

External Trigger Acquisition: External Trigger acquisition allows image acquires to be synchronized to external events, When acquiring an image in external mode, the acquisition will not start until the AM provides a trigger signal to the *IC-PCI*(FEN does not determine when the acquisition starts or stops).

Image Acquisition into Frame Memory: Images from the *AM* module are stored linearly into the *IC-PCI* frame memory. The image memory is 4KB wide by 512 rows or 1024 rows. Images may be landed anywhere within the *IC-PCI* memory(on 4096 byte boundaries).

Non-Interlaced Image Acquisition: An example of how a 1K x 1Kx8 bit non-interlaced image would land in *IC-PCI* memory is known here: an acquire with AQSTART set to zero.

Interlaced Image Acquisition: Interlaced images are acquired into *IC-PCI* image memory sequentially, meaning each field is acquired and stored one after the other. An example of how a 1K x 1Kx8 bit interlaced image would land in *IC-PCI* memory is known here: an acquire with AQSTART set to zero.

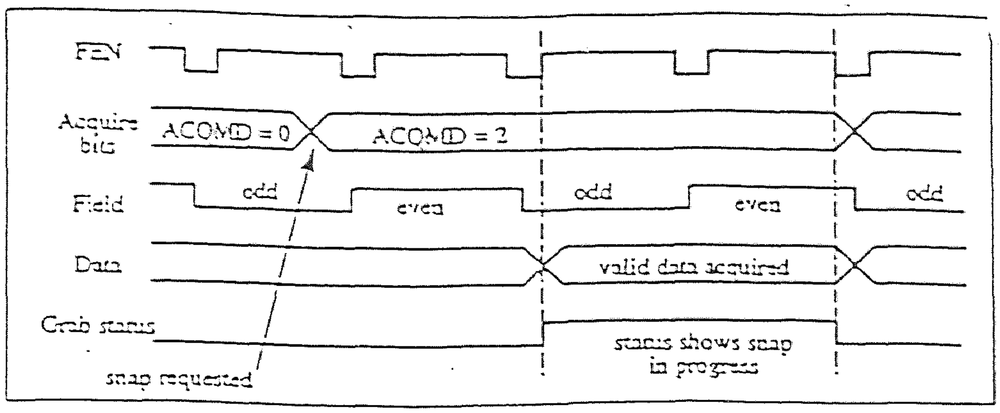

Figure 3.3 Interlaced Accquire

Acquiring into Linear Memory: One benefit of linear memory is that the acquired image size can be set to any value within the size of the image frame, which means it can be programmed on *AM*. If during an acquisition, the incoming line size crosses the row address boundary, the line simple continues to be written into memory starting on the next row address. An example using 640 by 480 by 8bit image acquire shown her to demonstrates these feather.

The *IC-PCI* has the ability to acquire and store up to eight images sequentially into image memory with the execution of one snap command.

3.3.3.3 Bus Master Operation: A bus master operation is when the *IC-PCI* is granted control of the *PCI* bus and becomes the source of both address and data to some other target location within the system. A set sequence of register operations must be performed to initialize and setup the *IC-PCI* for every bus master transfer. The size and start location of the data to be transferred within image frame memory is programmable, allowing *AOI*s to be sent to a *PCI* bus Target. Interlaced video images are stored sequentially in *IC-PCI* memory when acquired. In bus master mode, the *IC-PCI* provides hardware capability to transfer an image to the host with the even and odd fields assembled or interleaved into one complete image in host memory. This feature greatly reduces software overhead when transferring interlaced video images.

Register sequence for bus master transfer:

- Define the *AOI* size (SOURCE)
- Define the Target (Destination)
- Clear the FIFO
- Enable *IC-PCI* Bus Master Mode
- Request the *PCI* bus

Bus master transfer completion:

- Disable previous Transfer
- Disable the previous Bus Request
- Redefine the *AOI* Size (SOURCE)
- Redefined the Target (Destination)
- Clear the FIFO
- Re-enable *IC-PCI* Bus Master Mode
- Re-request the *PCI* Bus

3.3.3.4 Bus Master Latency: As shown before, each new bus master transfer requiems a minimum of 7 separate register writes to restart a bus master transfer at the end of a previous transfer. Transfer speed will be mainly depend on the host system and the targets capabilities. Bus traffic(other bus masters arbitraging for and gaining bus ownership) and target capabilities (the *IC-PCI* can only transfer data as fast as the target can receive it) will have implications on overall transfer speed).

The *IC-PCI* provides the capability to generate a *PCI* bus based on various events. Interrupts can be broken down into three separate categories: Acquisition interrupts, Bus Master Interrupts, and *PCI* interface interrupts.

3.3.4 IC-PCI Software

ITEX is the generic term for Imaging Technology software functions, callable and linkable with C language application programs ITEX-IC refers to the aggregate sum of sub-libraries or software modules for *IC-PCI* modules.

An MVC mother board is a hardware component that connects to the system back-plane, classics, or computer mother board. A module or daughter board is a hardware component that connects to a mother board. Mother boards and daughter boards both use the data structure MODCNF, describe below. From software point of view, there is no difference between a board and a module.

- System Data Structures

ITEX-IC uses four major data structures. The file /itexroot/common/include/itxsys/ module.h contains the details of the structures. The structures are:

1. MODCNF
2. MODCTL
3. MODREG
4. BOARDMOD

- System Configuration

The following are new functions provided for making program code .

Table 3.1 Common API Functions

Function	*Description*
itx_acqbits	Set or return acquisiton command bits
itx_clr_area	Clear an area in frame memory
itx_clr_frame	Clear an entire frame memory
itx_freeze	Stop acquisition in frame memory
itx_grab	Acquire continuously in frame memory
itx_read_area	Read from area in frame memory
itx_snap	Acquire single image in frame memory
itx_wait_acq	Wait for acquisiton to complete
itx_wait_vb	Wait for vertical blank
itx_write_area	Write to area in frame memory

The High-Level ACQUISITION API consists of three major functional areas:

- Frame Buffer to Host Ping-Pong Buffers
- Frame Buffer to Sequential Host Frames (called VCR record)
- Large Frame Acquisition (for frames large than the host frame)

All three of these functional areas make use of frames that are interrupt-driven and bus-mastered from the frame buffer to host memory.

The Ping-Pong function allows continuous transfer of frames to alternating ping and pong frames to occur in the background with minimal CPU overhead.

3.4 Application Program

According to the real procedure we use liquid crystal to get retarded images as we grab in image data from camera to PC memory. The key problem is to control the exact time of grabbing and speed up the grabbing process as quick as we can.

We improved the application program by modified the function collect1A and it's relating functions. The idea is to find the unneeded waiting time. At this stage, we get 23 images per second which is close to theoretical 30 /second. The further work will continuing to make approach.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Gain High Resolution

Synthesizing a high resolution solar image by merging several low resolution Ones The powerful processing and optimization techniques enable to improve the geometric resolution of observation solar images by merging several images of the same area.

The methodology that P. Leborgne developed to deal with this problem is to use accurate geometric and radiometric models of the image capture process. The interpolation of the final higher resolution image is performed by minimizing in the Fourier space a criterion leading to an optimal reconstruction of the spectrum, in terms of aliening and bandwidth.

Results will be presented reveal new image details that were distinguishable neither in original nor in monocularly deconvolved images.

Further research will be hinted, applying the wavelet theory to design a solution more robust to model noise.

4.2 Future Improvements

Some of the factors to consider when designing a PCI bus-based image application system.

- Transfer Speed

The *PCI* bus has a theoretical transfer rate of 132 MB/s (33 MHz x 32-bit width). But actual benchmarks reported in the field range from under 10 to 120 MB/s, depending on the details of the memory architecture and bus interface design.

- Interface Design

PCI frame grabbers with DMA interfaces allow the image subsystem to be decoupled from the host. Image acquisition, image transfer, and image processing can be pipelined. If a *PCI* frame grabber has on-board memory to buffer two frames of data, then while one frame is being processed by the host, a second frame can be *DMA* transferred to system memory, while a third frame is simultaneously acquired into frame grabber memory.

- On-Board Memory

Many low cost designs have no on-board memory and rely instead on fast transfer of data to host memory. This saves on component costs, but impacts performance. One can never be sure that the *PCI* bus will be available exactly when it is needed; data could be lost. Without on-board memory it is almost impossible to use external triggers,. a key machine vision enabler. And without memory, it is not possible to create the pipeline described above.

- Memory Architecture

Images are often captured and stored as fields, but processed as frames. A board designed for machine vision should de-interface data automatically when transferring images or *AOIs* to the host.

- Integrated Designs

The *PCI* bus, with its high speed capabilities, will revolutionize the use of PCs in a variety of applications. The reason to use a *PCI* bus framegrabber, as opposed to an ISA or EISA frame grabber, is to take advantage of the features of the *PCI* bus that are not available across all the other busses. Some of these features are speed, standardization and plug and play. As companies involved in vision processing look to *PCI* bus frame grabbers, there are several items which differentiate the available vision products based on these core criteria.

- SPEED--132 MB/s, Fact and Fiction

The *PCI* bus has a theoretical bandwidth capacity of 132 MB/s. This number is derived from the bus running at 33 MHz and being 4 bytes wide. However, there is a difference between sustained and theoretical rates. The 132 MB/s is the theoretical rate. The question to ask is

"What is the sustained rate onto the *PCI* bus?" Believe it or not, there are several PCI frame grabbers that can only move data at 15 MB/s (or less). Usually, this is due to the fact that they do not move data in "burst" mode, but sometimes the memory architecture is at fault, or the PCI/memory controller is not optimal.

- *DMA*

The *PCI* bus decouples data transfer devices from the CPU and its memory. Therefore, a PCI device that can become a Bus Master with *DMA* is highly advantageous because it allows for simultaneous data transfer and host processing! By allowing the frame grabber to become a *PCI* bus master and to quickly *DMA* the video frame to host memory, the host CPU is free to process the previously transferred frame of data. Without Bus Master and *DMA*, your high performance Pentium will be forced to perform the entire transfer...forcing the Pentium to become an expensive address generator!

- Linear Memory

For most frame grabbers that have linear memory, there are several problems posed by transferring images out of frame grabber memory to host memory. Consider a 512x512 pixel RS170 signal. Lets say that the frame grabber can put this data starting at memory location. Because it is interleaved, the first field will take up the first 512x256 (128K)bytes. The next field will start at location 128K+1.

Now when we want to transfer this data out to a VGA card or into host memory to process, we would get 2 consecutive fields, not the correct interlaced data. If the frame grabber is not a *DMA* bus master, this is of little consequence since the host CPU is acting as the address generator, and it will just take out the lines in the proper fashion.

On the other hand, if the frame grabber can *DMA* the data, it should ideally do the whole image, or as large a *DMA* block as possible. If the board has to *DMA* only a line at a time

(512 bytes), it will be very inefficient, since the *DMA* controller will have to set itself up as much as it is transferring. If the frame grabber has circuitry to interlace the data on the fly during the DMA transfer (in other words, generate the offset by itself), the user would get a very efficient transfer, since the whole image can be transferred in a single *DMA* operation.

The same situation is also true for the user who wants to transfer an Area of Interest *(AOI)* from the frame grabbers' memory to the *PCI* bus. In this case, the number of pixels will be small. If the *DMA* could only transfer a line at a time,
the overhead of setting up the *DMA* may be larger than the actual transfer time. For instance, the user wants to transfer a 64x64 block of pixels from the middle of the image.

This is only a 4K transfer, but it would be terribly inefficient to do a *DMA* of 64 bytes at a time vs. doing a single 4K transfer.

CHAPTER 5

CONCULSIONS

In this paper, we have presented an improved image processing system, which is used in solar observation of NJIT. The improvement is based on both hardware design and software application program side. In our experiments, the main achievement is to speed up the image grabbing process, and get enough time to control the liquid crystal reversing procedure. It is successful. The system is currently running well at Big Bear Solar Observation of NJIT, CA.

APPENDIX

PROGRAMS

This appendix contains all of the programs modified in speeding up frame-grab rate.

```
///////////////////////////////////////////////////////////////////////////
// Collect1A: Primary acquisition function which captures and processes single and multiple
images from
//                  the ICP board.

int CIML::Collect1A (CImageData *pCollectParams, int Mode)
{
          if (NeedImgTrans())
                    return Collect1B(pCollectParams, Mode);

          int Ret = Config();
          if (Ret)
                    return Ret;

          int BufPad = 1;
          int c, i;
          short *pStackBuf;
   static ACols = pCollectParams->Columns();
   static ARows = pCollectParams->Rows();
   static OrigCol = pCollectParams->OrigCol();
   static OrigRow = pCollectParams->OrigRow();
          static Count = pCollectParams->Avgs();
          static ColorPlanes = pCollectParams->ColorPlanes();

          // Keep track of when we need to bit-shift.
          int BitShift = m_PixBitShift;
          // State variables for live display mode and preparation success.
          static int DispOn = FALSE;
          static PrepDone = FALSE;
          // Acquisition buffers.
          static short *pOneFrame[3];
          static short *pR2Frame = NULL;
          // Temp buffers allocated?
          static int TmpBuf = FALSE;
          // Tap offsets
          static long TapOffs[4];

          // Preparation steps:
          if (Mode & PREP)
          {
                    if (PrepDone)
                    {
                              // If doing a non-stop acquire, stop it first.
                              if (ImapApp.ContDisp())
                              {
                                        ImapApp.ContDisp(FALSE);
                                        CollectCont();
```

```
				}
				else	// If an ongoing acquire is in place, then we can't continue.
						return -1;
			}
			PrepDone = FALSE;

	// Do we need to do a flat-field calibration?
			if (pCollectParams->Flatten())
			if (DoCalFrames() != -10)
						if (m_ValidCal != CAL_VALID)
						{
								pCollectParams->Flatten(FALSE);
								AfxMessageBox("Calibration being omitted. Prepare to
acquire image.");
						}

			ACols = pCollectParams->Columns();
			ARows = pCollectParams->Rows();
			OrigCol = pCollectParams->OrigCol();
			// JPL Cam is flipped in X
			if (IsCamera(CamJPL))
				OrigCol = short(m_camx - (OrigCol + ACols));
			OrigRow = pCollectParams->OrigRow();
			Count = pCollectParams->Avgs();
			ColorPlanes = pCollectParams->ColorPlanes();

			// Make sure Bus-Master mode is ON.
			if (ISICP(m_imlmod))
				g_ITEX.icp_bm_mode(m_imlmod, ICP_ENABLED);
			else
				g_ITEX.imp_bm_mode(m_imlmod, IMP_ENABLE);
			if (m_imlmod2)
			{
				if (ISICP(m_imlmod2))
						g_ITEX.icp_bm_mode(m_imlmod2, ICP_ENABLED);
				else
						g_ITEX.imp_bm_mode(m_imlmod2, IMP_ENABLE);
			}

			// If we came from display mode (most likely 8-bit wide frames)
			// then we need to reconfigure frames and input LUT.
			DispOn = m_DispOn;
			if (m_DispOn)
			{
				m_DispOn = FALSE;
				InitFrame();
			}
			MapLUTs (12, 12);

			// Configure correct pix depth and bit shifting setting.
			pCollectParams->PixBitsUsed(m_BitsPPixUsed);

			for (c=0; c < ColorPlanes; c++)
				pOneFrame[c] = NULL;

			for (c=0; c < ColorPlanes; c++)
```

```
		{
			long *pImageBuf;
			pImageBuf = (long *) pCollectParams->GetImgBuffer(0,
MapColorOrder(pCollectParams,c));
			if (pImageBuf == NULL)
			{
				for (i=0; i<c; i++)
					if (TmpBuf)
						free (pOneFrame[i]);
				return IDS_ERR_NULL_BUF;
			}

			// If we are averaging or summing frames, we need to allocate a separate
			// buffer to keep each incoming image,
			if (!pCollectParams->IsFrameFormat(CImageData::Stacked))
			{
				if (pCollectParams->Avgs() > 1)
				{
				pOneFrame[c] = (short *) malloc
(long(ACols)*(ARows+BufPad)*m_BitsPPix/8);
					TmpBuf = TRUE;
					if (pOneFrame[c] == NULL)
					{
						AfxMessageBox("Not Enough Memory for
Allocations");

						return IDS_ERR_ALLOC_BUF;
					}
				}
				// Otherwise just use already allocated buffer.
				else
				{
					long BufOffset = 0;
					if (pCollectParams->PixBytes() > m_BitsPPix/8)
						BufOffset = (pCollectParams-
>PixBytes()/(m_BitsPPix/8))*pCollectParams->NFrmPixs();
				pOneFrame[c] = (short *) (((UCHAR *)pImageBuf) + BufOffset);
				}

				// We need to lock the buffers before use
				if (m_am_frame >= 0)
					g_ITEX.itx_read_area(m_imlmod, m_am_frame,
short(OrigCol), short(OrigRow),
								short(ACols), short(ARows), (DWORD
*)pOneFrame[c]);
				else
					icpcj_LFormat_Prep(pCollectParams, pOneFrame[c]);
			}
			else	// For stacks, just lock the buffers
			{
				for (i=0; i< Count; i++)
				{
					pStackBuf = (short *)pCollectParams->GetImgBuffer(i,
c);

					if (m_am_frame >= 0)
						g_ITEX.itx_read_area(m_imlmod, m_am_frame,
short(OrigCol), short(OrigRow),
                                                                                short(ACols),
short(ARows), (DWORD  *)pStackBuf);
                                        else
                                                icpcj_LFormat_Prep(pCollectParams,
pStackBuf);
                                }
                        }
                }

        // If doing image subtractions, allocate second buffer
                if (pCollectParams->IMode() == R1MR2 AND ColorPlanes == 1)
                {
                        pR2Frame = (short *) malloc
(long(ACols)*(ARows+BufPad)*m_BitsPPix/8);
                        if (pR2Frame == NULL)
                        {
                                if (TmpBuf)
                                        for (int c=0; c < ColorPlanes; c++)
                                                free (pOneFrame[c]);

                                AfxMessageBox("Not Enough Memory for Allocations");
                                return IDS_ERR_ALLOC_BUF;
                        }
                        // We need to lock the buffer before use
                        if (m_am_frame >= 0)
                                g_ITEX.itx_read_area(m_imlmod, m_am_frame, short(OrigCol),
short(OrigRow),
                                                        short(ACols), short(ARows), (DWORD
*)pR2Frame);
                        else
                                icpcj_LFormat_Prep(pCollectParams, pR2Frame);
                }

                // Strobe time in msecs
                ULONG ulStrobeTime = ULONG(1000.0/((pCollectParams->StrobeFreq() > 0) ?
pCollectParams->StrobeFreq() : 3));
                if (ulStrobeTime > 30000) ulStrobeTime = 30000;

                // MODIFY - When do we need to generate external signal? Should we use the
AM's ExSync signal?
                // Generate a frame pulse just for setup.
                if (ESyncOn() & PCESYNC)
                {
                        CamIntegrate(100);
                        g_ITEX.itx_set_timeout(m_ammod, ulStrobeTime);
                }
                // We go into grab mode, which is needed when capturing multiple
                // consecutive frames.
                if (ISICP(m_imlmod))
                {
                        //g_ITEX.icp_put_acq_start_addr(m_imlmod, 0);
                        g_ITEX.icp_pixsz(m_imlmod,short((m_BitsPPix == 8) ? ICP_PIX8 :
ICP_PIX16));
                        g_ITEX.write_reg(m_imlmod,ICP_ACQMD, ICP_GRAB);
                }
                else
```

```
				{
						g_ITEX.write_reg(m_imlmod,IMP_A0AQ, IMP_GRAB);
						g_ITEX.write_reg(m_imlmod,IMP_A1AQ, IMP_GRAB);
				}

				// Let's calculate section offsets for the color camera.
				if (ColorPlanes > 1)
						GetTapOffsets(pCollectParams, TapOffs);

				// Announce acquisition has begun
				ImapApp.StatBarMsg("Acquisition in Process", StatBarMsgAcq);
				ImapApp.StatBarUpdate();

				// Ready to Acquire
				PrepDone = TRUE;
		}

		if (!PrepDone)
				return -1;

// FOR SPEED TESTING
static short Lines1[28];
int bQuit = FALSE;

if (Mode & ACQUIRE)
{
		SYSTEMTIME TimeStamp;
		// Save camera info
		CString sCamInfo;
		char *szCamInfo = sCamInfo.GetBuffer(100);
		g_SMDCAMS.SMD_GetCameraInfo(szCamInfo, 99, SMD_INFOSETTINGS1, -1);
		sCamInfo.ReleaseBuffer();
		sCamInfo = GetCamName() + ": " + sCamInfo;
		pCollectParams->SetCameraInfo(sCamInfo);

		// Integration time in msecs
		ULONG ulITime = (ULONG)(1000*pCollectParams->ITime());
		if (ulITime > 20000) ulITime = 20000;
		// If not integrating, the pulse width should be minimal
		if (!(ESyncOn() & EINTG))
				ulITime = 5;

		// Strobe time in msecs
		ULONG ulStrobeTime = ULONG(1000.0/((pCollectParams->StrobeFreq() > 0) ?
pCollectParams->StrobeFreq() : 3));
		if (ulStrobeTime > 30000) ulStrobeTime = 30000;
		// Does integration wait or executes asynchronously
		if (CamIntegrate(5) == 0)
				ulStrobeTime = ULONG(MAX(1, long(ulStrobeTime) - long(ulITime)));

		// Wait until line LineWait to readout a frame.
		int LineWait = 40;
		if (long(ACols)*ARows < 256*256)
				LineWait = 10;

		// Timeout period for acquire.
```

```
        long AcqTimeOut = 9000;
        if (ESyncOn() AND !(ESyncOn() & PCESYNC))
                AcqTimeOut = 0;

        // Send out the first pulse.
        if (ESyncOn() & PCESYNC)
        {
                // Complete any prior timeout periods.
        while (!g_ITEX.itx_get_timeout(m_ammod));
                CamIntegrate(ulITime);
                g_ITEX.itx_set_timeout(m_ammod, ulStrobeTime);
        }
        else if (!ESyncOn())
        {
                // We need a small delay to wait for grab mode to start.
                g_ITEX.itx_delay(20);
        }

        if (pCollectParams->IsFrameFormat(CImageData::Stacked))
        {
                // Make sure we have highest priority for Bus-Master transfers.
                SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);

                // Here we acquire a stack of consecutive frames.
                for (i=0; i<Count; i++)
                {
                        for (c=0; c < ColorPlanes; c++)
                        {
                                BitShift = m_PixBitShift;
                                // Color images come in groups of three.
                                pStackBuf = (short  *)pCollectParams->GetImgBuffer(i,
MapColorOrder(pCollectParams,c));
// SPEED TESTING
Lines1[(i MOD 3)*9 + c*3] = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -MTDREGOFFSET);
                                if (m_Camera == Cam1M60_10C OR m_Camera ==
Cam1M60_20C)
                                        if (ColorPlanes > 1 AND c MOD ColorPlanes == 0)
                                        {
                                                g_ITEX.ammtd_exsyncp(m_ammod,
AMMTD_EXSYNHI);
                                                // Need to skip 1st frame, which is invalid.
                                                bQuit = Wait4Line(short(m_camy -2), 1,
AcqTimeOut);
                                                if (bQuit < 0) break;
                                                bQuit = Wait4Line(short(LineWait), -1,
AcqTimeOut);
                                                if (bQuit < 0) break;
                                        }

                                // MODIFY - When do we need to generate external signal?
Should we use the AM's ExSync signal?
                                if (ESyncOn() & PCESYNC)
                                {
                                        // Complete any prior timeout periods.
                                        while (!g_ITEX.itx_get_timeout(m_ammod));
                                        CamIntegrate(ullTime);
                                        g_ITEX.itx_set_timeout(m_ammod, ulStrobeTime);
                                }

                                if (m_am_frame >= 0)
                                {
                                        // Normal image capture. Wait for the proper time in the
acquisition.
                                        bQuit = Wait4Line(short(LineWait), 1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        bQuit = Wait4Line(short(LineWait), -1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        // Place time stamp at start of acquire.
                                        GetLocalTime(&TimeStamp);
                                        pCollectParams->SetImgStartAcquireTime(TimeStamp,
i, MapColorOrder(pCollectParams,c));
// SPEED TESTING
Lines1[(i MOD 3)*9 +1 + c*3] = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -
MTDREGOFFSET);
                                        g_ITEX.itx_read_area(m_imlmod, m_am_frame,
short(OrigCol), short(OrigRow), short(ACols), short(ARows), (DWORD  *)pStackBuf);
// SPEED TESTING
Lines1[(i MOD 3)*9 +2 + c*3] = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -
MTDREGOFFSET);
                                }
                                else
                                {
                                        // Wait for acquire to begin
                                        bQuit = Wait4Line(short(m_camy), -1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        // Place time stamp at start of acquire.
                                        GetLocalTime(&TimeStamp);
                                        pCollectParams->SetImgStartAcquireTime(TimeStamp,
i, MapColorOrder(pCollectParams,c));
                                        // Call a special function to deal with images which are
                                        // larger than the ICP memory space.
                                        icpcj_LFormat_acquire(pCollectParams, pStackBuf);
                                }
                                if (m_Camera == Cam1M60_10C OR m_Camera ==
Cam1M60_20C)
                                        if (ColorPlanes > 1 AND c MOD ColorPlanes ==
ColorPlanes -1)
                                        {
                                                g_ITEX.ammtd_exsyncp(m_ammod,
AMMTD_EXSYNLO);
                                                // Wait for the filter to come to full rest.
                                                Sleep(40);
                                        }
                        }
                        // Check if we need to quit
                        if (bQuit < 0) break;

                        // Here we acquire a frame for subtraction mode
                        // Not applicable to color.
                        if (pR2Frame)
                        {
```

```
					// MODIFY - When do we need to generate external signal?
Should we use the AM's ExSync signal?
					if (ESyncOn() & PCESYNC)
					{
						// Complete any prior timeout periods.
						while (!g_ITEX.itx_get_timeout(m_ammod));
						CamIntegrate(ulITime);
						g_ITEX.itx_set_timeout(m_ammod, ulStrobeTime);
					}
				if (m_am_frame >= 0)
					{
						bQuit = Wait4Line(short(LineWait), 1, AcqTimeOut);
						if (bQuit < 0) break;
						bQuit = Wait4Line(short(LineWait), -1, AcqTimeOut);
						if (bQuit < 0) break;
						g_ITEX.itx_read_area(m_imlmod, m_am_frame,
short(OrigCol), short(OrigRow), short(ACols), short(ARows), (DWORD  *)pR2Frame);
					}
					else
					{
						// Wait for acquire to begin
						bQuit = Wait4Line(short(m_camy), -1, AcqTimeOut);
						if (bQuit < 0) break;
						icpcj_LFormat_acquire(pCollectParams, pR2Frame);
					}
					// Image subtraction.
					if (m_BitsPPix == 16)
						pCollectParams->Add2Image(i, (USHORT *)pStackBuf,
BitShift, (USHORT *)pR2Frame);
					else
						pCollectParams->Add2Image(i, (BYTE *)pStackBuf,
BitShift, (BYTE *)pR2Frame);
				}
				// Place time stamp at end of acquire.
				GetLocalTime(&TimeStamp);
				pCollectParams->SetImgEndAcquireTime(TimeStamp, i,
MapColorOrder(pCollectParams,c));
			} // end collecting stack.
			// Set priority back to normal.
			SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS);

			// If R1MR2, then no further processing is needed.
			if (!pR2Frame AND bQuit)
			{
				for (i=0; i<pCollectParams->Avgs(); i++)
				{
					pOneFrame[0] = (short  *)pCollectParams->GetImgBuffer(i,
MapColorOrder(pCollectParams,0));
					pOneFrame[1] = (short  *)pCollectParams->GetImgBuffer(i,
MapColorOrder(pCollectParams,1));
					pOneFrame[2] = (short  *)pCollectParams->GetImgBuffer(i,
MapColorOrder(pCollectParams,2));
					// Flat-field or correct for offsets or just do bit shifting.
				if (pCollectParams->Flatten())
					Calibrate(pCollectParams, pOneFrame, BitShift);
```

```
                                else if (IsCamera(Cam1M60_10C) OR
IsCamera(Cam1M60_20C))
                                NormalizeTaps(pCollectParams, pOneFrame, TapOffs, BitShift);
                                else if (BitShift)
                                        for (c=0; c < ColorPlanes; c++)
                                        {
                                                if (m_BitsPPix == 16)
                                                        pCollectParams->Add2Image(i,
MapColorOrder(pCollectParams,c), (USHORT *)pOneFrame[c], BitShift);
                                                else
                                                        pCollectParams->Add2Image(i,
MapColorOrder(pCollectParams,c), (BYTE *)pOneFrame[c], BitShift);
                                        }
                        }
                }
        }
        else
        {
                // Make sure we have highest priority for Bus-Master transfers.
                SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);

                for (i=0; i < Count; i++)
                {
                        // Color images come in groups
                        for (c=0; c < ColorPlanes; c++)
                        {
                                BitShift = m_PixBitShift;
// SPEED TESTING
Lines1[0 + c*3] = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -MTDREGOFFSET);
                                if (m_Camera == Cam1M60_10C OR m_Camera ==
Cam1M60_20C)
                                        if (ColorPlanes > 1 AND c MOD ColorPlanes == 0)
                                        {
                                                g_ITEX.ammtd_exsyncp(m_ammod,
AMMTD_EXSYNHI);
                                                // Wait to finish with any ongoing frame.
                                                bQuit = Wait4Line(short(m_camy -2), 1,
AcqTimeOut);
                                                if (bQuit < 0) break;
                                                bQuit = Wait4Line(short(LineWait), -1,
AcqTimeOut);
                                                if (bQuit < 0) break;
                                        }

                                // MODIFY - When do we need to generate external signal?
Should we use the AM's ExSync signal?
                                if (ESyncOn() & PCESYNC)
                                {
                                        // Complete any prior timeout periods.
                                        while (!g_ITEX.itx_get_timeout(m_ammod));

                                        CamIntegrate(ulITime);
                                        g_ITEX.itx_set_timeout(m_ammod, ulStrobeTime);
                                }
                        if (m_am_frame >= 0)
                                {
                                        // Normal image capture. Wait for the proper time in the
acquisition.
                                        bQuit = Wait4Line(short(LineWait), 1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        bQuit = Wait4Line(short(LineWait), -1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        // Place time stamp at start of acquire.
                                        GetLocalTime(&TimeStamp);
                                        pCollectParams->SetImgStartAcquireTime(TimeStamp,
0, MapColorOrder(pCollectParams,c));
// SPEED TESTING
Lines1[1 + c*3] = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -MTDREGOFFSET);
                                        g_ITEX.itx_read_area(m_imlmod, m_am_frame,
short(OrigCol), short(OrigRow), short(ACols), short(ARows), (DWORD  *)pOneFrame[c]);

// SPEED TESTING
Lines1[2 + c*3] = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -MTDREGOFFSET);
                                }
                                else
                                {
                                        // Wait for acquire to begin
                                        bQuit = Wait4Line(short(m_camy), -1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        // Place time stamp at start of acquire.
                                        GetLocalTime(&TimeStamp);
                                        pCollectParams->SetImgStartAcquireTime(TimeStamp,
0, MapColorOrder(pCollectParams,c));
                                        // Call a special function to deal with images which are
                                        // larger than the ICP memory space.
                                        icpcj_LFormat_acquire(pCollectParams, pOneFrame[c]);
                                }
                                if (m_Camera == Cam1M60_10C OR m_Camera ==
Cam1M60_20C)
                                        if (ColorPlanes > 1 AND c MOD ColorPlanes ==
ColorPlanes -1)
                                        {
                                                g_ITEX.ammtd_exsyncp(m_ammod,
AMMTD_EXSYNLO);
                                                // Wait for the filter to come to full rest.
                                                Sleep(40);
                                        }
                        }
                        // Check if we need to quit
                        if (bQuit < 0) break;

                        // Here we acquire a frame for subtraction mode
                        // Not applicable to color.
                        if (pR2Frame)
                        {
                                // MODIFY - When do we need to generate external signal?
Should we use the AM's ExSync signal?
                                if (ESyncOn() & PCESYNC)
                                {
                                        // Complete any prior timeout periods.
                                        while (!g_ITEX.itx_get_timeout(m_ammod));
                                        CamIntegrate(ullTime);
                                        g_ITEX.itx_set_timeout(m_ammod, ulStrobeTime);
                                }
                                if (m_am_frame >= 0)
                                {
                                        bQuit = Wait4Line(short(LineWait), 1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        bQuit = Wait4Line(short(LineWait), -1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        g_ITEX.itx_read_area(m_imlmod, m_am_frame,
short(OrigCol), short(OrigRow), short(ACols), short(ARows), (DWORD  *)pR2Frame);
                                }
                                else
                                {
                                        // Wait for acquire to begin
                                        bQuit = Wait4Line(short(m_camy), -1, AcqTimeOut);
                                        if (bQuit < 0) break;
                                        icpcj_LFormat_acquire(pCollectParams, pR2Frame);
                                }
                        }
                        // Flat-field or correct for offsets.
                else if (pCollectParams->Flatten())
                        {
                        Calibrate(pCollectParams, pOneFrame, BitShift);
                                // Bit-shifting is done.
                                BitShift = 0;
                        }
                        else if (IsCamera(Cam1M60_10C) OR IsCamera(Cam1M60_20C))
                        {
                        NormalizeTaps(pCollectParams, pOneFrame, TapOffs, BitShift);
                                // Bit-shifting is done.
                                BitShift = 0;
                        }

                        // Image accumulation.
                        for (c=0; c < ColorPlanes; c++)
                        {
                                if (m_BitsPPix == 16)
                                        pCollectParams->Add2Image(0,
MapColorOrder(pCollectParams,c), (USHORT *)pOneFrame[c], BitShift, (USHORT *)pR2Frame);
                                else
                                        pCollectParams->Add2Image(0,
MapColorOrder(pCollectParams,c), (BYTE *)pOneFrame[c], BitShift, (BYTE *)pR2Frame);
                        }
                        // Place time stamp at end of acquire.
                        GetLocalTime(&TimeStamp);
                        pCollectParams->SetImgEndAcquireTime(TimeStamp, 0,
MapColorOrder(pCollectParams,c));
                } // end collecting averages.
                // Set priority back to normal.
                SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS);
        }
        if (bQuit < 0)
                ImapApp.OnStopAcquire();
}

if (Mode & DOSTATS)
```

```
{
        // Take care of any statistics calculations.
        pCollectParams->AvgNStats();
        // JPLCam image needs to be flipped in X.
        if (IsCamera(CamJPL))
        {
                CWhichBuffer Buf(pCollectParams, CWhichBuffer::ImageBuffer, 0);
                CImageRegion ROI(Buf, CImageRegion::NoDIB);
                ROI.SetRegionBox(0, 0, pCollectParams->Columns(), pCollectParams->Rows());
                // Flip in X, full stack if necessary
                ROI.RegionFlip(1, true);
                // Flip any stats buffers
                if (pCollectParams->NStatBufs() > 0)
                {
                        CWhichBuffer Buf(pCollectParams, CWhichBuffer::StatBuffer, 0);
                        CImageRegion ROI(Buf, CImageRegion::NoDIB);
                        ROI.SetRegionBox(0, 0, pCollectParams->Columns(), pCollectParams-
>Rows());
                        // Flip in X, full stack if necessary
                        ROI.RegionFlip(1);
                        if (pCollectParams->NStatBufs() > 1)
                        {
                                Buf.SetBufIndex(1);
                                CImageRegion ROI(Buf, CImageRegion::NoDIB);
                                ROI.SetRegionBox(0, 0, pCollectParams->Columns(),
pCollectParams->Rows());
                                // Flip in X, full stack if necessary
                                ROI.RegionFlip(1);
                        }
                }
        }
}

if (Mode & CLEANUP)
{
        if (TmpBuf)
                for (int c=0; c < ColorPlanes; c++)
                {
                        free (pOneFrame[c]);
                        pOneFrame[c] = NULL;
                }
        TmpBuf = FALSE;

        if (pR2Frame != NULL)
        {
                free (pR2Frame);
                pR2Frame = NULL;
        }

        // Announce acquisition is complete
        ImapApp.StatBarMsg("Acquisition Done", StatBarMsgAcq);


        if (DispOn)
                Continuous();
        else if (ISIMP(m_imlmod))
```

```
            ResetLiveDisp();

        PrepDone = FALSE;

        // TESTING - 5 lines
        char LineMsg[200];
        sprintf(LineMsg, "Readout: %d, %d, %d, %d, %d, %d, %d, %d, %d. \n",
            Lines1[0], Lines1[1], Lines1[2], Lines1[3], Lines1[4], Lines1[5],
            Lines1[6], Lines1[7], Lines1[8]);
        g_ITEX.itx_display(LineMsg);
    }
        if (bQuit < 0)
            return bQuit;
        else
            return FALSE;
    }
```

```
int CIML::Wait4Line(short Line, int Pos, long TimeOut, int ChangePriority)
{
	static long Count = 0;
	short LineNow;
	// Start another timer to check for ESC keys.
	static ULONG m_ulStartTicks = GetTickCount();

	if (TimeOut > 0)
		g_ITEX.itx_set_timeout(m_imlmod, TimeOut);
	while ((TimeOut > 0) ? !g_ITEX.itx_get_timeout(m_imlmod) : TRUE)
	{
		if (m_IsAMDIG)
			LineNow = am_lstat(m_ammod);
		else
			LineNow = g_ITEX.read_reg(m_ammod, AMMTD_LSTAT -
MTDREGOFFSET);
			// Below is for getting the line number from the ICP board, which is not as
accurate.
			//LineNow = (g_ITEX.icp_acq_addr_status(m_imlmod)
+1)*(4096/m_camx)/(m_BitsPPix/8);

		switch (Pos)
		{
			case 1:
				// Greater than
				if (LineNow >= Line)
					return TRUE;
				break;
			case 2:
				// Greater and even
				if (!(LineNow MOD 2))
					if (LineNow >= Line)
						return TRUE;
				break;
			case 3:
				// Greater and odd
				if (LineNow MOD 2)
					if (LineNow >= Line)
						return TRUE;
				break;
			case -1:
				// Less than
				if (LineNow <= Line)
					return TRUE;
				break;
			case -2:
				// Less and even
				if (!(LineNow MOD 2))
					if (LineNow <= Line)
						return TRUE;
				break;
			case -3:
				// Less and odd
				if (LineNow MOD 2)
					if (LineNow <= Line)
						return TRUE;
```

```
                                        break;
                                default:
                                        if (LineNow == Line)
                                                return TRUE;
                                        break;
                        }
                        // Below we implement the ability to get interrupted by the ESC key.
                        if (ImapApp.m_DisplayOptions.CheckESC() AND (Count MOD 5000) == 4999)
                        {
                                // Check time elapsed.
                                ULONG m_ulTicksNow = GetTickCount();

                                // Check for ESC key every 50 msecs or so.
                                if (m_ulTicksNow - m_ulStartTicks > 50 OR m_ulTicksNow <
m_ulStartTicks)
                                {
                                        m_ulStartTicks = GetTickCount();

                                        if (ChangePriority)
                                                SetPriorityClass(GetCurrentProcess(),
NORMAL_PRIORITY_CLASS);

                                        MSG Msg;
                                        int bQuit = PeekMessage (&Msg, NULL, WM_KEYUP,
WM_KEYUP, PM_REMOVE);

                                        if (ChangePriority)
                                                SetPriorityClass(GetCurrentProcess(),
REALTIME_PRIORITY_CLASS);

                                        if (bQuit)
                                                if (VK_ESCAPE == int(Msg.wParam))
                                                        return -1;
                                }
                        }
                        Count++;
                }
                return FALSE;
}
```

REFERENCES

1. Harry Newton, *Newton's Telecom Dictionary 15th*, ed. Rag harak, Miller Freeman Inc, NewYork, 1999.

2. Leborgne .P, "Synthesizing A High-Resolution Satellite Image by Merging Several Severed Low Resolution Ones", *1992 International Geoscience and Remote Sensing Syinposium,* 1992.

3. Konig .F, Pracfcke .W, "The Practice of Multispectal Image Acquisition", *Electronic Imaging,* Zurich, Switzerland, P18-20 May 1998.

4. Jeff Child, "Image Processing Boards Leverage PCI & Multimedia Technology," *Electronic Design,* July 1998.

5. Sam shearman, "Digital Video Grabs at Scientific/Industrial Images," *Personal Engineering & Instrumentation News,* July 1998.

6. John C. Russ, *The Image Processing Handbook,* Second edition, CRC Press, London, 1995.

7. Jain, Anil K., *Fundamentals of Digital Image Processing,* ed. Thomas Kailath, Prentice Hall, London, 1989.

8. Moham M. Trivedi, "Analysis Of High-Resolution Aerial Images", *Image Analysis and Processing, 8th International Conference ICIAP '95,* ed. San Remo, Italy, Sept 1995.

9. *AM-DIG Software Manual,* Imaging Technology Inc, 1997.

10. *IC-PCI Hardware Reference,* Imaging Technology Inc, 1997.