Developing an Automated Explosives Detection Prototype Based on the AS&E 101ZZ System

Panagiotis J. Arvanitis

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Richard W. Conners, Chair
A. Lynn Abbott
Peter M. Athanas

September 22, 1997
Blacksburg, Virginia

Keywords: airport security, re-configurable computing, FPGA, device drivers

Developing an Automated Explosives Detection Prototype Based on the AS&E 101ZZ System

Panagiotis J. Arvanitis

Dr. Richard W. Conners, Chair

(ABSTRACT)

This thesis describes the development of a multi-sensor, multi-energy x-ray prototype for automated explosives detection. The system is based on the American Science and Engineering model 101ZZ x-ray system. The 101ZZ unit received was an early model and lacked documentation of the many specialized electronic components. X-ray image quality was poor. The system was significantly modified and almost all AS&E system electronics bypassed: the x-ray source controller and conveyor belt motor were made computer controllable; the x-ray detectors were re-positioned to provide forward scatter detection capabilities; new hardware was developed to interface to the AS&E pre-amplifier boards, to collect image data from all three x-ray detectors, and to transfer the data to a personal computer. This hardware, the Differential Pair Interface Board (DPIB), is based on a Field Programmable Gate Array (FPGA) and can be dynamically re-configured to serve as a general purpose data collection device in a variety of applications.

Software was also developed for the prototype system. A Windows NT device driver was written for the DPIB and a custom bus master DMA collection device. These drivers are portable and can be used as a basis for the development of other Windows NT drivers. A graphical user interface (GUI) was also developed. The GUI automates the data collection tasks and controls all the prototype system components. It interfaces with the image processing software for explosives detection and displays the results. Suspicious areas are color coded and presented to the operator for further examination.

Acknowledgments

I would like to thank Dr. Richard Conners, my committee chairman, for his advice and guidance, and also for having given me the opportunity to work in many exciting research projects during my years of employment in the Spatial Data Analysis Laboratory. I also thank my committee members, Dr. Lynn Abbott and Dr. Peter Athanas, for their assistance in writing this thesis and their teachings throughout my student years.

Several people in the Spatial Data Analysis Laboratory at Virginia Tech have assisted in the completion of this research work. I would like to express my appreciation to Dr. Thomas Drayer for his many suggestions in the art of digital design. I also like to thank the following members of the SDA Lab for all these years of putting up with me: Paul LaCasse, Yuhua Cui, Xinhua Shi, Qiang Lu, Srikathyayani Srikanteswara, Jinhuo Shan, William King, and Chase Wolfinger. Finally, I thank Mr. Bob Lineberry and Mr. Farooq Azam for all their help.

I would like to dedicate this thesis to my parents, Jason and Despina, and my brother, Nicholas. Thank you for all your love and support… I love you guys ☺

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Introduction

1.1 Motivation

Over half a billion people fly in the United States each year, with the number of worldwide travelers exceeding one billion [BOU94]. As the volume of passengers increases, a rise in the number of terrorist attacks on commercial airlines is expected. Since the first airline explosion attributed to plastic explosives in 1982, and after the 1988 tragedy of PanAm flight 103 over Lockerbie, Scotland, the Federal Aviation Administration (FAA) has funded various research projects for the advancement of explosive detection in airport luggage [NOV92].

Older airport security systems rely on human operators to recognize suspicious luggage and then manually inspect it. The system simply presents a number of x-ray images to the operator, who must then identify the shape and material of the contents. The decision whether the luggage will be allowed on the aircraft or not rests solely with the inspector. Although this approach was sufficient in the past, terrorists have found new ways to conceal explosives and mislead security personnel [POL94]. Furthermore, the tedious and repetitive nature of the task, as well as the adverse work conditions in busy airports, have been shown to significantly reduce the ability of security system operators to detect suspect material effectively [NRC96]. In the United States, inspection system operators are hired by private companies and are usually paid only minimum wage; the turn-over ratio of operators reaches 200-300% per year, as most prefer to switch to an easier, better paid position at a local fast-food restaurant.

These facts dictate the need for an automated system that can detect explosives reliably and without human intervention. This new system must have a high probability of explosives detection, a low false alarm rate, and high luggage throughput [BOU94]. Although operators will still be required to analyze luggage that is marked dangerous by the inspection system, the primary decision will rest with the computer. This will reduce

the number of bags that must be visually inspected and, combined with a friendly yet functional user interface, can alleviate the task of the operator. The overall result will be a highly effective security mechanism that will deter terrorism and dramatically improve the safety of air travel.

1.2 Research Objectives

In 1994, the Spatial Data Analysis Laboratory (SDAL) at Virginia Tech, under a grant from the Federal Aviation Administration, initiated a research project for the development of a prototype airport security system for automated luggage inspection. The prototype is based on an American Science and Engineering model 101ZZ system. The 101ZZ was chosen because it utilizes multiple sensor technologies to collect transmission and back scatter images. Transmission images are obtained from energy that directly penetrates the luggage, whereas back scatter images are obtained from energy that did not penetrate the luggage and was scattered back towards the x-ray source. The 101ZZ was modified in the SDAL to collect forward scatter images, by measuring energy that penetrates the luggage but is scattered forward. Combining all three sensor modalities can improve the probability of explosives detection and reduce false alarm rates. The AS&E system was also chosen because of its "flying-spot" technology (described in Chapter 2), which provides very high quality x-ray images from all three sensors. Despite its advantages over conventional systems, "flying-spot" technology has never been explored in airport security for automated explosives detection.

The purpose of the SDAL project is threefold: a) investigate new materials characterization techniques using a multi-energy and multi-sensor approach, b) develop image processing algorithms to detect overlapped objects, and c) design the hardware and software to collect, display and process high-resolution images. The intended outcome of the project is a prototype system that can reliably, quickly and automatically detect explosives without the presence of a human operator. A computer will be used to host the custom data collection hardware, execute the image processing and materials

characterization software, display the processed results, and sound an alert if explosives are detected. The system will be tested by the FAA to determine whether it can be certified as an automated Explosives Detection System (EDS) for check-in luggage, but can also be used to inspect carry-on luggage.

The technologies researched and the overall prototype system must be thoroughly documented, so that they can later be used for the development of a commercial product. The system must provide an easy-to-use operator interface to minimize the migration time from existing technologies, and must be a financially competitive solution to existing technologies and systems.

Finally, the results of this research effort will greatly benefit the scientific and academic community, by providing information on subjects that have so far remained proprietary and classified.

1.3 Contributions to this research

The 101ZZ system used provided very limited explosives detection features and required complete manual control. Data collection and analysis was performed by a human through the operator control panel. Furthermore, the system received by the SDAL was itself a prototype manufactured in the late 1980s. It was controlled by an outdated Intel 8086 based personal computer, and was equipped with mostly wire-wrap boards, rather than printed circuit boards (PCB). No documentation was provided on the operation of the system or the hardware used, and some of the features available on the control panel were not functional.

The decision was therefore made to discard most AS&E system electronics and develop custom hardware and software for the control of the prototype. Furthermore, the original system was modified to better suit the objectives of this research activity. The purpose of this thesis is to describe the following:

- **Modifications** to 101ZZ system. This includes the addition of a new forward scatter detector, as well as changes made to provide computer control of all system functions and develop an automated EDS.

- **Hardware development**. A hardware collection device was developed to collect data from three input sources, pre-process the data and multiplex it on a single high speed bus. Flexibility was maintained at the board level as well as the logic level by using a Field Programmable Gate Array (FPGA). The FPGA can be dynamically re-programmed and allows the same hardware to be implemented in many different applications. The device has been used as a general purpose collection board with CCD array and linescan cameras using differential pair signals.

- **Software development.** Windows NT device drivers were developed for the hardware described in this thesis, as well as a custom Multiple Channel PCI board (MCPCI) used for bus master DMA transfers to the host computer. Utilities to control the hardware were ported from DOS to Windows NT, and a new utility was developed to display color and black and white images under the Windows operating system. Finally, a graphical user interface (GUI) was written as the front-end to the explosives detection system. The GUI automates the collection process, interfaces to the image processing and materials characterization software, analyzes the results, and displays the processed images to the operator. It also sounds an alarm if explosives are detected.

1.4 System Overview

The prototype system is based on the American Science and Engineering (AS&E) 101ZZ airport security system. The 101ZZ is originally equipped with two x-ray sources and three detectors: a transmission and back scatter for one side of the luggage, and a

back scatter detector for the other side. The second x-ray source was removed from the prototype, and its corresponding back scatter detector moved to create a forward scatter detector. The advantages of this approach are explained in Chapter 2. The 101ZZ is equipped with a computer controllable x-ray source controller, and two infrared beam break sensors used to determine the position of the luggage on the conveyor belt. It also uses a motor controller for the conveyor belt.

The 101ZZ uses a digitizing pre-amplifier board to convert the analog voltage from an x-ray detector to a digital signal. These boards use a differential pair bus for communications and require external control, since they contain no "intelligent" hardware for autonomous operation. There are three pre-amplifier boards in the system, one for each of the three detectors. The original AS&E pre-amplifier boards are used in the prototype, since they provide a simple external interface and are integrated in the AS&E system. Using this existing hardware reduces development time and cost.

To control the pre-amplifier boards and transfer data to the PC, a Differential Pair Interface Board (DPIB) was developed. The DPIB interfaces to the three digitizing boards and multiplexes the output on a single bus. To minimize development time and cost the DPIB was designed as an ISA device. A re-programmable FPGA allows the DPIB to be used in many different applications, of which the current AS&E system configuration is only one example.

The Multiple Channel PCI (MCPCI) board, developed by Paul LaCasse in the SDAL, is used to transfer image data to the host computer through the PCI bus. The MCPCI is a PCI bus master DMA device and can achieve data rates over 100 times faster than ISA hardware, allowing real-time system operation. The DPIB and the MCPCI communicate over an external Zee bus, a high speed data bus developed for inter-board communications [DRA97b]. Using the MCPCI and DPIB together, instead of developing a PCI version of the DPIB, has numerous advantages:

a) Reduced system cost. An ISA device is cheaper to develop and implement than PCI hardware, which requires special chipsets to operate.
b) Reduced development time. PCI hardware is more complex to design than ISA hardware.
c) Reduced debugging time. The MCPCI existed and had already been tested in other applications when the DPIB was being developed. The debugging effort therefore concentrated on the simpler ISA device.
d) Future usability. In designs where real-time operation is not a necessity, the DPIB can be used stand-alone to transfer data through the ISA bus, eliminating the additional cost of the PCI hardware.

Furthermore, using the Zee bus allows the DPIB to interface to the MORRPH (Modular Re-programmable Real-time Processing Hardware), which uses multiple FPGAs to perform complex image processing tasks in real-time [DRA95a]. The DPIB-MORRPH-MCPCI combination yields a powerful computing device that can be implemented in many image processing and other data processing applications.

In the prototype system, the DPIB also interfaces to the conveyor belt motor and infrared sensors of the 101ZZ. The on/off state and direction of the conveyor belt, as well as the status of each infrared sensor are accessible through software on the host computer.

A single IBM compatible PC running Windows NT 4.0 is currently used for system control and image processing. Windows NT was chosen for its stability, user friendliness, and multi-tasking and multi-processor capabilities. All software for system control, image processing, and the GUI execute on this PC. The GUI presents the processed images to the operator and allows simple image manipulation functions, such as zooming and inverting, to better interpret the results.

An overview of the system configuration using the custom hardware and the host computer is shown in Figure 1.1. The modifications to the AS&E system are discussed in

Chapter 3. Hardware and software development are discussed in Chapter 4 and Chapter 5 respectively.

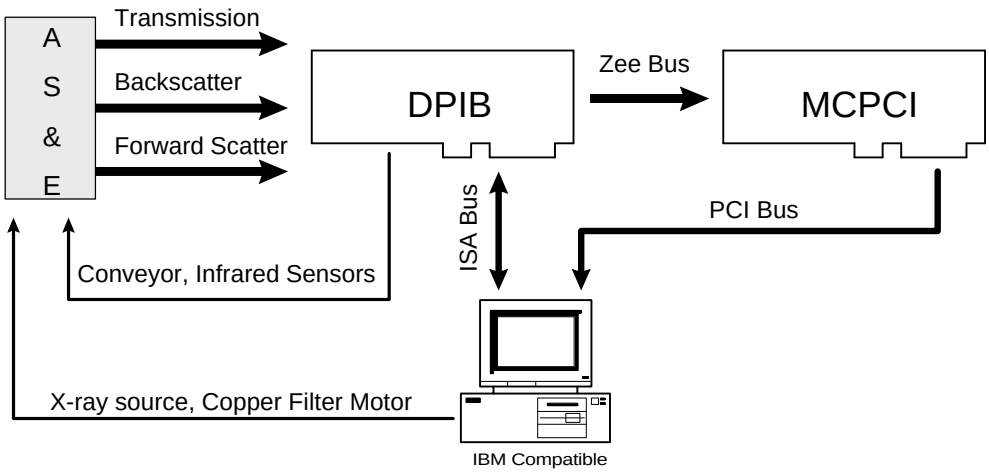


Figure 1.1 Prototype system overview

Chapter 2. Background

This chapter describes the physical principles used in x-ray scanning. The x-ray imaging techniques used in the prototype luggage scanning system are explained. Also, three commercially available airport security systems are examined, and their strengths and weaknesses discussed.

2.1 Principles of x-ray imaging

There is a number of methods used in the detection of explosives in airport luggage, including conventional x-ray imaging, vapor detection [CHU96], quadrupole resonance analysis [RAY96], nuclear techniques [GOZ91] and x-ray based computed tomography (CT) [ROD91]. X-ray imaging is the most widely used method.

A typical x-ray imaging system includes a radiation source to generate the x-ray field, and a detector to convert x-ray energy to an electrical signal. The source and detector are placed on opposite sides of the x-ray tunnel, directly across from each other. The detector is used to measure the attenuation of the x-ray energy, called the transmission energy, as it penetrates through the object of interest.

Early security systems used high energy x-ray for the detection of weapons. At higher energy levels (over 100 KV), the absorbed energy depends primarily on the density of the material: the higher the density, the more energy is absorbed by the object, therefore the darker the image. A metal object, such as a weapon, or an explosive device, both dense materials, would appear very dark in the transmission image and would be detected. However, an explosive device could be concealed behind a denser material, making it invisible to the system operator [VOU94].

To resolve this problem, luggage is scanned at two energy levels. At lower energies (usually 80 KV), the absorption depends mainly on the effective atomic number

(Z_{eff}), as well as the thickness of the material. Using multi-energy transmission images, high density and low Z_{eff} explosives can be identified. Figure 2.1 illustrates the density and effective atomic number of materials of interest in luggage inspection systems. A system that uses two x-ray energy levels for scanning is called a dual-energy system. Several techniques exist for the collection of multi-energy images, including varying the input energy of the x-ray source [EUR96], filtering the energy at the x-ray sensor [EIL92] and using multiple sensors with different spectral responses [MIC93].

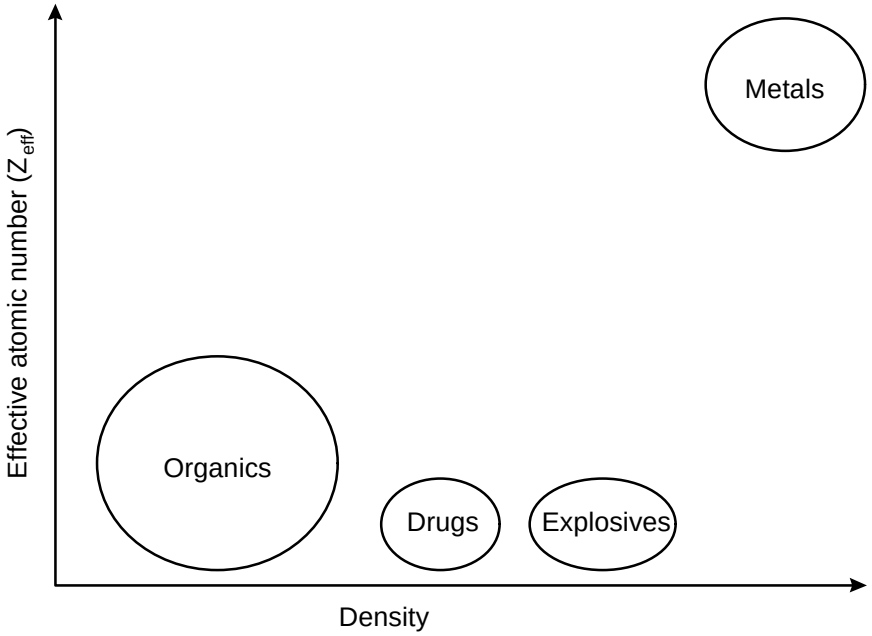


Figure 2.1 Z_{eff} vs. Density of selected materials

Yet another technique in x-ray imaging is to measure energy that is scattered due to the Compton effect [FAI94]. Although some radiation penetrates the luggage in a straight path and is measured by the transmission detector, some of the x-ray energy is scattered either forward through the scanned object, or backwards, towards the x-ray source. The images obtained by measuring scattered energy are called the forward scatter and back scatter images. A single energy system using back scatter images for explosives detection was developed by American Science and Engineering [SCH91].

The prototype system discussed in this thesis is a true dual-energy system that collects transmission and scatter images. The x-ray energy is varied by changing the input energy to the x-ray tube, rather than filtering. The transmission images are used to obtain an estimate of Z_{eff}. Using both high and low energy images provides a more accurate estimate of the effective atomic number. Density information is obtained using the forward and back scatter images. Scatter energy provides more useful density information than any transmission image. The relationship between measured energy and material density is rather complicated and is analyzed in other literature [ARE96].

2.2 Sophisticated Commercial Luggage Inspection Systems

There are several luggage inspection systems available in the market. Some use single energy, transmission images and are considered outdated. Others use advanced scanning techniques and provide automated luggage inspection. These systems do not require a human operator and are used in a number of airports today. Three of the more sophisticated airport security systems are examined in this section.

2.2.1 Vivid Technologies

Vivid Technologies was formed in July 1989, following the explosion aboard PanAm 103. The company specializes in airport security systems for luggage inspection. The most sophisticated system manufactured by Vivid Technologies is Model VIS, introduced in 1993 and intended for the inspection of checked luggage. The VIS is a dual-energy system and uses a transmission detector for image collection. It can scan 900-1500 bags per hour without a human operator [VIV97].

To enhance the explosives detection capabilities of their systems, Vivid also offers the Scatter Detection Enhancement (SDE) and SDE-2 modules. The SDE uses a forward scatter detector, whereas the SDE-2 uses both a forward and back scatter detector. These modules are available for certain Vivid systems, and are only used in the detection of sheet explosives.

Vivid systems are not very widely used in the United States. Although they are capable of detecting many types of explosives, they do not meet FAA requirements for EDS certification.

2.2.2 American Science and Engineering

American Science and Engineering (AS&E) is the inventor of flying-spot x-ray technology, which eliminates the need for an expensive sensor array by using a concentrated beam of x-rays and a few large photo-multipliers for detection (see Chapter 3). Most AS&E systems are equipped with scatter detectors and can collect back scatter images. Forward scatter detectors are an option on a limited number of systems. Using these technologies, high Z (metallics) and low Z (organics, plastics) images are simultaneously presented to the operator to help identify attempts to conceal dangerous substances [ASE96].

Using flying-spot technology, AS&E systems can obtain very high quality scatter images. Most conventional x-ray scanning systems provide good quality transmission images, but suffer from poor quality scatter images, yielding mediocre density estimations. With good transmission and scatter quality, materials characterization can greatly be improved, increasing the probability of detection and reducing the number of false alarms.

Despite the advantages of the flying-spot technology, it has been implemented on only a small number of airport security systems, none of which provide automated explosives detection. AS&E, the only flying-spot developer, concentrates on systems for car and truck (cargo) search to identify illegal substances, and personnel inspection systems, to detect weapons. Airline luggage presents a much different problem: explosives are present in smaller quantities and are better concealed.

2.2.3 Invision Technologies

Invision is the developer of a very sophisticated, automated airport security system, the CTX 5000. This system uses computed tomography (CT) and a rotating x-ray source and detector pair to obtain a three-dimensional view of the luggage. The typical two-dimensional x-ray view provides no depth information and can be confusing. The operator can be mislead when certain objects are used to conceal explosives. By presenting a three-dimensional view, however, luggage contents are easily identified and any ambiguous overlap is detected. This assists the human operator and also improves the detection probability of the explosives detection algorithm.

The CTX 5000 scans luggage using a single energy x-ray source and a transmission detector, and presents the projected luggage view to the operator. Areas that might contain explosives are identified by a red vertical line on the image. To further analyze a region, the operator clicks on one of the lines. The x-ray source and detector are then re-positioned and the luggage scanned at the location selected by the operator. The cross sectional view at the plane of the red line is then presented on a separate monitor. Explosives are painted in red, and explosive type and quantity information is displayed [INV96].

The CTX 5000 is the only system certified by the Federal Aviation Administration as an Explosives Detection System (EDS). However, its excellent detection capabilities come at a very high price: a single unit currently sells for over $1,000,000. Furthermore, although a throughput of 300 bags per hour is claimed, actual airport trials have averaged 100-150 bags per hour, causing concerns whether the system can be integrated into airports without causing significant delays. The large size of the system (14 ft. long, 9,350 lbs) has also been a problem in placing it in already crowded airports. Finally, an operator trained to detect explosives on a typical, two-dimensional system requires several months of training to identify luggage contents on the new CT images [NEW96].

2.3 Summary

Of the three major airport security system manufacturers, the Invision CTX 5000 is the only true EDS. Other systems provide a good solution for inspection of carry-on luggage with the assistance of a human operator, but are not robust enough to become the primary detection technology in airports. However, the CTX 5000, despite its excellent capabilities, remains slow, big, and very expensive. Although many foreign airports are subsidized by government grants and can afford a CTX 5000, in the United States airport security systems are purchased by individual airlines. The high cost of the Invision system, as well as the large number required to maintain a minimum luggage inspection rate, has prevented most domestic carriers from using this system.

None of the systems available has explored the high quality forward and back scatter images provided using the flying-spot technology for automated explosives detection. Using both scatter measurements is significantly less expensive than a computed tomography system, and may also provide a "smarter" system, with a higher probability of explosives detection and a lower false alarm rate. This research project provides an automated, multi-sensor and multi-energy prototype to assist in the development of new algorithms for explosives detection. The combination of these techniques and the new image processing software could pave the road to a new approach in aviation security technology.

Chapter 3. System Design

The purpose of this chapter is to describe the overall design of the AS&E 101ZZ luggage inspection system, and the changes and additions made to the hardware for the purpose of this research project. The type and positioning of the x-ray source and detectors, the "flying-spot" technology, the detector pre-amplifier boards, as well as all the computer controlled hardware are examined.

3.1 General Introduction

To develop new image processing algorithms for explosive detection in airport luggage a system for exposing objects to x-ray radiation and collecting raw images is necessary. The Federal Aviation Administration requested that the prototype be based on the American Science and Engineering 101ZZ system. This system was chosen because of its transmission and scatter collection capabilities, and also to further explore the applications of flying-spot x-ray technology (see Section 3.2.2) in automated explosives detection for airport luggage.

The hardware delivered to Virginia Tech was an early version of the 101ZZ. The system electronics cabinet contained an outdated 8086 PC motherboard. The data collection activity was controlled through an array of special purpose ISA hardware, some of which were on wire-wrap, rather than printed circuit boards. No technical or other documentation was provided about this specialized hardware. The images collected by the system were of low resolution and rather poor quality, with obvious vertical striping. There was no method of collecting raw, uncompensated images from the x-ray detectors, and the nature of the image processing filters applied to the data remained unknown.. The system provided no features for automatic control, and most system components had to be accessed manually from the operator console. Furthermore, the system computer provided only a simple, text-based user interface through a monochrome monitor.

The decision was therefore made to completely bypass the majority of the system electronics and develop hardware and software to control the collection sequence, obtain x-ray images and transfer them to a PC for processing. This new hardware provides high resolution, high quality images that can be transferred to the PC either raw or after some simple image processing filters. The thorough hardware documentation provided in this thesis can be consulted to correct any system failures. If the original 101ZZ electronics were used, determining the cause of any hardware problems and correcting them would be an extremely difficult and time-consuming task. Bypassing the system electronics also allows modification of other system components to eliminate the need for an operator control panel and to completely automate the collection sequence through a personal computer. Finally, the custom software integrates the new hardware with the existing system, simplifies system control and provides an intuitive graphical user interface that minimizes the chance of operator error.

Data is collected from the 101ZZ using the AS&E digitizing pre-amplifier boards. These boards convert the analog x-ray detector signal to digital format, perform some primitive processing, and transmit the output value over a differential pair bus. The original digitizing boards were retained because they are simple in operation and could be easily analyzed, and also because they are controlled completely through external signals and operate autonomously without affecting other system components. Re-using the digitizing hardware reduced development time and cost.

To control the AS&E pre-amplifier boards and retrieve image information, a Differential Pair Interface Board (DPIB) was designed for the ISA bus. The DPIB uses a Field Programmable Gate Array (FPGA) chip for re-configurable computing. The FPGA can be dynamically re-programmed by the host computer, making the DPIB a general purpose data collection and processing hardware, rather than restricting it to the AS&E system application. The DPIB is paired with the Multiple Channel PCI board (MCPCI) developed by Paul LaCasse, a bus master DMA device for the PCI bus, which allows real-

time data collection. The MCPCI also uses a FPGA for added versatility. Development of the DPIB is analyzed in Chapter 4.

Other 101ZZ components that are also used in the prototype include the conveyor belt motor, the infrared luggage sensors, and the x-ray controller, source and detectors. Some modifications, which are explained later in this chapter, were performed to these components to make them computer controllable. A retractable copper filter was added to the system to filter the output energy at the x-ray source, as is discussed in Section 3.3.4.

The prototype system is controlled by a Dell Optiplex P120 PC, running Windows NT Workstation 4.0. This PC hosts the DPIB and MCPCI, and also interfaces to every system component either through the DPIB (conveyor belt, infrared sensors), or through a serial communications port (x-ray controller, copper filter motor). Software was developed to control and automate the data collection sequence, interface with the explosives detection algorithms and present the results to the system operator. Device drivers were also developed to access the custom hardware (DPIB and MCPCI) under Windows NT. The software development is discussed in Chapter 5.

3.2 AS&E System Design

The 101ZZ resembles a typical airport security system. It uses a conveyor belt for the transport of luggage, and two black and white monitors to display the resultant images. Most functions are available through the operator control panel: conveyor direction control, x-ray power switch, and simple image processing functions, such as invert and zoom. The system is controlled through an 8086 computer, enclosed in the system electronics cabinet. All data collection and processing boards reside on the ISA bus of the host computer. A keyboard and CGA monitor connector are available for access to a text-only interface, allowing the user to alter system settings or save and

retrieve images. The original configuration of the AS&E 101ZZ system is illustrated in Figure 3.1.

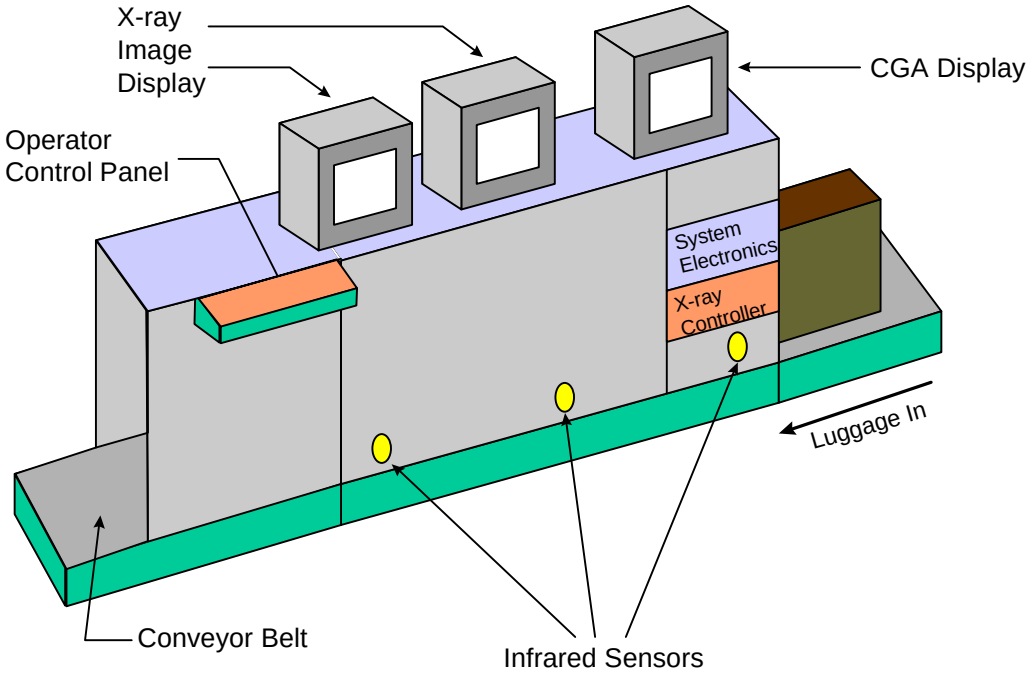


Figure 3.1 AS&E 101ZZ System

The AS&E system was originally equipped with two x-ray sources, one for each side of the luggage. The first x-ray source is used to collect a transmission and back scatter image for one side of the bag. The other is used to collect only a back scatter image of the other side of the bag, to assist in overlapped object resolution.

The AS&E system is also equipped with three infrared beam break sensors, used to detect the presence of luggage. The sensors are positioned in the front, middle and rear of the tunnel. The conveyor belt and x-ray source are continuously turned on, and collection begins when either the front or rear sensor is broken. The conveyor belt can be controlled only from the operator panel. The x-ray source controls are found on a separate panel, directly underneath the system electronics housing. Data collection with the 101ZZ is hardly an automated task; a trained human operator is required.

3.2.1 X-ray source and detector positioning

The materials characterization algorithms developed for this project only require image information from a single x-ray source. The second source, used to collect only back scatter information, was removed. The back scatter detectors from the obsolete source were then placed directly in front of the transmission detector, in order to investigate forward scatter images. The digitizing board was connected to the new forward scatter detector. Figure 3.2 illustrates the current placement of the x-ray source and the sensing elements.

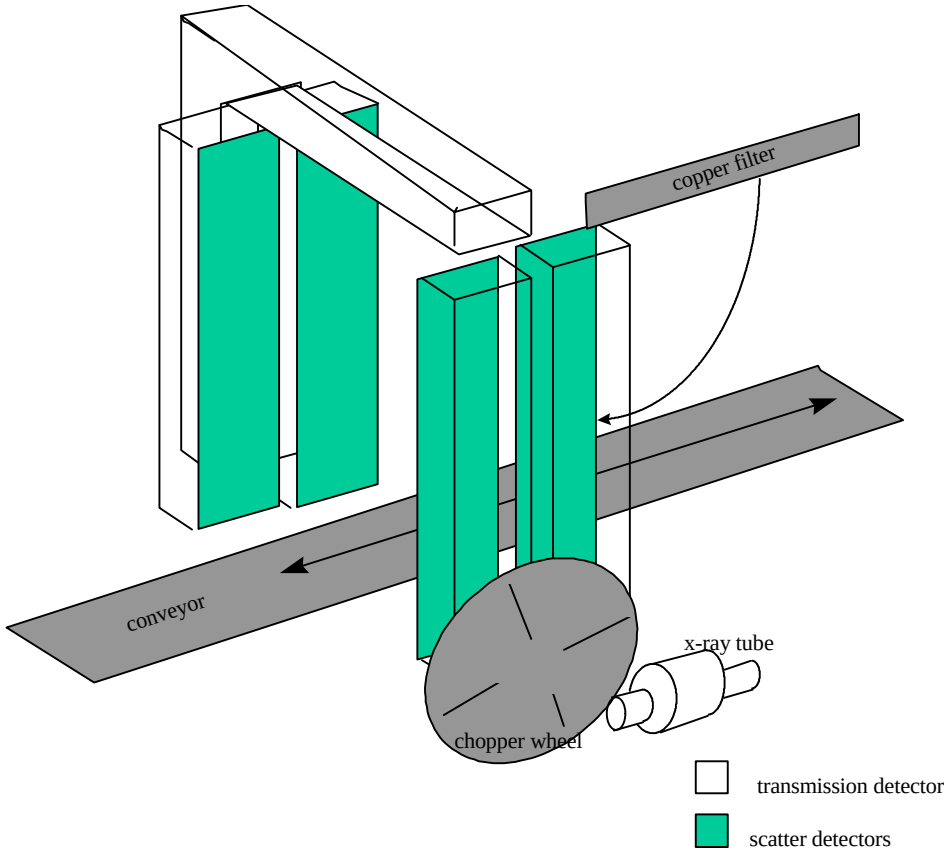


Figure 3.2 Modified source and detector placement

3.2.2 Flying-spot technology

An x-ray source, much like any radiation source, generates a field of energy, rather than a concentrated beam or plane. Most systems utilize a slit collimator to restrict the output to a narrow plane of radiation. The collimator is commonly made of lead or another shielding material, with a narrow vertical slit through which x-rays can pass. Only a small portion of the luggage is viewed at a time; the collimator effectively partitions the luggage into smaller vertical regions, or scan lines. An array of sensing elements is then used to sub-divide the exposed region into smaller parts and effectively generate a pixelated image. This configuration is shown in Figure 3.3. Using this approach, the vertical image resolution is limited to the number of sensing elements in the array, whereas the horizontal resolution can be controlled by varying the conveyor belt speed. Moreover, the large number of sensors in the detector significantly increases overall system cost.

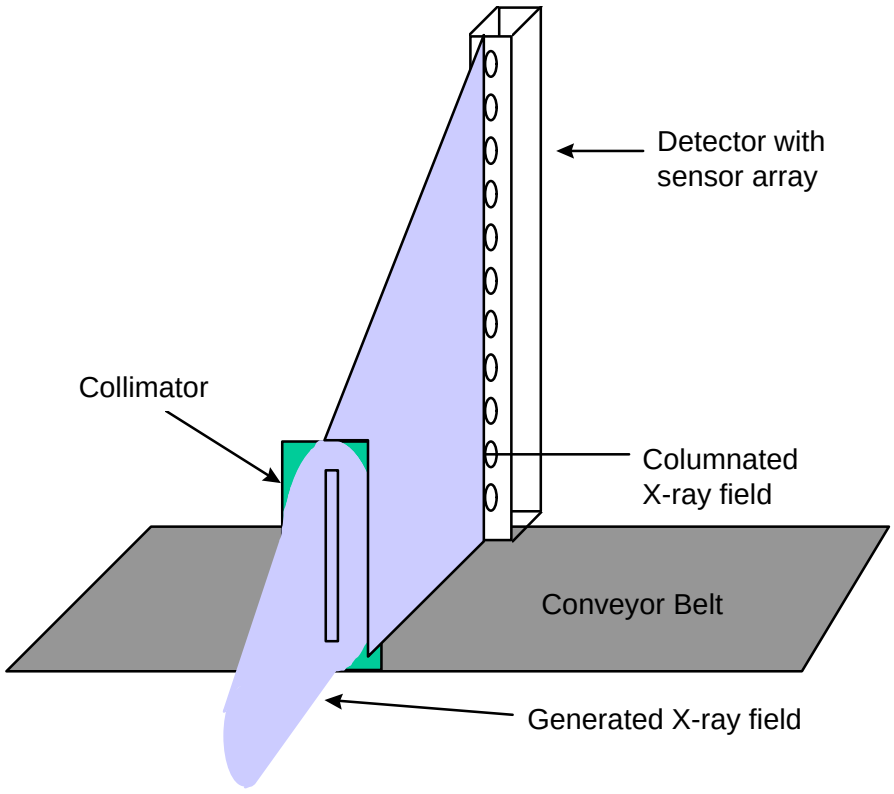


Figure 3.3 Collimated x-ray with sensor array

American Science & Engineering has developed a different, more flexible approach to sampling the exposed object at discrete points. In addition to the collimator, a chopper wheel fabricated of shielding material, is inserted in the x-ray path. The wheel has four narrow slots and rotates at a constant speed. The effect of the wheel is to block the collimated x-ray plane and create a narrow beam. As the wheel turns, the beam moves from bottom to top, scanning an entire vertical line. The movement of the beam creates a pixelated image, eliminating the expensive sensor array. Instead, a few large photo-multiplier tubes are used. The operation of the flying-spot technology is shown in Figure 3.4. Using a traveling beam allows control of both the horizontal and vertical resolution, and reduces system cost by using a less expensive detector element.

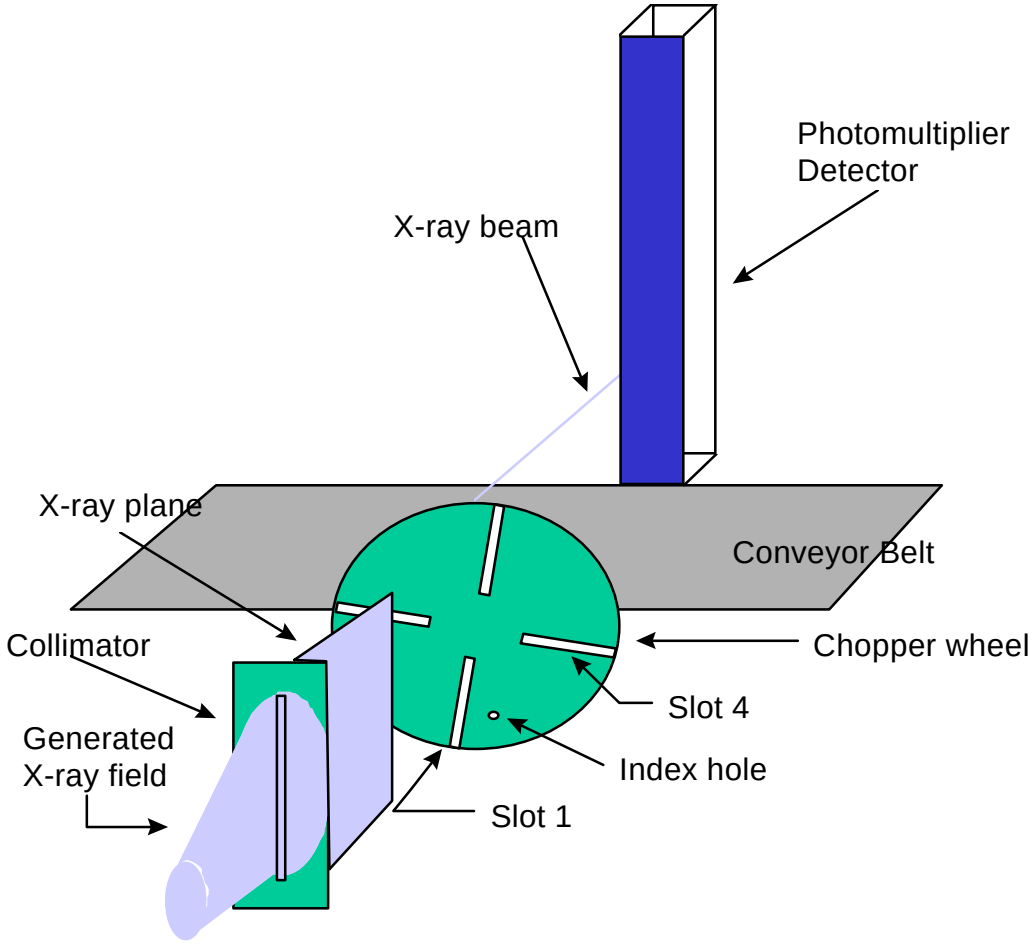


Figure 3.4 Flying-spot technology operation

To synchronize data collection with the position of the chopper wheel, two pairs of incandescent lamps and photo-transistors are used. One pair is aligned with an index hole on the chopper wheel (see Figure 3.4) and is used to identify Slot 1. A **WHLRST** (wheel reset) pulse is generated by the chopper wheel logic when Slot 1 enters the field of view. The other lamp-transistor pair is aligned to indicate when any of the four slots enters the field of view and generates a **WHLSYNC** (wheel reset, also **SLOTSYNC)** pulse. Since there are four chopper wheel slots, four **WHLSYNC** pulses are generated for every **WHLRST** pulse. The chopper wheel rotates at 1800 RPM, generating the signals shown in Figure 3.5. **WHLSYNC** indicates when the x-ray beam is at the lowest point of its path, and is used to start the collection of a new line. **WHLRST** is used to start a new frame, an operation that is further explained in Chapter 4.

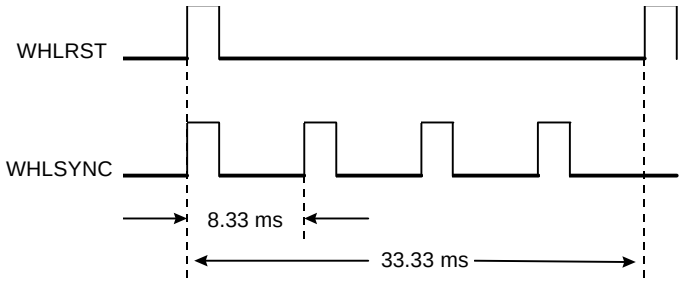


Figure 3.5 Chopper wheel synchronization pulses

3.2.3 Digitizing pre-amplifier boards

The signal obtained from the x-ray detectors is analog in nature and must be converted to a digital value for image processing. The conversion process, as well as a very primitive shading correction operation, is performed on the AS&E pre-amplifier boards. Each x-ray detector uses a separate pre-amplifier board. These boards require external control and contain no “intelligent” hardware, such as a micro-controller.

The data and control bus of the pre-amplifier boards are accessible through a 50-pin protected header connector. There is an 8-bit bi-directional data bus, and a 4-bit, input only control bus. Power is also supplied to the boards through the 50-pin protected

header. These signals make up the Data and Control Interface (DCI) of the pre-amplifier boards. The DCI uses differential pair signals.

The block diagram of the digitizing boards is shown in Figure 3.6. The incoming analog signal first passes through sample and hold circuitry, which stores the current analog pixel value. Shading correction follows. There, a DC offset is added to the analog value. The purpose of this process is to correct for the non-linearity and slight geometrical imperfections of the chopper wheel slots, allowing uniform image quality. A narrower slot reduces the x-ray energy projected on the object and results in slightly darker pixel values. Images that are not shade compensated contain visible vertical striping. The shade compensation value for a pixel is placed on the DCI data bus by the external hardware. A digital-to-analog converter is then used to convert the digital value to an analog offset that is added to the sample-and-hold output.

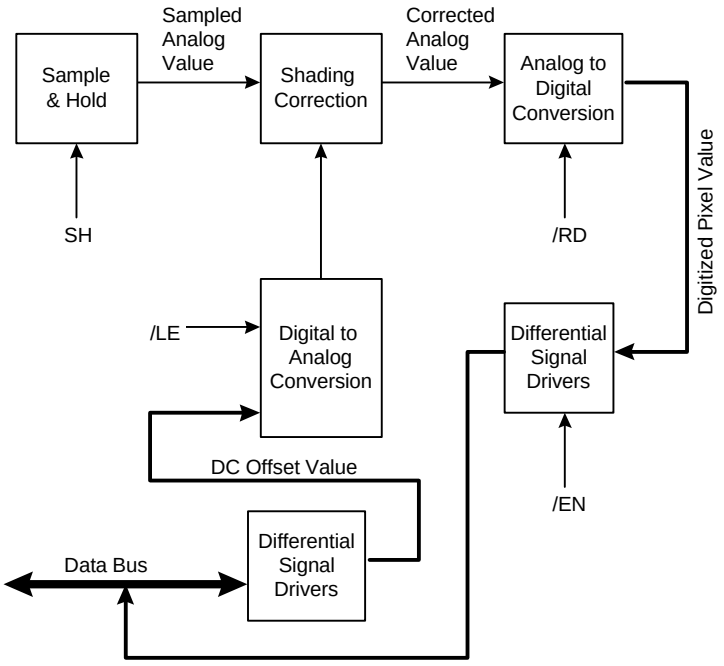


Figure 3.6 Digitizing pre-amplifier board block diagram

The compensated value is then digitized by a digital-to-analog converter. The output of the converter connects to a set of differential pair drivers for output on the DCI. The drivers can either operate in high-impedance mode (while the pre-amplifier boards read the shade compensation value from the DCI), or in output mode. The direction of the data bus is controlled through an external control signal. An external pulse is also used to initiate the conversion process.

Figure 3.7 shows the signal assignment on the DCI. The polarity of the incoming and outgoing data bus signals is reversed, per AS&E convention.

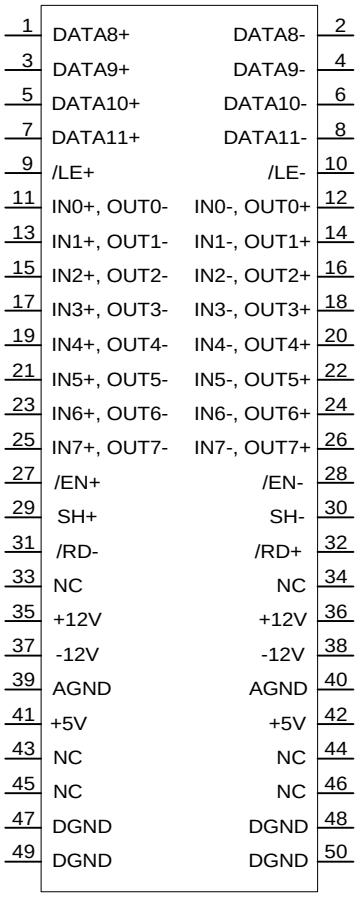


Figure 3.7 AS&E pre-amplifier board signal assignment

There are four control signals on the DCI that fully control the circuitry on the digitizing boards. These are as follows:

- **/EN**: used to determine the direction of the differential pair data bus. When high, the pre-amp board is in input mode and the shade compensation value should be available on the bus. When low, the drivers are enabled, and the digitized pixel value is placed on the data bus [MOT93].
- **SH**: gate pulse for the sample and hold circuitry. The incoming analog value from the x-ray source is sampled on the rising edge of this pulse.
- **/LE**: latches the data compensation value on the data bus into the digital-to-analog converter. This value is used to generate the offset added to analog x-ray detector signal.
- **/RD**: controls the conversion process on the digital-to-analog converter. The conversion is started on the falling edge of this pulse. After completion of the process, the output of the DAC remains constant only while this signal is low. The DAC output is placed in high-impedance mode when this signal is high [ANA91].

The waveforms generated by the AS&E system electronics are shown in Figure 3.8. The timing values for a vertical resolution of 128 pixels are shown in Table 3.1.

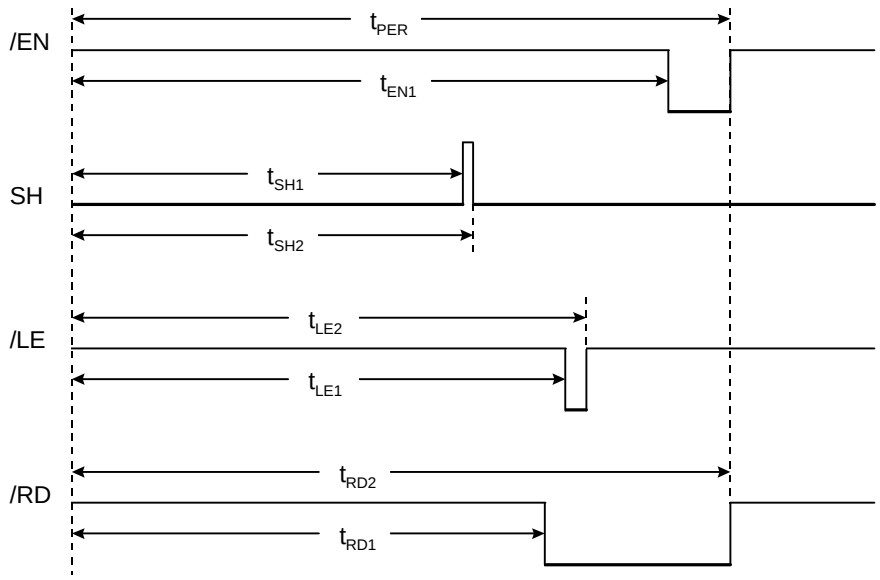


Figure 3.8 AS&E pre-amplifier board waveform

Table 3.1 AS&E system timing values

Variable	Time (μs)
t_{PER}	53.62
t_{EN1}	53.28
t_{SH1}	45.89
t_{SH2}	46.32
t_{LE1}	47.78
t_{LE2}	49.99
t_{RD1}	46.94
t_{RD2}	53.62

3.3 System Components

3.3.1 X-ray source controller

The prototype system is intended to collect image data at multiple energy levels. Therefore, computer control of source settings, such as x-ray voltage and current, is necessary. The AS&E 101ZZ system is equipped with a Gemini 2000 x-ray controller, manufactured by Gulmay, Inc. The controller features an operator panel, where system settings can be changed, and an RS-232 serial port for data communications. A comprehensive set of codes to either transmit commands or request system information is provided [GUL95]. These commands are summarized in Table 3.2 and Table 3.3.

Table 3.2 x-ray controller request codes

Command	Description
?V	X-ray voltage. Responds ?Vnnn<CR>
?I	X-ray current. Responds ?Innn<CR>
?T	Elapsed time. Responds ?Tnnn<CR>
?M	Current mode. Responds ?Mnnn<CR>

All requests codes use a single character, followed by a carriage return. The result is usually a question mark, followed by the same character, and then a three digit number. The voltage, current, and time requests return the corresponding value of the x-ray settings. The mode command, used to determine the state of the source, returns one of the following values:

- 000: Key is in position 2, x-ray source can not be turned on
- 001: x-ray off
- 002: x-ray warming up
- 003: x-ray switching on or off.
- 004: x-ray on.

Commands consist of a exclamation point, then a one character code, and are terminated by a carriage return. A three digit numerical value is used with some commands and follows the character code.

Table 3.3 x-ray controller command codes

Command	Description
!V	Set voltage. Format: !Vnnn<CR>
!I	Set current. Format: !Innn<CR>
!T	Reset timer
!X	Turn x-ray on.
!O	Turn x-ray off.

3.3.2 Infrared luggage sensor

Imperative to a completely automated system is a sensor that can report the position of the luggage in the x-ray tunnel. This information can be used to enable and disable image collection, and also to turn the x-ray source on when luggage is present, and off when the tunnel is empty.

The prototype system uses two of the three infrared beam break devices, located at the front and rear of the tunnel. These devices are powered by the AS&E system and return a binary value. A value of one (high) indicates that the path is clear, whereas a value of zero (low) is returned when the beam path is broken.

3.3.3 Conveyor belt

Luggage is commonly transported by means of a conveyor belt. The AS&E system is equipped with such a belt, a motor and a motor controller. However, the system

motor controller could only be controlled through the operator control panel and provided no data interface for computer control.

To allow for automatic control of the belt, the motor controller was modified. The controller uses two lines for conveyor operation. If both lines are open, the conveyor is stopped. By shorting one of the two lines, the conveyor can move in either the forward or reverse direction. For the prototype system, these lines were interrupted and a relay inserted in the current path. The relays are controlled by the DPIB and determine the on-off state and direction of the belt. There is still no control of the motor speed, unless the setting is altered on the motor controller housing.

3.3.4 Copper Filter

As discussed previously, the prototype system scans luggage at high and low x-ray energy settings. The energy level of the output photons, however, is not limited to the input energy, but is distributed over a range of frequencies. There is also significant overlap between the low and high energy spectral distributions, and the total output energy at 150 KV is much higher than at 80 KV, as shown by Xinhua Shi [DRA97a]. For materials characterization purposes, it is desirable to minimize the overlap region between the two energy spectrums and balance the total output energy. To achieve this, a copper filter was added to the system, as shown in Figure 3.2. The filter is inserted only during high-energy collection, and is removed at all other times. The subsystem was designed by Jinhuo Shan, using a Velmex 8300 series stepper motor driver [VEL85]. The filter is attached to the cylindrical motor mount, and can be rotated between the fully removed and fully inserted position. Protection switches are installed on the assembly to prevent motor damage if rotation is attempted outside the system boundaries.

The motor controller can be accessed either through the external control panel, or an RS-232 serial port. The controller uses a BASIC interpreter to accept and run simple programs to control the motor. In the prototype system, the control code is downloaded

through the serial port and then executed. The motor then awaits a numerical value (followed by a line feed character), and moves the motor to the absolute angle designated by the input value. The origin point (position of angle 0), is the filter position at the time the controller code is executed.

3.4 Workstation Setup

The outdated 101ZZ host computer was replaced by a high performance personal computer to reduce algorithm execution times and provide computer control of all the prototype system components. The PC is equipped with a high resolution monitor and video adapter to display the processed results to the system operator. The new computer is used to control the system hardware, collect data through the custom hardware boards (DPIB and MCPCI), process the x-ray images, and display the output. If explosives are detected in the luggage, they are highlighted in the output image and an alarm is sounded.

A Dell Optiplex P120, running the Windows NT 4.0 Workstation operating system is used. Windows NT was chosen for its reliability, availability and low cost. The multi-processor capabilities of this operating system are also very important as plans exist to move to a dual or even quad CPU system. The current image processing algorithms and software structure support a high degree of parallelism. Moving to a symmetric multiprocessing system will significantly reduce the time required to scan and process a bag.

The PC hosts the DPIB and MCPCI boards. The DPIB resides on the ISA bus, whereas the MCPCI uses the PCI bus for high speed transfers. Both devices are controlled by the system software through devices drivers developed specifically for these boards. The development of these drivers is discussed in Chapter 5. The DPIB is used to interface to the AS&E pre-amplifier boards and collect image data. It is also used to return luggage sensor information to the PC, and allow conveyor belt control.

All system automation is controlled by the graphical user interface, *Galaxie*. This program controls the high and low energy collection sequence, interfaces to the image processing and materials characterization programs, and outputs the processed data to the operator, producing an alarm if dangerous substances are detected. The software development effort is discussed in Chapter 5.

Chapter 4. DPIB Hardware

The purpose of this chapter is to describe the hardware structure of the Differential Pair Interface Board. The DPIB is described at the board level, analyzing the functionality of each external functional block, and the logic level, describing the operation of each module in the FPGA. The information in this chapter is intended to aid in the initial setup and configuration of the DPIB by the end user, as well as a reference guide to a hardware designer wishing to modify the internal logic. Although the discussion focuses on the DPIB design for the x-ray imaging system, the re-configurable nature of the board supports many data collection applications with minimal developer effort. The re-configurability of the DPIB is explored in Section 4.4.

4.1 Design Overview

The decision to discard the AS&E system electronics created the need for the development of new hardware to collect data from the AS&E pre-amplifier boards and transfer the data to the PC. The new hardware must meet the following design requirements:

- **Collect data from three input sources**. Differential pair signals, with a bi-directional data bus, are required. The data interface should be based on the AS&E system protocol to simplify connection to the pre-amplifier boards.
- **Multiplex data**. The three input sources must be combined on a single output bus and transferred to the PC.
- **Control system components**. Access must be provided to the conveyor belt motor controller, the infrared sensors, and any other system devices that will be controlled by the host computer.
- **Interface to the PC**. This interface will be used to access I/O ports to control system components, and also to access image data.

- **Programmability**. Any system variables, such as timing signal parameters or shading correction values, can be changed from the PC through IO ports.
- **Flexibility**. The hardware design, both at the board level and the logic level, must be flexible and allow interfacing to other data sources, such as CCD or linescan cameras. Designing a general purpose hardware device will increase its applications and reduce development time and costs in other projects, by eliminating the need to design specialized hardware.

To satisfy these requirements, the Differential Pair Interface Board (DPIB) was created. The DPIB is designed to serve as a general purpose data collection device, as is discussed in Section 4.4. It is based on a Xilinx XC4000 series Field Programmable Gate Array (FPGA) chip. The FPGA can be re-programmed through the PC interface, allowing the DPIB to be used for a variety of image processing and general data processing applications. The flexibility of the DPIB continues at the board level to simplify connection to different external data sources. A block diagram of the DPIB is shown in Figure 4.1.

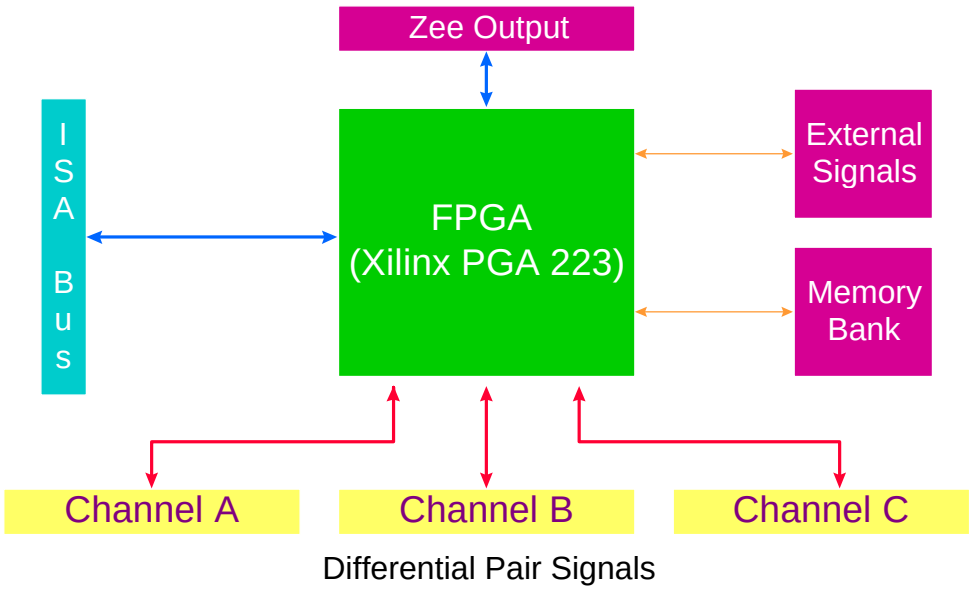


Figure 4.1 DPIB Block Diagram

The main processing unit of the DPIB is a Xilinx 4000 series FPGA. The FPGA is a fully re-programmable computing resource consisting of Configurable Logic Blocks (CLBs) and routing resources to connect the CLBs. The number of CLBs and their propagation delay is determined by the family type, size and speed rating FPGA.

Each CLB uses two independent function generators (FG) to implement any Boolean function of four variables. A third 3-input function generator is used to combine the outputs of the two FGs with a third input from outside the CLB. Two edge-triggered D-type flip-flops with a clock enable are also used as storage elements. The flip-flops can be programmed to operate in synchronous or asynchronous mode and can be triggered on either the rising or falling clock edge.

The DPIB uses a 223-pin PGA socket that can accept most FPGA chips in the XC4000 family. The size and speed rating used on the DPIB is determined by the specific application, rather than the architecture of the board. Smaller designs can use a slower, smaller FPGA (such as the XC4005H), whereas large designs, such as the current AS&E system design use larger, more expensive chips (XC4013). Table 4.1 lists some FPGA chips that are compatible with the DPIB and their characteristics [XIL94a], [XIL94b].

Table 4.1 XC4000 Family FPGA chips accepted on the DPIB

FPGA type	CLBs available	Equivalent Gates	Price
XC4005H	196	5,000	$266
XC4010E	400	10,000	$203
XC4013E	576	13,000	$393
XC4020E	784	20,000	$460

Communication with the external data sources is accomplished through the differential pair channels. A 12-bit bi-directional bus and four output only control signals are provided on each channel connector. The specific function of each pin on these connectors is determined by the FPGA program and can be changed depending on the current application. Also available on the DPIB are a memory bank for data storage and an external signal interface. The latter provides access to TTL or other level signals that can not be connected through the differential pair interface. Finally, the ISA interface connects to the ISA bus of the host PC. It is used to upload the FPGA program during initialization, or to communicate with FPGA I/O registers while the DPIB is in operation. Image data can be transferred to the DPIB through this interface.

The Zee output connector is used to interface the DPIB to other image processing hardware. The Zee bus was developed as a standard for inter-board communications by Thomas Drayer and William King [DRA97b]. This bus can be used to connect the DPIB to the MCPCI, for high-speed data transfers. The MCPCI accepts data from up to six channels from a Zee connector. The data is de-multiplexed and then transferred to the host PC using bus master direct memory access (DMA). The high throughput of the PCI bus (up to 132 Mbytes/sec) [PCI93] compared to that of the ISA bus (up to 1 Mbyte/sec) [EGG91] makes real-time system operation possible. The Zee output connector can also be used to interface the DPIB to the MORRPH board (MOdular Re-programmable Real-time Processing Hardware) [DRA97a]. The MORRPH is a general purpose, FPGA based processing unit intended for real-time image processing. It can be used with the DPIB and MCPCI in a variety of applications requiring real-time performance to collect data from multiple input sources, process and transfer the data to a personal computer.

4.2 Board Level Description

The structure of the DPIB may be divided into the following functional blocks:

a) *data interface*, which connects to the input data source, such as the AS&E pre-amplifier boards,

b) *Zee bus interface*, which allows data processed by the DPIB to be transferred to another device (MCPCI, MORRPH) for further processing,

c) *ISA interface*, which provides communication with the host PC during and after initialization of the hardware,

d) *sensor signal interface*, which makes external control signals available to the DPIB, and

e) *memory bank*, which is used to store data compensation values for the shading correction circuitry.

Each of these interfaces is discussed in further detail. A component location diagram identifying the location of each connector and integrated circuit in the DPIB is provided in Appendix A. The board level schematics of the DPIB are also included in Appendix A.

The DPIB currently uses a 14.318MHz oscillator for the FPGA. The clock frequency is determined by the FPGA speed and program, and can easily be changed by inserting a new oscillator in the socket.

4.2.1 Data Interface

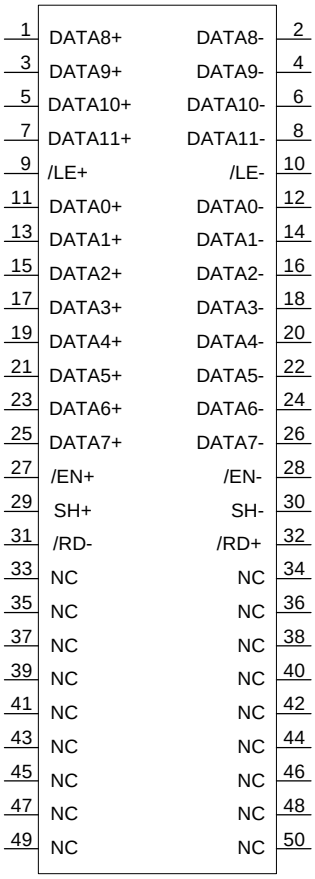
The data interface is a bi-directional data and control signal bus used to connect to the image data source. The bus was designed using the AS&E pre-amplifier bus protocol as a guide, although certain extensions have been made to allow for a wider variety of input sources. The data interface uses RS-422 differential pair signals exclusively.

A total of twelve data bits and four control bits are available on each data interface connector (J1, J2, and J3). The bi-directional data bus is used to receive image data and transmit shading correction coefficients. A set of three DS26LS31 drivers and three DS26LS32 receivers is used to convert between TTL level signals used on the DPIB and RS-422 level signals used on each connector. When not used, the drivers are placed in high impedance mode to prevent bus contention with the AS&E hardware. It should be noted that, although the DPIB is configured for twelve bit operation, the current AS&E system configuration can only accept eight bit data. The four higher order bits have been used with other input data sources, such as B&W cameras.

The control bus is a unidirectional output bus. A single DS26LS31 driver is used for each group of control signals. The output enable pins of these drivers are hardwired and can not be used to select high impedance mode.

A set of resistor network packs is used with each data bit to allow impedance matching between the DPIB and the input data source. Some data sources require the use of termination resistors for noise reduction and line balancing. The resistor SIPPS should remain empty when using the AS&E system, since termination resistors are provided on the sending end.

The data interface bus is physically available through a 50-pin polarized protected header. The pin assignment for this connector is shown in Figure 4.2. The polarity of the lower eight data bits, as well as the polarity of the control signals, is determined by the AS&E protocol. The change in polarity in the **/RD** signal (pins 31 and 32) is not a typographical mistake, but rather the choice made by AS&E in the pre-amplifier board connectors.



Pin Name	Function
DATA[11:0]	Data Bus
/LE	Latch enable
/EN	Driver Enable
SH	Sample & Hold
/RD	Convert
NC	No Connection

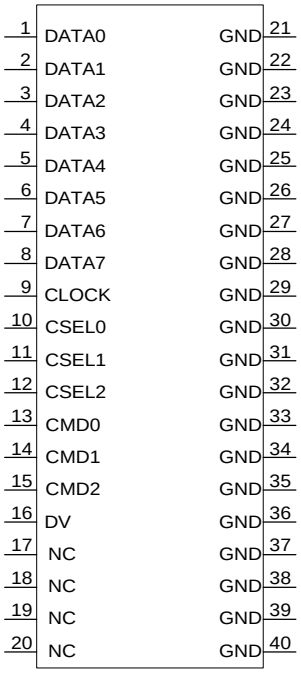
Figure 4.2 Data Interface Connector signal locations

4.2.2 Zee Bus Interface

The Zee bus was developed by Thomas Drayer and William King as a standard bus for inter-board communication in an image processing system [DRA97b]. It is a unidirectional, synchronous, 16-bit bus with an 8-bit data bus. The Zee bus connector on the DPIB is used for communication with either the MCPCI board, to transfer image data to the host PC via the PCI bus, or the MORRPH board, for real-time processing of the data before transferring it to the PC. All Zee bus signals are buffered through two 74LS245 data buffers (U5 and U6).

The pinout of the 40-pin, polarized Zee bus connector (J4) is shown in Figure 4.3. The lower eight bits are the data bus. The higher eight bits constitute the control bus,

used for synchronization and transfer of Zee bus commands. All signals are valid during the high portion of the clock. The channel select bits, **CSEL[0:2],** are used to determine which channel is currently being transferred. The command bits, **CMD[0:2],** are used to indicate a line start, a line end, or a marking cycle (a cycle during which nothing happens, there is no valid data on the bus, nor is a Zee command being issued). Finally, the **DV** bit is used to indicate valid data. The data bus should be sampled only when **DV** is high.



Zee Bus commands used

if **DV** = 0	
CMD = 000	marking cycle
CMD = 010	line start
CMD = 011	line end
If **DV** = 1	
CMD = 100	valid 1 byte data
CMD = 101	valid 2 byte data

Figure 4.3 Zee bus signal locations and commands

4.2.3 ISA Interface

The ISA interface is used during initialization to program the Xilinx FPGA, but also while in operation, to access registers connected to the luggage sensors and the conveyor belt motor. The interface is based on an Altera EP324 EPLD and was developed by William King for use on the MORRPH-ISA board. It has been modified by the author to use the **IORDY** signal on the ISA bus [EGG91]. It was experimentally observed that on some faster motherboards FPGA programming would occasionally fail. The timing of two consecutive ISA write cycles is faster than the time required for an

FPGA cell to be programmed and would result in data loss. By connecting the **RDY/BUSY** line of the FPGA to the ISA **IORDY** signal, bus activity is suspended until the FPGA cell has been programmed.

The ISA interface provides access to the DPIB through three consecutive ISA ports: the *address port*, *data port*, and the *program port.* The address port holds the address of the DPIB register to be accessed. The lower five bits of the address port are used, allowing access to a total of 32 registers on the FPGA. The data port is used to hold the data written to or read from the DPIB register. Finally, a write to the program port selects the re-program line on the FPGA. This erases the current FPGA configuration and prepares it to be re-programmed, possibly for a different application. The operation of the ISA interface is summarized in Table 4.2 below. The ISA base address of the DPIB is determined by the EPLD and can not be changed, unless a new EPLD is used. The current base address is set at 0x0304.

Table 4.2 ISA Interface port description

Port	**Offset from base ISA**	**Description**
address (W)	0	Address of FPGA port to access, only lower 5 bits are used
data (RW)	1	Data read from or written to port
program (W)	2	Re-program FPGA, only bit 0 used

To properly control the DPIB, the desired FPGA port address must first be written to the address port, regardless of whether a read or write operation will be performed. The ISA interface then performs a mapping of the contents of the address port to the address select lines of the FPGA. The programmer can access the FPGA port by either reading from the data port, which will return the data contained in the FPGA, or writing to the data port, which will transfer the data to the FPGA register. Any port operation,

therefore, will take two ISA cycles to complete: a mandatory write cycle, followed by a read or write cycle.

4.2.4 Sensor Signal Interface

The Sensor Signal Interface (SSI) is used to control outside devices, such as the conveyor belt, and return information to the DPIB, such as luggage sensor information, or chopper wheel synchronization signals. The SSI uses a 9-pin female D-sub connector (J5). The signal assignments on the connector are shown in Table 4.3.

All signals coming into the DPIB pass through a potentiometer and a 74LS245 buffer (U7) before entering the FPGA. The potentiometer is used to lower the voltage of certain incoming signals (such as the chopper wheel synchronization pulses) to appropriate TTL levels. Any signals exiting the DPIB, such as the conveyor belt control signals, pass through a relay, to isolate the board from the external device.

Table 4.3 Signal assignment on SSI connector

Pin	Description
1	Wheel sync signal
2	Slot sync signal
3	Common ground
4	Relay input for conveyor reverse
5	Relay input for conveyor forward
6	Front infra-red sensor
7	Rear infra-red sensor
8	Relay output for conveyor reverse
9	Relay output for conveyor forward

4.2.5 Memory Bank

The memory bank of the DPIB is used to store a look-up table of shade compensation values. In the current system configuration, shading correction is performed by the pre-amplifier board using DPIB supplied offset values. The same memory bank may be used to hold correction coefficients for digital shading correction, performed on the DPIB.

There is one memory slot for each data interface, configured to use MCM6264 8Kx8 static RAMs [MOT92]. Due to the limited number of pins available on the FPGA, the three memory chips share a common address and data bus. The output enable lines of the memory ICs are used to access an individual memory bank and prevent bus contention.

4.3 Logic Level Description

This section describes the internal FPGA logic at the gate level for the prototype system design. Some of the modules discussed here are specific to the AS&E system and the digitizing pre-amplifier boards, but most modules can be re-used in a variety of designs. The schematics for the AS&E design are available in Appendix B.

To facilitate the design of the DPIB and to have access to an extensive library of commonly used components, the MORRPH Development System (MDS) was used to compile this design [DRA97b]. MDS is a collection of software tools and hardware libraries developed by Thomas Drayer for image processing applications. MDS modules exist on several levels and are flattened by the MDS and Xilinx software. For example, an ADDER module would require the following components:

- ADDER symbol, used on the top level schematics with other MDS modules. Typically accepts and outputs data in SUIT bus format [DRA97b].

- ADDER schematic, which connects the ADDER symbol input signals and the common clock and reset lines of the MDS design to the XADDER symbol. Buses for FPGA IO registers also connect to the ADDER symbol here.
- XADDER symbol, which contains the actual XADDER schematics.

The MDS multi-level architecture is shown in Figure 4.4.

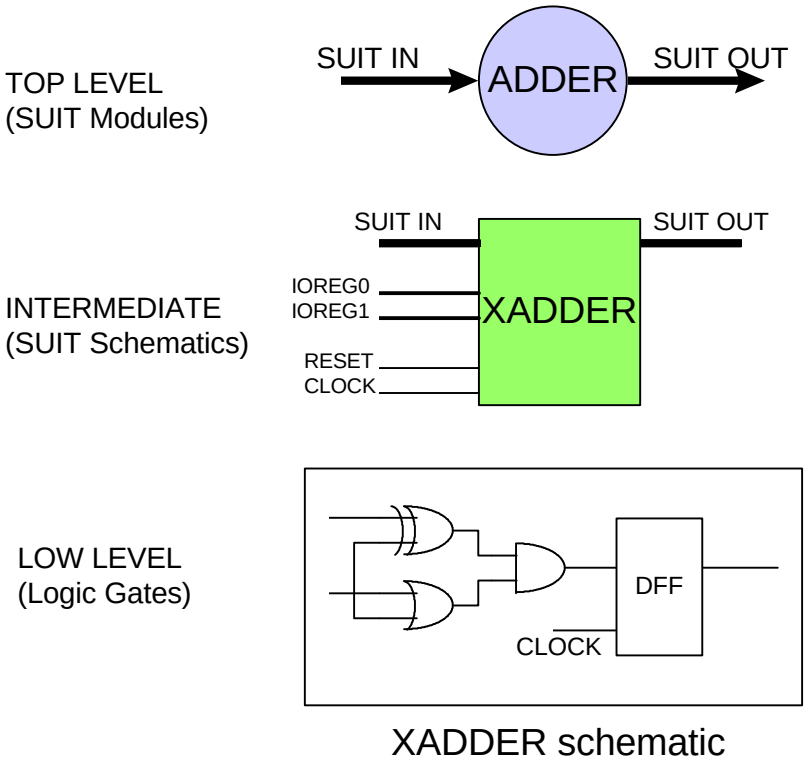


Figure 4.4 MDS Multi-level Architecture

An FPGA register is automatically created by the MDS using the Architecture Configuration File (.ACF) (see Section 4.3.9). Any pre-existing MDS modules are only discussed briefly in this thesis. For a more detailed analysis of MDS, please refer to [DRA97b].

4.3.1 Data Interface Connector Modules (ACON, BCON, CCON)

The purpose of the ACON, BCON, and CCON modules is to interconnect the FPGA to each of the data interfaces. There is no processing performed in these modules, only a mapping of signal nets to Xilinx FPGA pins. Each outgoing signal passes through an output buffer (OBUF) and then connects to a pad (PAD). Each incoming signal connects to a PAD and then an input buffer (IBUF). Tri-state buffers are used with the data bus, to allow bi-directional data transfer. Due to an error in the early documentation of the AS&E pre-amplifier boards, which mislabeled the polarity of the signal connectors, the incoming data bits must be inverted.

4.3.2 Zee Bus Connector Module (MCON)

The MCON module connects the FPGA to the Zee bus connector. All signals to the Zee bus are registered to guarantee that they will remain unchanged during the high portion of the clock.

4.3.3 Sensor Signal Connector Module (DCON)

The DCON module is used with to pass sensor signals into the FPGA, and output control signals through the Sensor Signal Interface connector. Unlike the previous connector modules, there is some signal conditioning performed.

The chopper wheel synchronization pulses and the infrared sensor signals are registered through a D flip-flop (INFF), to meet the setup and hold times of any circuitry using these signals. The chopper wheel signals are used in the data collection process, whereas the beam-break sensor signals are connected to an ISA port for use with the control software. There are two infrared sensor bits, one for the front and one for rear beam-break devices. These bits are set when the beam path is clear, and are reset when the path is interrupted.

The conveyor belt control signals pass through DCON and are connected directly to the 74LS245 buffer and then the relays. These signals need not be registered, as they are obtained directly from a registered ISA port.

4.3.4 Control Signal Generator (CONTROL)

The CONTROL module generates all timing signals required by the AS&E pre-amplifier boards. Its inputs are the chopper wheel signals, available from DCON. Its outputs are: a) **WHLSIGS**, the processed wheel signals, b) **SIGS**, the AS&E timing signals passed to the back scatter and forward scatter detectors, and c) **FASTSIGS**, the faster control signals used with the transmission detector. The vertical resolution of the transmission image must be twice the resolution of the back scatter and forward scatter images. The frequency of the transmission control signals is therefore doubled.

4.3.4.1 Control Signal Generator Sub-module (XCONTROL)

The lower level XCONTROL module accepts nine 8-bit data values, which are used to generate the timing pulses. Although they are currently hardwired and defined at compile time, an FPGA port may be used to dynamically program these values. The cost of such a design is increased utilization of FPGA resources.

The XCONTROL module also accepts two 10-bit values: **NUMPIXF**, which is the vertical resolution of the transmission image, and **NUMPIXS**, which is the resolution of the back scatter and forward scatter images. Due to the nature of the pixel counting circuitry, the resultant image size is actually less than these values by one pixel. Therefore, for an image resolution of 450 pixels, these values should be set to 451.

4.3.4.2 Function Generator Sub-module (SIGGEN2)

The SIGGEN2 module is used to generate pulses with programmable period and duty cycle. It uses three 8-bit input variables (**LTIME, HTIME, RERIOD**) to determine

the pulse shape. This module is composed of an 8-bit counter, three 8-bit comparators and a flip-flop. The counter is used to count clock pulses.

The output of the SIGGEN2 module is low until **LTIME** pulses have been counted. Once the **LTIME** count is reached, the output goes high and remains so until **HTIME** pulses are counted. The output then falls and stays low until **PERIOD** is reached, at which time the circuitry is reset and the cycle is repeated. The relationship between the input values and the output waveform is shown in Figure 4.5. The SIGGEN2 module is also equipped with an output enable signal **EN**. When this signal is low, there is no output from the SIGGEN2 module.

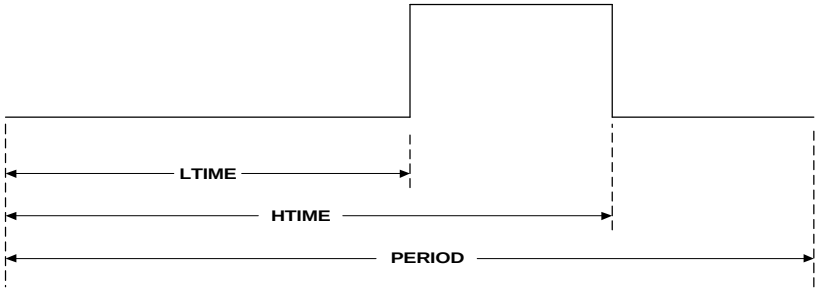


Figure 4.5 SIGGEN2 output waveform

A SIGGEN2 module is used for each AS&E timing signal. Since there are three pre-amplifier boards with four control signals per board, a total of twelve modules should be used. However, because the resolution and chopper wheel timing of the forward scatter and back scatter images is equal, the same timing signals can be used for these detectors. As a result, only eight SIGGEN2 modules are required. Their outputs are combined in the **CSIGSFAST** and **CSIGSNORM** buses, and are output from the CONTROL module.

The output waveform of the SIGGEN2 module depends on the clock frequency of the design, as well as the value of the input variables. For a circuit operating at frequency f, and for low, high and period times of T_{high}, T_{low} and T_{period} respectively, the corresponding input values can be determined from the following equations:

$$PER = \frac{T_{period}}{f} \qquad \text{(Eq. 4.1)}$$

$$LTIM = \frac{T_{low}}{f} \qquad \text{(Eq. 4.2)}$$

$$HTIM = \frac{T_{high} + LTIM \cdot f}{f} \qquad \text{(Eq. 4.3)}$$

The values used with the 14.318 MHz DPIB oscillator are shown in Table 4.4. The resultant timing signals is shown in Figure 4.6, and their timing values are shown in Table 4.5. The function generator output is enabled only during the collection of a line. Once the target line width has been reached, the SIGGEN2 output is disabled until the collection of a new line.

Table 4.4 Timing values for CONTROL module

Variable	Value (hex)
PERIOD	70
LTIM_EN	6C
HTIM_EN	70
LTIMSH	50
HTIMSH	52
LTIM_LE	5E
HTIM_LE	5F
LTIM_RD	59
HTIM_RD	70

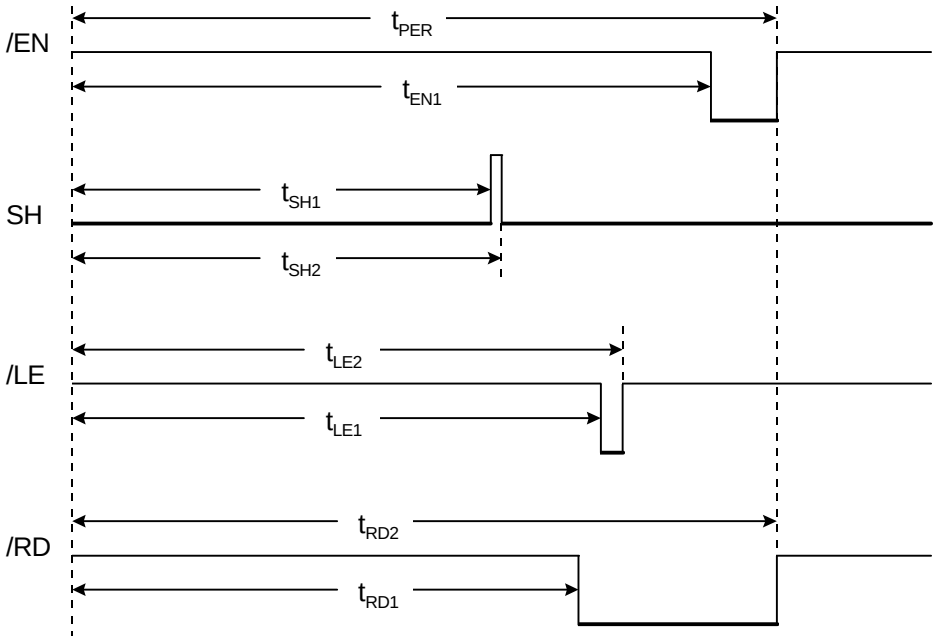


Figure 4.6 Pre-amplifier board timing diagram

Table 4.5 Pre-amplifier signal times

Variable	Transmission Time (μs)	Forward and Backward Scatter Time (μs)
t_{PER}	15.64	31.28
t_{EN1}	15.09	30.18
t_{SH1}	11.17	22.34
t_{SH2}	11.45	22.90
t_{LE1}	13.13	26.26
t_{LE2}	13.27	26.64
t_{RD1}	12.43	24.86
t_{RD2}	15.64	31.28

4.3.4.3 Variable Hysteresis Sub-Module (VARHYST)

Another function contained in the CONTROL module is the conditioning of the chopper wheel synchronization signals. Because these are TTL signals traveling a long path, they are prone to line noise and ringing. A VARHYST module is used to perform hysteresis on these signals.

The module uses a programmable 4-bit value to determine how many clock cycles the input signal must stay valid for the output state to change. The output of the VARHYST module maintains its old value until **HYSTVAL** clock pulses have been counted and the input signal has remained stable. The objective is to filter any glitches in the input signal, which would reset the system logic and erroneously start the collection of a new line. The processed chopper wheel signals are output on the **WSIGS** bus, which is explained in Table 4.6.

Table 4.6 WSIGS bus description

Net Name	Description
WHLSIGS0	Slot sync pulse, after hysteresis and one-shot
WHLSIGS1	Wheel sync pulse, after hysteresis and one-shot
WHLSIGS2	Slot sync pulse, hysteresis only
WHLSIGS3	Wheel sync pulse, hysteresis only

4.3.5 AS&E format to SUIT format conversion (ASE2SUIT)

The purpose of this module is to synchronize the collection process, and output incoming data in SUIT bus format. The SUIT bus is used at the top level of the DPIB design for compatibility with the MDS, and to allow access to MDS library components.

The ASE2SUIT module has three input buses. It connects to a DCI Module (ACON, BCON, CCON) for access to the pre-amplifier board signals. It also connects to

the CONTROL module to obtain timing pulses (**SIGS** bus) and processed chopper wheel signals (**WHLSIGS** bus). It only has one output, which is in 16-bit SUIT format.

4.3.5.1 AS&E to Suit Conversion Sub-module (XASE2SUIT)

The ASE2SUIT module connects to the pre-amplifier data bus to either output a shading correction value, or input image data. The **/EN** signal is used to toggle the data bus direction. When **/EN** is low, the AS&E system is in output mode and is driving the differential pair bus. The DPIB then reads the digital pixel value. At all other times, the DPIB is driving the bus and is transmitting the shading correction value.

A state machine is used to enable the differential pair drivers on the DPIB and AS&E system and avoid bus contention. Simply using the **/EN** pulse to disable the AS&E drivers and an inverted **/EN** to enable the DPIB drivers would result in brief periods of bus contention. The **EN** pulse would turn on the DPIB drivers quickly, but would not have propagated to the AS&E drivers to place them in high impedance. To avoid this situation, a separate pulse is used on the DPIB and the AS&E. The DPIB drivers are first placed off-line using the **DRVEN** signal. The output enable (**/EN**) is transmitted to the AS&E two clock cycles later. Similarly, a two clock cycle delay is allowed between the signal to disable the AS&E drivers and before the DPIB drivers are enabled. The operation of the state machine is illustrated in Figure 4.7.

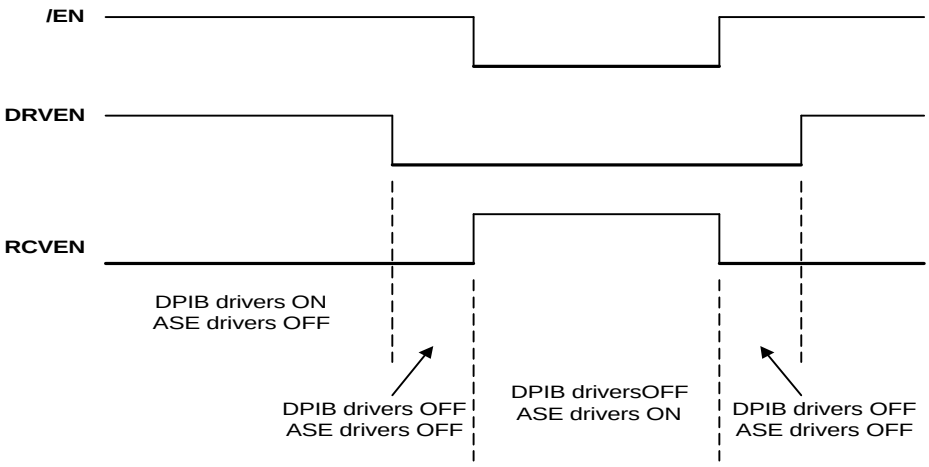


Figure 4.7 Differential bus arbitrator state machine

4.3.5.2 SUIT bus command generation

The ASE2SUIT module generates the necessary SUIT bus commands and data. The data portion of the bus is simply the incoming data bus from the AS&E system. The data is registered to assure stability during the active portion of the clock.

The SUIT bus commands are generated by a 4x8 multiplexer, which selects from one of the following options: *marking*, *start of line*, *end of line* or *valid data*. A *valid data* command is issued one clock cycle after the DPIB RS-422 receivers are enabled. The one cycle delay is generated by a one-shot and allows for signal propagation on the DPIB-AS&E connection. An *end of line* is signaled at every occurrence of the chopper wheel reset pulse. The same pulse, delayed by two clock cycles, is used to indicate the start of a new line. Any clock cycle where none of these conditions are met selects the *marking* command.

The chopper wheel reset pulse (**WHLRST**), instead of the slot synchronization pulse (**SLOTSYNC**) is used to start a new line. This results in a line of data that is four times wider than the programmed width, and contains image information from all four

slots combined. Each slot information must be extracted on the PC. The purpose of using the wheel reset pulse is to simplify the shading correction software. To correct for the non-uniformity between the different chopper wheel slots, there must exist a method of identifying the starting slot on a collected image. A solution would be to use a command to start collection on the DPIB. However, this approach would significantly complicate the DPIB and the MCPCI logic, as they are both designed for real-time operation. Using the **WHLRST** pulse, a line of data always starts with Slot 1 and is followed by the other three slots. Extracting image information only requires changing the file header, an operation that can be performed very quickly in software.

4.3.6 SUIT bus multiplexer (MULTIPLEX, MULTIPLEX4)

The SUIT bus multiplexer is a standard library component of the MDS. It is used to combine two SUIT buses on a single output bus. The module is used in the DPIB to multiplex the outputs of the three ASE2SUIT modules onto a common bus for output to the Zee connector.

The multiplexer includes a memory element (FIFO) for each incoming bus. A SUIT bus command, other than a marking cycle, is first stored in the FIFO. The data available flag of each FIFO and a priority arbitrator are then used to transfer data to the output. Any data on the first FIFO is transferred on each consecutive clock pulse, until that FIFO is empty. The second FIFO data is then transferred. If both FIFOs are empty, a marking cycle is issued on the SUIT bus.

The standard FIFO depth of a multiplexer module is three. Experimental results showed that a standard MULTIPLEX module was insufficient for the final stage multiplexer and caused data drop-out, demonstrated by very bright pixels in the output image. A larger multiplexer module (MULTIPLEX4) was therefore used to correct the problem.

4.3.7 Suit to Zee bus conversion (SUIT2ZEE_SLOW)

The SUIT2ZEE_SLOW module is a modified version of SUIT2ZEE, an MDS library component. It is used to convert the SUIT bus format to a clocked Zee bus, suitable for output to other hardware. The SUIT bus is used for board level designs and uses a common system clock. The Zee bus is used for inter-board communication and adds a clock to the bus specification [DRA97b].

The SLOW qualifier was added to the module name to indicate that the speed of the Zee bus clock is actually half of the DPIB logic clock. Operating at 14.318MHz, the DPIB was transmitting data at 7.5MHz. This can cause problems with the MCPCI board, which uses a 16MHz oscillator, but samples at 8MHz. Therefore, the Zee clock was reduced to 3.25MHz to ensure reliable data transmission, achieving a data rate of 3.25 Mbytes/sec. The DPIB and MCPCI data rates will improve significantly when printed circuit board (PCB) versions are manufactured (both boards currently exist only in wire-wrap format).

4.3.8 Self-test (CHECK)

The CHECK module is used to verify the operation of the FPGA after it has been configured. It is accessible through three FPGA registers: a write and read register pair, use to write and read back a value, and a read-only register which should always return the same check value (0x05a). It is recommended that these ports be read after FPGA programming to verify that the operation was completed successfully.

4.3.9 Control Registers

The ISA ports available to the host computer are created by the MDS using an Architecture Configuration File (ACF). The header of the ACF file used in the DPIB design is included in Appendix B. A list of IO registers available on the DPIB and their addresses is shown in Table 4.7.

Table 4.7 DPIB port locations and description

Port Name	Module	Address	Type	Bits	Description
OUT	CHK1	0	R	7:0	Check port, read back port 1
IN	CHK1	1	W	7:0	Check port
CVAL	CHK1	2	R	7:0	Check port, always 5A
CORVAL	ASE1	3	W	7:0	Correction value to pre-amp
CORVAL	ASE2	4	W	7:0	Correction value to pre-amp
CORVAL	ASE3	5	W	7:0	Correction value to pre-amp
CBUS	DCON	6	W	0	Conveyor forward
CBUS	DCON	6	W	1	Conveyor reverse
CBUS	DCON	7	R	2	Infra-red sensor, front
CBUS	DCON	7	R	3	Infra-red sensor, rear

4.4 Other DPIB Applications

The discussion of the DPIB has so far concentrated on the its application to the explosive detection system. The overall design of the hardware, however, is intended to allow for the DPIB to operate as a general purpose data collection device with any hardware that utilizes differential pair signals. A recent application where the DPIB was used to collect image data from B&W cameras is briefly discussed here.

Using the DPIB with a different source must be addressed at two levels: the board level, where the physical connection between the DPIB and the source is established, and the logic level, where hardware modules are created to control the source and retrieve data.

At the board level, the DPIB is equipped with three 16-bit busses, each with twelve bi-directional bits and four output only signals. This allows for 8-bit data transfers with eight control bits, or 12-bit transfers with four control bits. The on-board 50-pin

connectors were chosen to allow expandability. In the case that the input source uses a different size connector, or a different pin assignment, a passive break-out board can be quickly constructed to convert to the different format. This is illustrated in Figure 4.8.

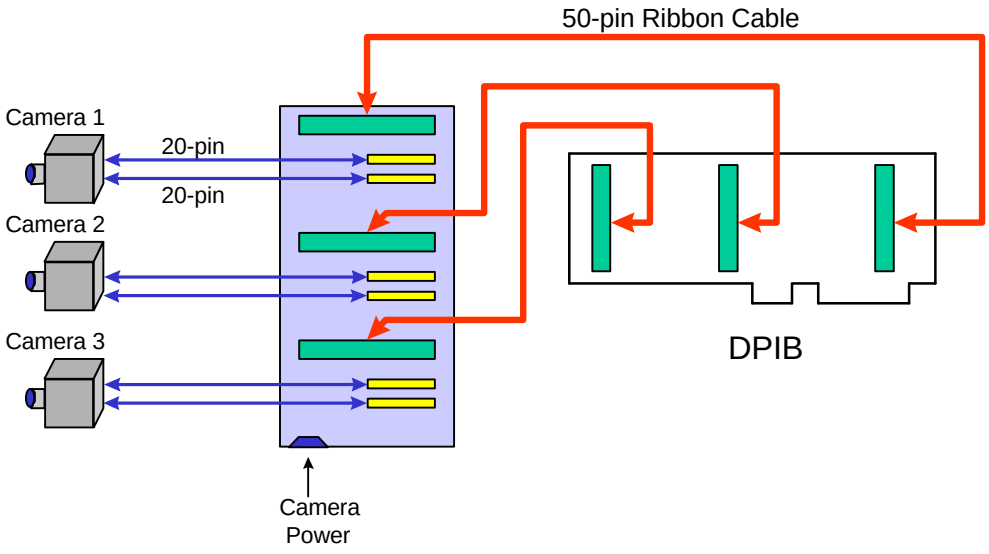


Figure 4.8 Break-out board example

The modular nature of the FPGA logic allows for easy modification of the existing design. The primary purpose of the DPIB is to multiplex data sources on a single output bus. The current design serves as an excellent basis for further development. A new design can simply replace the ASE2SUIT module with one configured to convert the new source data to the SUIT bus format. The rest of the design remains unchanged.

The re-configurable nature of the DPIB was recently explored in another research project. Dalsa 9200 series black and white cameras were used for image collection of hardwood lumber. These cameras use an 8-bit differential signal bus, with an additional four control bits. The signals are available on two 20-pin protected header connectors.

The DPIB was used with an external break-out board that accepted 50-pin ribbon cables from the DPIB and two 20-pin ribbon cables from each camera. These connectors were wired to move Dalsa pin assignments to the appropriate pins on the DPIB connectors. A new module, DALSA2SUIT (Appendix B) was created to convert Dalsa

data to SUIT bus format. The only change to the current design was the replacement of the ASE2SUIT with the DALSA2SUIT module.

Chapter 5. Software

This chapter describes the software developed for the prototype system. The Windows NT device drivers that provide access to the hardware, the system utilities and the user interface are presented. The source code for the software developed is included in Appendix C and Appendix D.

5.1 Overview

The software development effort is divided into three areas: *device drivers*, *graphical user interface* and *system utilities*.

Device drivers were developed to control the custom hardware. Windows NT is a sophisticated, multi-tasking operating system where applications are executed in protected mode. Functions commonly used to access hardware ports have no effect under Windows NT, as allowing an application control of system devices would jeopardize the stability of the operating system. Therefore, hardware ports can only be accessed through device drivers. The drivers developed here were written for the DPIB and MCPCI, but are very portable and can be used to control many other ISA or PCI devices. There is very limited literature available on developing Windows NT device drivers for PCI hardware, and the source code and documentation provided here will greatly reduce development time and effort for future programmers.

The graphical user interface (GUI) was developed as the front-end of the entire system. It is designed to accomplish the following tasks:

- automate the data collection process by controlling the AS&E and custom hardware,
- interface with the image processing software that performs materials characterization and explosives detection,

- analyze and combine the output of the processing software,
- present the results to the system operator in an easy to understand, graphical form, and
- alert the operator if explosives are detected in the luggage.

The GUI provides the operator with enough functions to assist in identifying the contents of the luggage, but eliminates screen clutter and repetitive user input by automating the collection process.

System utilities were written to access the custom hardware (program the FPGA chips, or collect data from the MCPCI into main memory), and also to display black and white or color images under the Windows operating system. These utilities can be executed either manually by the user, or through the graphical user interface. Configuration files or command line options are used to set run-time parameters for each application. Using separate utilities, instead of incorporating all the functions into a single program, reduces the code size of the GUI and assists in code maintenance. Furthermore, memory requirements are reduced, since the application is executed only when necessary, and is unloaded from memory when not in use. Finally, debugging and upgrading can be performed on the source code of the utility, rather than the source code of every application that, for example, programs the FPGA chips.

The functions necessary to access the device drivers were grouped into the *hardware.h* function library. The functions used to control the automated prototype system were grouped in *sensor.hpp*. Creating libraries of commonly used functions allows for code re-usability and maintenance, and also simplifies documentation.

5.2 Device Drivers

To allow access to the two custom hardware boards two device drivers were developed for this research project: PCIDMA.SYS, which is used with the MCPCI, and DPIB.SYS, which is used with the DPIB.

Any system level development for Windows NT requires the Windows NT Software Developer's Kit (SDK) [SDK96], and the Device Driver's Kit (DDK) [DDK96]. They are available through an annual subscription service from Microsoft Corporation, and are updated quarterly. These tools provide libraries and documentation for the development of system level drivers.

The drivers developed for the MCPCI and the DPIB are *kernel mode* drivers and bypass all operating system functions. They have full access to Windows NT Ring 0, the lowest level functions of a PC. Figure 5.1 shows a diagram of how hardware devices are accessed in Windows NT. The application, which runs in user mode (restricted access), passes information to the driver through an I/O Request Packet (IRP). The IRP contains the device driver path (name of device driver), and the I/O Control Code (IOCTL), which is used to identify the function that the driver must perform. Also included in the IRP are any values that will be passed to the device driver, such as the port address or data value. The IRP is passed to the NT I/O Manager through the *DeviceIoConrtol* function. The I/O Manager is part of the Windows NT kernel and handles device drivers as file objects that can be opened, read from or written to, and closed. The final level between the hardware and the NT kernel is the Hardware Abstraction Layer (HAL). The HAL exports routines that abstract platform-specific hardware details about caches, I/O buses, interrupts, etc., and provides an interface between the platform's hardware and the system software [DDK96]. The HAL communicates with the hardware device and returns any information to the device driver. That information is passed back to the application by the I/O Manager through the IRP. For a further analysis of the Windows NT device driver model, please refer to [DDK96].

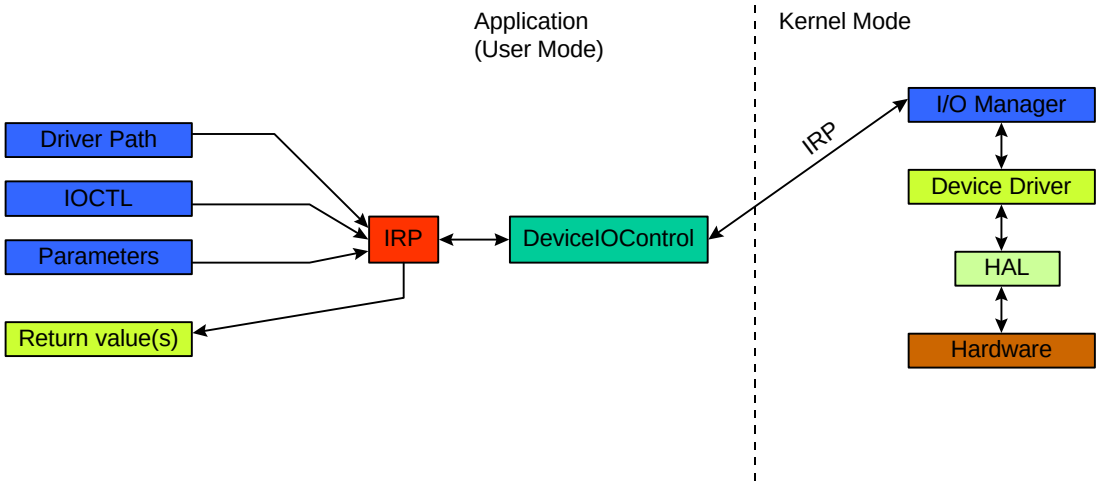


Figure 5.1 Hardware access through a device driver

5.2.1 Common driver functions

There are some functions that are common to all device drivers. These are used by the operating system during startup, and to dynamically load and unload a device driver. They are as follows:

- **DriverEntry:** this is the "main" function of the device driver and is executed upon startup. It initializes the driver and provides the operating system with a pointer to the device object data structure. It also informs the operating system of the location of the *Dispatch* routine. The return code of *DriverEntry* is used to determine whether the device driver was loaded successfully.
- **Dispatch:** when a request is made to a device driver by a user program, the operating system passes the request to the driver in the form of an IRP (I/O Request Packet) structure. It is the purpose of this routine to determine whether the IRP contains a valid request and, if so, execute the appropriate function to handle the request.
- **Unload:** this function is executed when a request is made to dynamically unload the driver from memory. Although the earlier releases of Windows NT (versions 3.5 and lower) required that the system be restarted to unload a

device driver, drivers can now be dynamically loaded through the "NET START" command, and unloaded with the "NET STOP" command. This routine is called when a "NET STOP" request is made. Here, any memory must be released, and the device driver object deleted.

5.2.2 Installing and starting a device driver

The executable file of a Windows NT device driver is appended the .SYS extension by the compiler. In order for the driver to be loaded when the operating system starts up, two requirements must be met:

a) the executable file must be placed in the Windows NT driver directory, commonly \Winnt\System32\Drivers, and
b) a registry entry must exist for the device driver. The entry must have the same name as the driver executable (without the .SYS extension) and should be located in \System\CurrentControlSet\Services\<DriverName>. This field must be placed in the HKEY_LOCAL_MACHINE key of the NT registry.

The Windows NT registry is a database of configuration entries for system and applications settings. It can be viewed using the *regedt32* utility. Manual changes to the registry can only be performed by the system administrator and are highly discouraged. A simple error in the registry hive can cause catastrophic system failure. Instead, an initialization (.INI) file is provided with each device driver. Changes to the registry are made with the *regini* utility, available with the SDK. Executing *regini <ini filename>* will update the system registry with the fields provided in the .ini file.

An initialization file contains certain values used by Windows NT to determine the type and parameters of the device driver. An explanation of these values is available in the DDK. The only .INI entry of interest in this discussion is the *Group* entry. Windows NT maintains a group order list, which determines the order in which device drivers are

loaded. This list is stored in the system registry, under \System\CurrentControlSet\Control\ServiceGroupOrder. The *Group* entry is an alphanumeric string identifying the group of the device driver. At system start-up, the order list is examined and the device drivers belonging to the first group are loaded. The other groups follow in order, until all drivers are loaded. The purpose of the group order list becomes apparent in Section 5.2.3.3.

5.2.3 PCIDMA.SYS - A device driver for the MCPCI

The MCPCI is a sophisticated bus master DMA device residing on the PCI bus. It is used to collect data from a single Zee bus connector and transfer the data to the host computer using direct memory access (DMA). The incoming data stream may contain information from up to six sources, identified by the channel select lines of the Zee bus (please refer to Chapter 4 for a more detailed analysis of the Zee bus). The MCPCI reconstructs the data and separates each channel, then stores the values on the on-board memory bank. When enough data is received, the MCPCI initiates a DMA transfer and places the image data from on-board memory onto system (PC) memory. This operation is performed transparent to the PC and without any CPU intervention. Each of the six incoming streams is placed on a separate block of system memory. The beginning address of the memory blocks is loaded on the MCPCI by collection utility.

As with any PCI device, the MCPCI is completely software configurable and supports Plug-n-Play configuration. The I/O address and interrupt line used by the hardware is determined by the operating system at boot time. The MCPCI also requires the allocation of a large DMA memory buffer under Windows NT. These features make the MCPCI driver very sophisticated and complex. At the time of this writing, there were no source code samples for a PCI bus master DMA device in the Windows NT DDK.

The MCPCI driver, named PCIDMA, supports the following functions: accessing IO ports for reading or writing, mapping and un-mapping the DMA buffer memory into user memory space, and also returning the physical address of the DMA buffer to the application. A chart illustrating these functions is shown in Figure 5.2. The IRP is passed

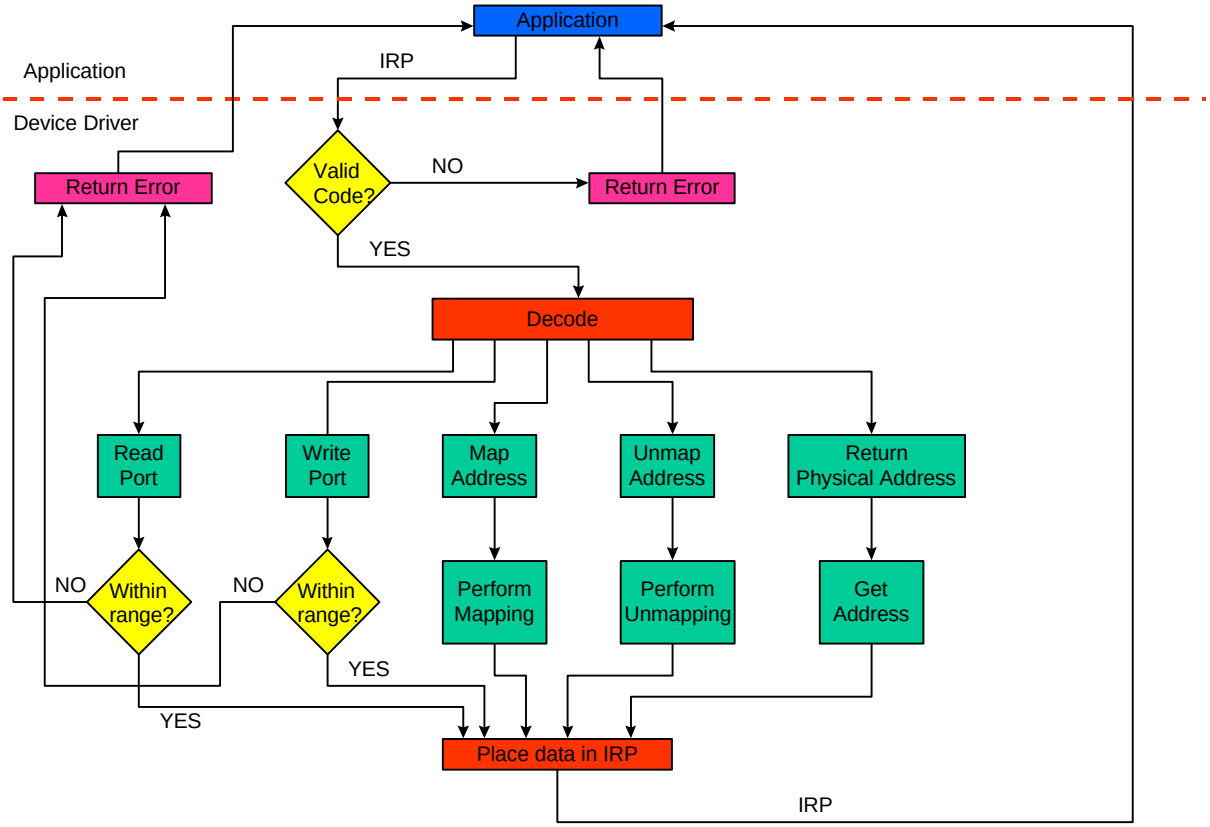


Figure 5.2 PCIDMA Function Chart

to the device driver and is checked to ensure that it contains a valid IOCTL. If so, the IOCTL is decoded and the appropriate function executed. The results are placed in the IRP. A read or write port operation also performs range checking, to ensure that the relative port address passed through the IRP is actually on the MCPCI. Any data is passed from the device driver to the IRP and returned to the application layer.

The PCIDMA driver consists of the following files:

- **pcidma.c:** the source code of the device driver
- **pcidma_dev.h:** contains the device driver data structure

- **pcidma_ioctl.h:** contains driver and device constants (vendor and device ID, and DMA buffer size), as well as I/O Control Codes (IOCTLs) for the device driver.

The source code of the PCIDMA driver is included in Appendix C.

5.2.3.1 PCIDMA function overview

This section provides an overview of some major PCIDMA driver functions. A flowchart illustrating the initialization process of the PCIDMA is shown in Figure 5.3. For further detail, please examine the source code or consult the NT DDK.

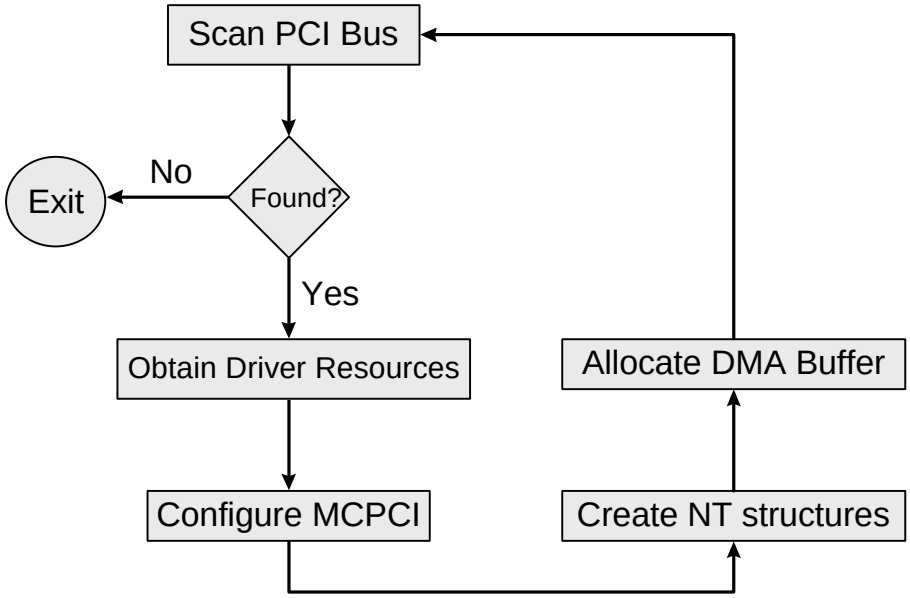


Figure 5.3 Flowchart of PCIDMA Initialization

ProbePci

This function scans the PCI bus and locates all the MCPCI boards. The board is identified by its unique vendor and device ID numbers, which are included in the pcidma_ioctl.h header file. If a MCPCI board is found, a device for it is created.

CreateDevice

Creates a Windows NT device for a MCPCI board. First, a symbolic link is created so that applications can access the device driver. Then, the Windows NT Hardware Abstraction Layer (HAL) is notified of the device, and the appropriate resources are reserved. This includes I/O ports and interrupts, as well as certain Windows NT constants for a bus master device. The former are defined in the device description structure (devDesc).

The DMA memory buffer is also allocated in this section. The maximum specified buffer size (included in pcidma_ioctl.h) is requested as a contiguous memory block. The MCPCI does not support scatter and gather, i.e. dividing the DMA buffer into smaller, distributed blocks of memory. If memory allocation succeeds, the physical address of the buffer is returned. Otherwise the function fails.

ServiceInterrupt

This routine is called when the interrupt assigned to the MCPCI occurs. Since the interrupt might actually be shared with other devices, this function should check to determine if the MCPCI device actually caused this interrupt. If the interrupt is not from the MCPCI, the function should simply exit. The system does not currently support interrupts, therefore this function simply returns to the caller without making any changes.

PPciDmaIoctlReadPort

This function reads a MCPCI port. It is called through the *Dispatch* routine following an application request. There are three types of read operations supported: byte, word and double word. The appropriate IOCTL code (READ_PORT_UCHAR, READ_PORT_USHORT, or READ_PORT_ULONG) determines which type read will occur. The port address passed to the function is actually the relative board address. The base address of the MCPCI is assigned by the *CreateDevice* routine and is not known to the application.

PPciDmaIoctlWritePort

Similar in operation to *PPciDmaIoctlReadPort,* this functions writes a value to an I/O port. A byte, word or double word can is written, depending on the IOCTL code used.

PPciDmaReturnMemoryInfo

Returns the DMA buffer physical address to the caller. This value is used to program the MCPCI address registers with the appropriate start address for each channel. From the device and the device driver point of view, the buffer is simply a large memory pool. A memory block is assigned to each MCPCI channel by the application. If some of the six MCPCI channels are not used, the memory pool can be divided into fewer, larger blocks. It rests with the application to ensure that the maximum DMA buffer size is not exceeded. If incorrect values are written to the MCPCI address registers, data might be transferred to memory that is in use by other programs or even the operating system, causing the system to crash.

PPciDmaMapBuffer

Maps the DMA buffer address returned by *PPciDmaReturnMemoryInfo* into user space. Windows NT is a protected operating system and the application memory space may or may not reside in actual RAM. The memory addresses seen by an application are "virtual" and are mapped to RAM by the operating system. The operation performed by this function allows the calling process to access the DMA buffer, by bringing the DMA buffer address space into application address space.

5.2.3.2 PCIDMA Installation

The initialization file for the PCIDMA.SYS driver is shown in Figure 5.4. It should be installed in the registry by running the REGINI utility. Please consult Section 5.2.3.3 for additional instructions on the installation of the PCIDMA driver.

```
\Registry\Machine\System\CurrentControlSet\Services\PciDma
    Type = REG_DWORD 0x00000001
    Start = REG_DWORD 0x00000001
    Group = PCIDMA Driver
    ErrorControl = REG_DWORD 0x00000001
```

Figure 5.4 PCIDMA.SYS initialization file

5.2.3.3 Memory allocation considerations

Certain applications using the MCPCI, such as multiple channel color image collection, require a large DMA buffer to store image data. DMA buffer sizes can easily reach 10-20 MBs. The Windows NT memory manager makes no effort to "create" a contiguous memory block. If an area of the requested size does not exist at the time the driver is loaded, the device driver will fail.

The only solution to locking a large memory block is to request it early during the boot sequence, when most device drivers and no applications have yet been loaded. Windows NT allows the user to specify the order in which device drivers will be started through the ServiceGroupOrder list, maintained in the registry (please refer to Section 5.2.2). As shown in Figure 5.4, the PCIDMA device driver belongs to the "PCIDMA Driver" group. This group does exist on a typical Windows NT operating system and must be added to the group order list.

The procedure for adding the PCIDMA driver group to the existing execution order requires extreme caution and can only be performed by the system administrator, or a user with administrative privileges. The steps to this procedure are:

1) Run the regedt32 utility and open the HKEY_LOCAL_MACHINE hive.
2) Descend to the \System\CurrentControlSet\Control\ServiceGroupOrder field.
3) Edit the entry value. It is a text string containing all the device driver group names. The "PCIDMA Driver" line should be added right after the "Event

Log" driver (because a failure to load the device driver can only show up in the event log if this driver is started) and before the network drivers. The registry can then be saved and the system restarted.

5.2.4 DPIB.SYS - A device driver for the DPIB

The DPIB is based on the sample source code for a generic ISA device driver (genport.h) provided in the Microsoft DDK. The DDK licensing agreement allows developers to modify the samples provided and distribute the executable freely.

A chart illustrating each DPIB driver function is shown in Figure 5.5. The driver only supports reading from or writing to an ISA port. The DPIB does not use interrupts

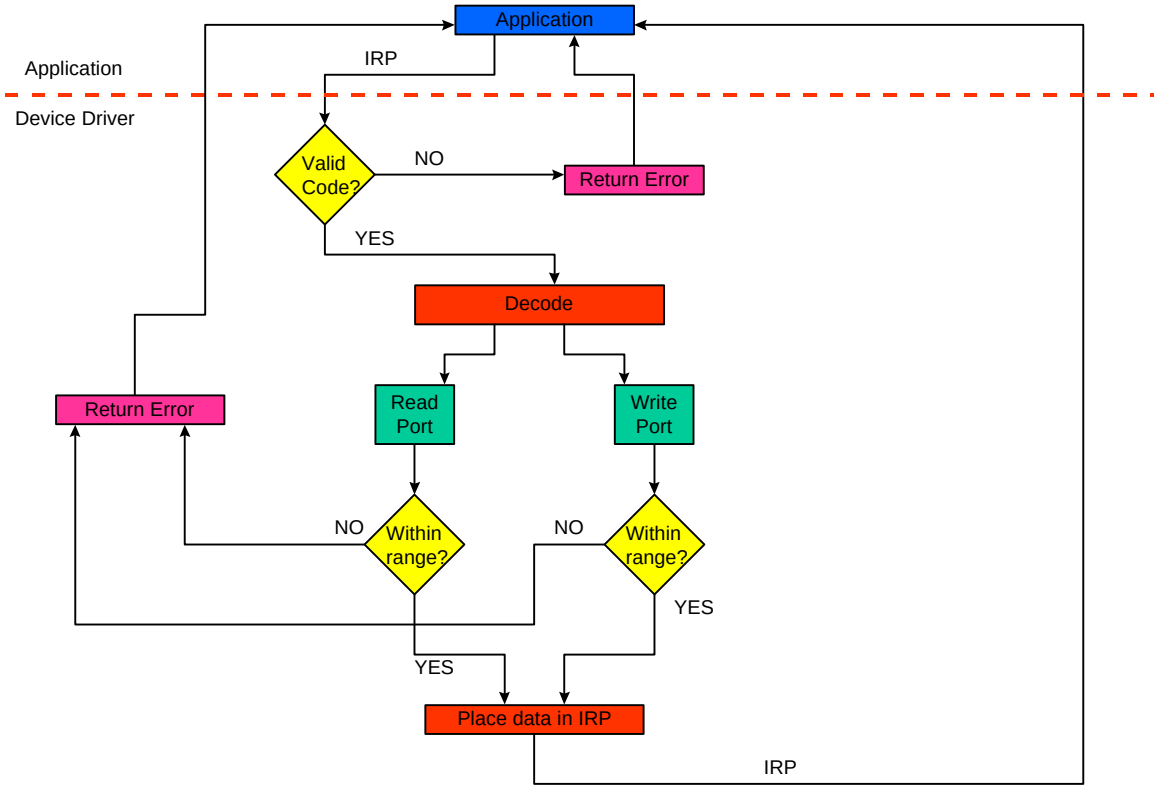


Figure 5.5 DPIB Function Chart

or DMA, therefore reducing the complexity of the DPIB driver. An IRP is passed from the application to the device driver and is first checked to ensure that the IOCTL is valid. If it is, the read or write function is called, depending on the IOCTL. Range checking is

performed in both functions, to ensure that the ISA address is within the DPIB. The data and a return code, indicating success or failure of the function, are placed in the IRP and are returned to the application.

The files used for the DPIB driver are as follows:

- **dpib.c:** the source code of the device driver
- **dpib.h:** an include file that contains the device name, default base port and address range, and the device structure.
- **dpib_ioctl.h:** contains the IOCTL codes for the DPIB.

The source code for the DPIB driver is also included in Appendix C.

5.2.4.1 DPIB function overview

DriverEntry

This *DriverEntry* routine differs from most, because it also notifies the HAL of the device properties and IO address. The DPIB IO address is first read from the device driver entry in the system registry. If there is no such entry, the default hard-coded value is used. If the address requested is in use by another device, the driver is not loaded.

Although the EISA bus is backwards compatible with ISA peripherals, Windows NT does distinguish between devices installed on EISA and ISA buses. Therefore, if the DPIB is installed in an EISA bus, the device driver will be unable to access the hardware. To operate the DPIB on an EISA system, the *InterfaceType* value in the *ResourceList* structure must be changed from “Isa” to “Eisa”. The source code must then be re-compiled and the new executable installed on the system.

DpibIoctlReadPort

Reads a DPIB port and returns the value to the IRP. A byte, word, or double word may be accessed. This function performs boundary checking to ensure that the address requested is actually a DPIB address. If the requested relative address exceeds the port count of the DPIB (defined through the registry, or set to 3 by default), an error code is returned.

DpibIoctlWritePort

Write to a DPIB port, similar to *DpibIoctlReadPort*. Boundary checking is also performed for a write operation.

5.2.4.2 DPIB.SYS installation

To install the DPIB driver, execute REGINI and provide the initialization file shown in Figure 5.6.

The DPIB driver allows the user to set the base address and port count of the hardware through the registry. The two keys are available in the *Parameters* field of the DPIB registry entry and may be modified using the REGEDT32 registry editor. Using address ranges that are not handled by the DPIB can cause hardware conflicts or a system crash.

```
\Registry\Machine\System\CurrentControlSet\Services\Dpib
    Type = REG_DWORD 0x00000001
    Start = REG_DWORD 0x00000002
    Group = Extended Base
    ErrorControl = REG_DWORD 0x00000001
    Parameters
        IoPortAddress = REG_DWORD 0x00000304
        IoPortCount = REG_DWORD 0x00000003
```

Figure 5.6 DPIB.SYS initialization file

5.3 Software Libraries

In order to assist in the further development of the prototype system, a set of libraries containing commonly used functions has been created. The *hardware.h* library contains functions used with the MCPCI, DPIB or MORRPH device drivers. The *sensor.hpp* library includes functions used to control the prototype system components, such as the conveyor belt, and the x-ray and copper filter controller. The source code for these libraries is included in Appendix D.

5.3.1 HARDWARE.H - a library for device driver access

This library contains functions to access the DPIB, MORRPH and PCIDMA drivers. Each driver has a unique name, known as the device path. The device paths are defined in *hardware.h* and are used to open a specific device driver. The IOCTL codes for each driver are also provided in *hardware.h.*

A description of individual library functions follows. All functions return STATUS_SUCCESS or STATUS_FAILURE upon exit. For more information, please refer to the source code (Appendix D).

OpenDriverHandle

This function must be called before using any device driver. It opens a handle to the device through which subsequent requests will be processed. Windows NT accesses a device driver similarly to a file: it must be opened first, and then can be written to or read from. Prior to exiting an application, all open handles must be freed using the *CloseHandle* function.

WritePort

Writes a byte to a device IO port. The port used is the relative device port, as applications are not allowed access to absolute hardware addresses. The IOCTL passed to the function should match the code used for a UCHAR write to the device driver.

These codes are found in the _ioctl include files for each device driver and are included in *hardware.h*.

WritePortDouble

Writes a double word (32-bits) value to an IO port. Similar to *WritePort*.

ReadPort

Reads a byte from an IO port. The IOCTL passed must match the UCHAR read code used with the device driver.

ReadPortDouble

Similar to *ReadPort*, but reads a double word (32-bit) from the hardware.

MapPciDmaBuffer

Maps the DMA buffer memory space into user space, to allow the application access to the image data. There is no need to allocate a separate buffer when using the DMA board; the buffer allocated for storage by the device driver will be used. Writing to the mapped memory will also alter the contents of the DMA buffer.

UnMapPciDmaBuffer

Executed before an application exits, this function un-maps the previously mapped DMA buffer. Memory is not actually freed, since the DMA buffer is allocated by the device driver and not the application, but a handle to the memory is freed. There is a limited number of handles available for applications and failing to un-map the memory can exhaust the handle supply, rendering the operating system inoperable.

GetPciDmaAddress

Returns the physical address of the DMA buffer. This value can not be used to access the memory pool. It is only used to program the MCPCI address registers with a

pointer to where image data should be stored. The application must divide the memory pool into blocks for each MCPCI channel.

5.3.2 SENSOR.HPP - a library of prototype system control functions

The *sensor.hpp* library includes functions for controlling the prototype system. The x-ray controller, copper filter motor controller, infrared sensors and conveyor belt are controlled by functions available in this library.

The DPIB port addresses, as well as the configuration data for the serial ports are included in the beginning of the file. For easy reference, changes should be made to the constants in the #define statements, rather than be hard-coded in specific functions.

Any functions that write to the x-ray or the filter motor controller use the serial port for communication. Neither controller provides handshake lines; they only echo the transmitted data back to the sender. To avoid the complexity of reading and parsing the returned data, a delay is inserted after the transmission of each character. This allows the controllers time to accept and process the data. If a delay is not used, data dropout will occur.

A description of the functions available in *sensor.hpp* follows. For more information, please refer to the source code in Appendix D.

WaitSeconds

Delays program execution by the specified time, in seconds. The process is not actually asleep when this function is called, but processor usage should be negligible.

WaitTSeconds

Delays program execution by the specified time, in tenths of a second. Similar to *WaitSeconds*.

MoveBeltForward

Starts the conveyor belt in the forward direction. A handle to the DPIB device driver must be available.

MoveBeltReverse

Starts the conveyor belt in the reverse direction by writing to the DPIB.

StopBelt

Stops the conveyor belt. This function must be executed before the conveyor belt direction is changed.

BreakFrontSensor

Waits until the front infrared sensor is interrupted. A valid handle to the DPIB is required.

BreakRearSensor

Waits until the rear infrared sensor is interrupted.

UnBreakFrontSensor

Assuming an object is interrupting the front infrared sensor, this function will wait until the object is removed. If the sensor path is clear, this function returns immediately.

UnBreakRearSensor

Similar to *UnBreakFrontSensor*. Waits for the rear sensor path to be cleared of any obstructions.

SetUpXrayController

Configures the serial port connected to the serial port controller (usually COM2). The baud rate, parity and stop bits are set. This function returns a FALSE value if the serial port could not be configured.

The x-ray controller can only be remotely controlled if it is placed in mode 800 from the operator control panel. For further information on setting the operating mode, please refer to the controller documentation [LUM95].

SetUpMotorController

Configures the serial port connected to the copper filter motor controller (presently COM1).

SetUpDPIB

Initializes the DPIB function generator with the appropriate timing values. This function should be executed only if the design loaded in the DPIB supports programmable timing signal generation.

ProgMotorController

Programs the filter motor controller. The BASIC program is downloaded to the VELMEX controller and then executed. The controller is first placed in remote access mode by this function.

SetKV75, SetKV150

Set the x-ray voltage to 75 and 150 KV, respectively. These are the two energy levels used on the prototype system.

SetmA300

Set the x-ray current to 300mA. This is the only current setting used.

TurnXrayON, TurnXrayOFF

Turn the x-ray source ON and OFF respectively. The operator key must be inserted in the x-ray controller and turned to position 3.

LowerFilter

Lowers the copper filter in front of the x-ray source. Unfortunately, the Velmex motor controller can not provide information on the position of the motor. For this function to operate, the filter must be in the low position when the controller is configured. This sets the origin angle (position of zero degrees) to the lowest point of the filter path.

RaiseFilter

Raises the copper filter and removes it from the field of view. The filter is equipped with safety switches that will stop the motor controller if the angle requested exceeds the maximum range. This can occur if the filter is not completely lowered when the system is started.

SetUpCorrVal

Writes the data compensation values to the DPIB.

5.4 Utilities

A set of utilities was developed to control the prototype system and the custom hardware. These can be executed as stand-alone programs, or through the graphical user interface. The source code for these utilities is shown in Appendix E.

5.4.1 PROGALL

PROGALL programs the FPGA chips on the DPIB and MCPCI. A .POD file is used to program an FPGA, and is obtained from the .MCS file output by the Xilinx XACT software [INT90]. The two files used by PROGALL are named PCIDMA.POD and DPIB.POD. There is no method to verify that the FPGA was programmed successfully; it rests with the hardware designer to incorporate test circuitry in the FPGA logic.

5.4.2 COLPUL

Used to initiate and control a DMA transfer from the MCPCI, this is a Windows NT port of a DOS collection utility written by Paul LaCasse. A newer version, named COLPUL-SILENT, was created to eliminate user input and automate the collection process.

Data collection is initiated by a "start" command issued to the MCPCI. The program then probes the hardware to determine if the requested number of lines has been transferred, and issues a "stop" command when this occurs. If a time-out period has elapsed and collection has not yet been completed, the program exits and issues an error code.

Since the MCPCI will only stop collection after an appropriate command is received, but not when the requested number of lines is transferred, COLPUL should never be abnormally terminated by the user. Doing so can cause memory outside the DMA buffer to be overwritten, resulting in a system crash.

COLPUL is configured for six channel operation, and divides the DMA buffer into six regions. The size of each region is determined by the requested width and length of each image, derived from the COLPUL configuration file (see Section 5.4.2.1). The output is stored in six files, named "one.img" through "six.img" using the ELAS file format (see Section 5.4.2.2).

5.4.2.1 COLPUL Configuration File Format

A configuration file (PCIDMA.CFG) is used to determine the number of lines to collect, and the resolution of each line. A sample configuration file is shown in Figure 5.7.

```
NUMFRAMES = 200
WIDTH1 = 1796
WIDTH2 = 896
WIDTH3 = 896
STARTPIX = 0
CHANNELS = 7
```

Figure 5.7 Sample COLPUL configuration file (PCIDMA.CFG)

The NUMFRAMES entry determines the length (number of lines) of the collected image, whereas the following three WIDTH entries are the image width for the first three channels. The remaining three channels are also of width WIDTH3. STARTPIX is used to ignore a number of pixels at the beginning of a line and is commonly set when the first few pixels fall outside the region of interest. Finally, CHANNELS is a 6-bit value that enables a DMA channel. Setting a bit 0 enables channel 1 for collection, setting bit 1 enables channel 2, and so on.

5.4.2.2 ELAS Image File Format

The ELAS file format was developed by the National Aeronautics and Space Administration (NASA) to store satellite images [ELA89]. It is suitable for image processing applications because there is no compression or image quality loss. It has been established as a standard in the Spatial Data Analysis Laboratory and is used with all data collection devices.

The length of the image header is equal to the width of a line of data, but must be at least 28 bytes wide. This produces seven 8-bit values, which provide image size information, as shown in Table 5.1. Certain fields defined by the ELAS format are not

Table 5.1 ELAS header description

Bytes	Name	Description
0-3		Always set to 0.
4-7		Always set to 0.
8-11		Always set to 1.
12-15	numframes	Length of image (number of lines)
16-19		Always set to 1.
20-23	width	Width of image (pixels per line)
24-27	numchan	Number of channels, 1=BW, 3=color

used and are assigned a constant value. The *numframes* and *width* fields indicate the size of the image, whereas *numchan* is the number of channels. A black and white image has one channel of data, whereas a true-color, 24-bit image uses three channels. Color image data is arranged in lines: a line of red, followed by a line of green, then a line of blue values.

5.4.3 EDISP

EDISP was developed to display color and black and white ELAS images under the Windows operating system. It accepts the image file name as a command line parameter and displays the image on the desktop. If the file is not a valid ELAS file, the user is notified and the program terminates.

After some initialization tasks required by Windows, the ELAS file is read and processed. To take advantage of hardware and software acceleration functions provided for image display, the file is converted to Windows Device Independent Bitmap (DIB) format. A DIB arranges pixel values as Blue-Green-Red, instead of RGB, and requires that a line be padded to 32-bits. Furthermore, each pixel is handled as a triplet of BGR values, whereas the ELAS format stores the entire line in red, then green and then blue.

The DIB image is then displayed using the BitBlt function. The same function is called when the window must be redrawn, for example when it is moved to a different location on the desktop. BitBlt is designed to take advantage of display driver hardware and accelerate the display of DIBs. If the DIB is to be stretched to fill a box bigger than it's original size, the StretchBitBlt function should be used instead.

5.5 Galaxie - Graphical User Interface

Galaxie is the main interface between the system operator and the hardware. It is designed to provide automated collection and processing, while also allowing the user some control over the system. Galaxie was developed using Borland C++ 5.02 and is packaged as a Win32 project [BOR96]. The source code shown in Appendix E is only a portion of the overall application code, which includes a resource file for the GUI objects.

Upon start-up, Galaxie initializes the motor and x-ray controller, and the DPIB. It also raises the copper filter and turns on the x-ray source. The main dialog box is then presented and the application awaits user input. The user can select SCAN, to start image collection for a new bag, DISPLAY, to display previously collected images, with or without processing results, and EXIT, to exit the software. A set of radio buttons is also provided to determine the image type that will be displayed. The high or low energy transmission, and the back scatter and forward scatter images can be displayed. An overlap selector is also provided: when ON, the processing results will be displayed and the image is color coded to show explosives, detonators and thick objects. When OFF, the raw x-ray image is displayed. Finally, a Status box is provided to inform the user of the stage of an operation in progress.

When SCAN is selected, the x-ray source is set to low energy and turned on. The conveyor belt is started and the program awaits for the front sensor to be broken. As soon as the beam is interrupted, COLPUL-SILENT is executed in the background to

collect image data. Galaxie stops the conveyor belt when the luggage exits the field of view (the rear sensor is broken and then cleared) and waits for the collection utility to finish. The x-ray is then set to high energy, the copper filter lowered and the luggage is reversed to its original position, before the front infrared sensor. The high energy x-ray images are then collected and the luggage is allowed to exit the tunnel, since no more images are necessary. The x-ray energy is then lowered, the copper filter raised and the conveyor belt stopped.

Next, the collected images are processed. First, the chopper wheel slots are separated. Because the system is configured to start a new line at each **WHLRST** pulse, the images collected by COLPUL contain all four chopper wheel data in one line. *Imgconv* is used to separate each slot in the ELAS file. The processing then continues in two stages: first, *process1.bat* is executed to rotate, crop, resize and shade correct the raw image. Shading correction is performed in software to further improve image quality. The low energy transmission image is displayed when pre-processing is done and remains on the desktop while the actual explosive and detonator detection algorithms execute in the background. This is performed through *process2.bat*. When processing is finished, the status display is updated and the user can view the processed results, or scan new luggage.

Highlighting of dangerous or suspicious regions is performed in *Galaxie*, using output data provided by the image processing algorithms. Three binary data files (defined as EXPLOSIVEBITMAP, DETONATORBITMAP, and THICKNESSBITMAP in galaxie.hpp) are overlapped on the original image. If a bit in the binary files is set, the original pixel is substituted with a different color to indicate danger. A priority list is established to highlight pixels that have been marked in two or three binary files. The priority list, with the corresponding color coding is as follows:

- **explosives** receive highest priority and are marked red.
- **detonators** are marked blue

- **thick** areas are marked yellow.

Figure 5.8 shows a flowchart of the operation of the graphical user interface during the collection process.

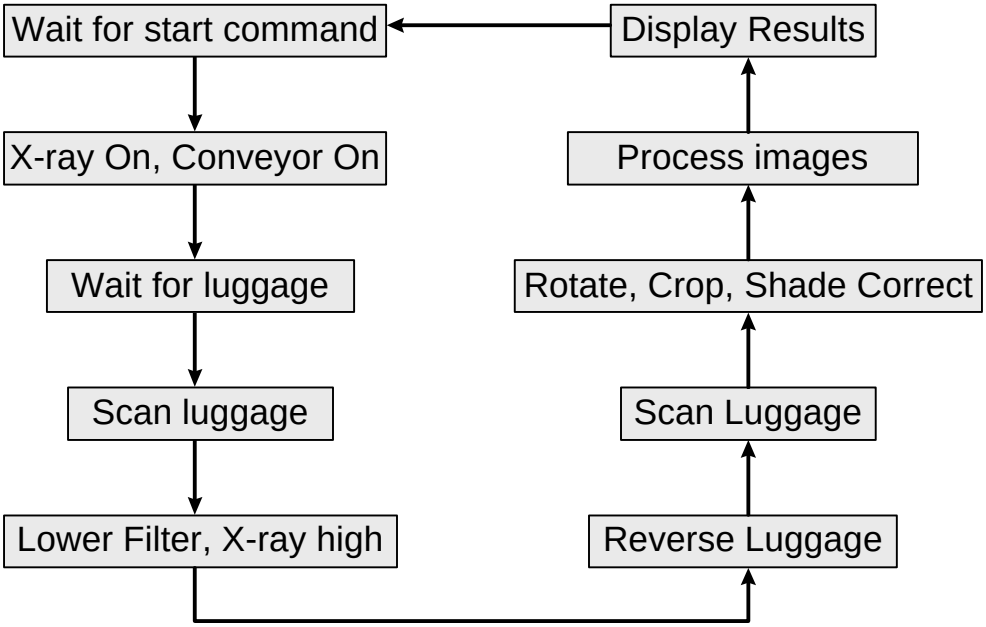


Figure 5.8 Flowchart of Galaxie operation

Chapter 6. Results

This chapter contains results obtained from the prototype system. The images displayed were collected using the hardware and software discussed in the previous chapters. The three different sensor technologies are demonstrated through transmission, back scatter and forward scatter images, collected at dual energy levels (75KV and 150KV). The difference in image quality with and without the copper filter is also illustrated.

The images in Figure 6.1 through Figure 6.4 are raw, uncompensated images collected with the DPIB. They have been cropped and rotated using Adobe Photoshop 3.0. Images collected with the DPIB are rotated by 90 degrees and are of fixed length, containing static background information that is removed before processing.

Figure 6.1 shows the low energy transmission image obtained from a typical piece of luggage. This luggage contained mainly articles of clothing, a shoe and a package of chocolate squares used for testing. A belt buckle is also shown and appears dark, as it is made of metal. Figure 6.2 shows two high energy transmission images of the same luggage. Image 6.2a was collected without the copper filter. It is saturated and has a much narrower histogram of pixel values, providing less useful information than image 6.2b. The latter was collected after the insertion of the copper filter and is clearer than image 6.2a, especially in the upper portion of the luggage. All three images are 324x286 pixels. The actual image height of DPIB transmission images is 450 pixels, but a portion of the image has been cropped as it only contained a dark background.

Figure 6.3 shows the back scatter images of the same luggage. Image (a) is the low energy image, and images (b) and (c) are the high energy images. Using the copper filter improves image quality dramatically in this situation. The chocolate squares, which are placed in the middle of the bag and were visible in the transmission image, stand out in image (c). Another object (a book), which also appeared in the transmission images as a

dark rectangular region, is not visible in any back scatter image, indicating that it is on the other side of the bag and should therefore appear in the forward scatter images.

Figure 6.4 shows the low and high energy forward scatter images. Again, using the copper filter improved image quality. The chocolate bars appear in image (c), as does the book, indicating that it is on the side of the bag that is facing the forward scatter detectors. The back scatter and forward scatter images are both 324x150 pixels. The original image height was 225 pixels, half the height of a transmission image.

Figure 6.5 shows a screen capture of Galaxie, the graphical user interface, taken immediately after Galaxie was started. The status bar indicates the current system state and is updated during image collection and display. The Scan button is used to start the collection sequence. When Scan is pushed, Galaxie waits for a luggage to enter the x-ray tunnel and then collects all the necessary images through the DPIB and MCPCI. The image processing software is then executed and an output displayed. The user can select which image to view by using the *Image Source* and *Energy Source* areas. Pressing the Display button shows the selected image on the screen.

Figure 6.6 is a screen capture of Galaxie after the processing software has been executed. The results are overlapped on the image selected by the user, in this case the back scatter, low energy image. The Overlap button can be used to toggle between the raw and processed images, and a legend is provided for the meaning of each color. The areas of this luggage painted in red contain honey and chocolate, two substances that were used extensively to evaluate the system before x-ray simulants were supplied.

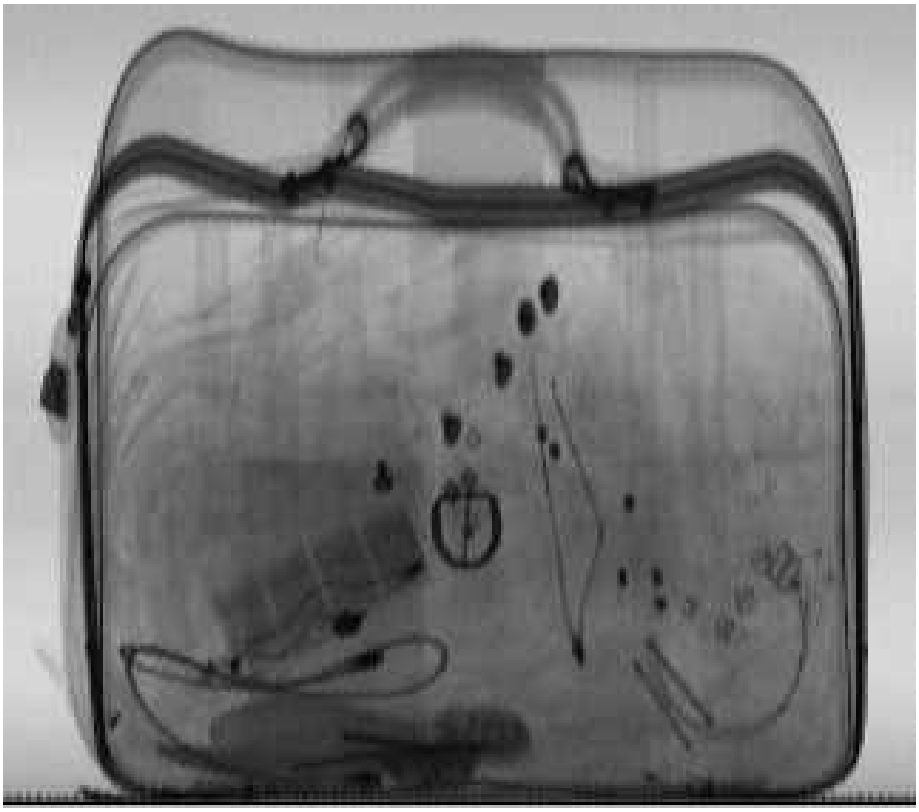
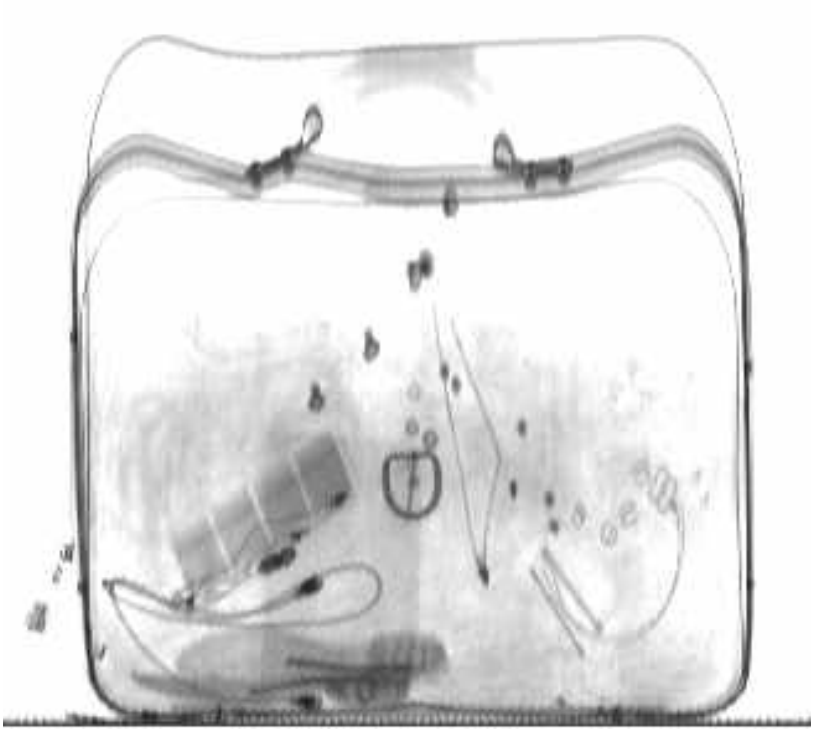
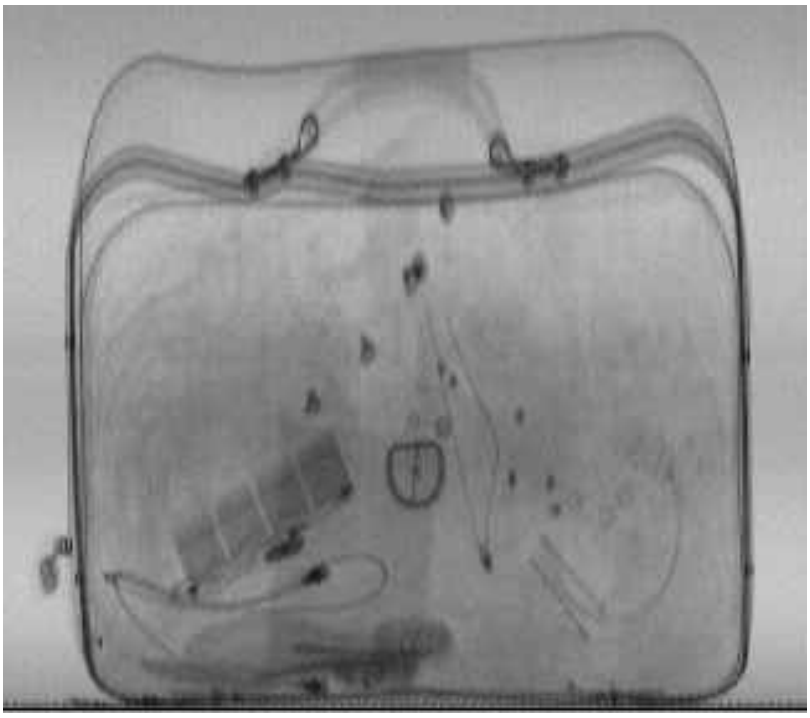


Figure 6.1 Transmission image at 75KV

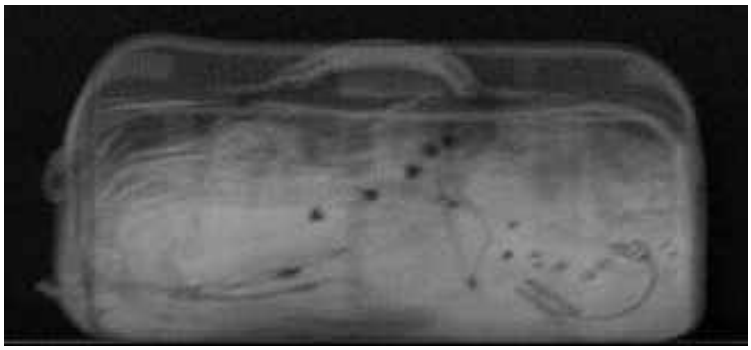


(a)

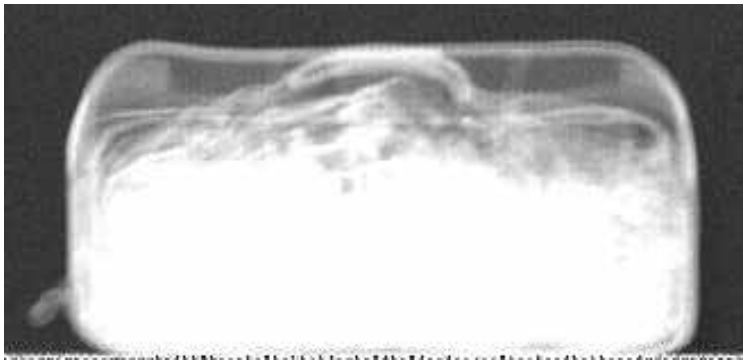


(b)

Figure 6.2 Transmission image at 150KV (a) without filter, and (b) with filter



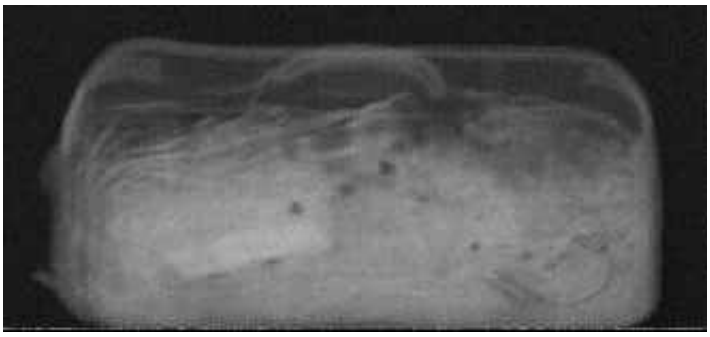
(a)



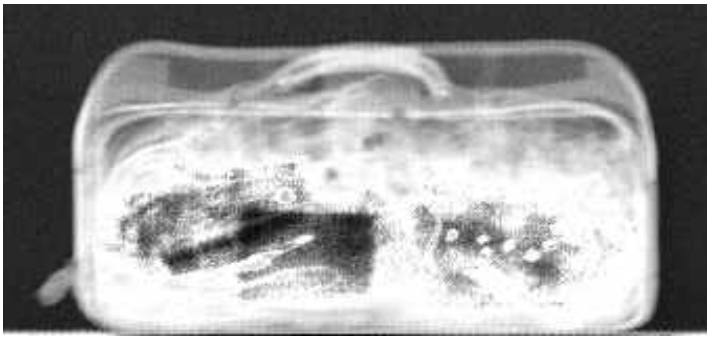
(b)

(c)

Figure 6.3 Back scatter images at (a) 75 KV, (b) 150KV without filter, and (c) 150KV with filter



(a)



(b)

(c)

Figure 6.4 Forward scatter images at (a) 75KV, (b) 150KV without filter, (c) 150KV with filter

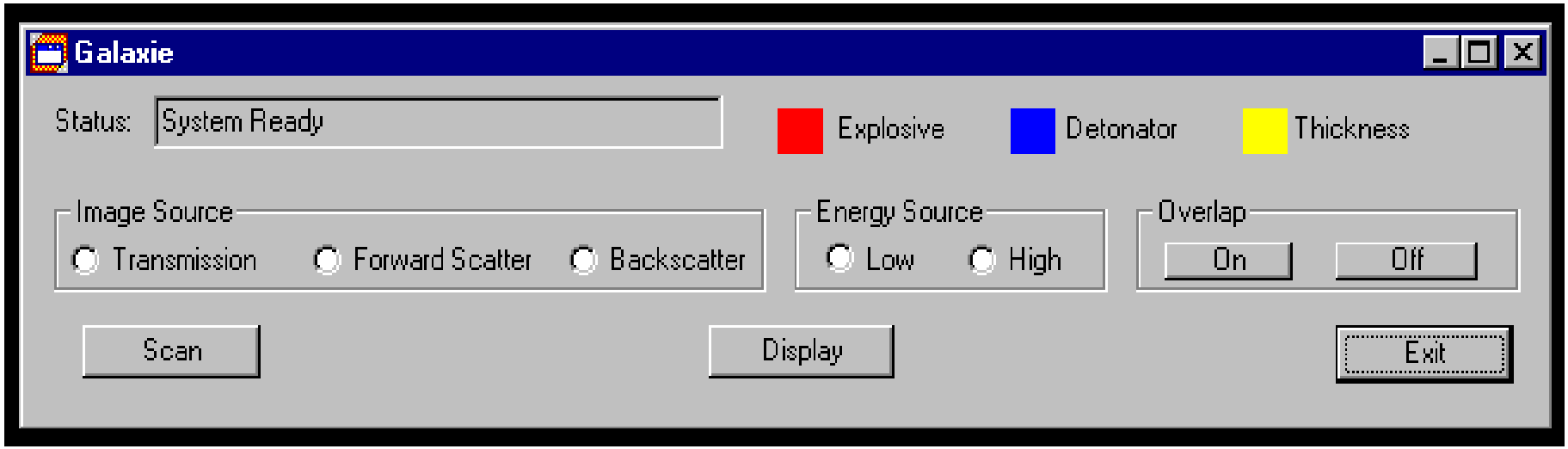


Figure 6.5 Galaxie screen capture at start-up

Figure 6.6 Galaxie screen capture with processed image

Chapter 7. Future Developments

Although every effort has been made to produce a complete system, this research effort focused on the development of a prototype. New algorithms and ideas are continuously being examined and may be implemented in the future. The purpose of this chapter is to examine certain concepts that can improve the operation of the original system and discuss how they can be incorporated with the existing hardware and software.

7.1 Orthogonal x-ray view

A problem with typical x-ray imaging systems is that they project a three dimensional object into a two dimensional image: any perception of depth is lost. This restriction has been explored in the past by terrorists, to conceal explosives in areas that an operator could not detect. Under certain circumstances, it might be possible to mislead an automatic explosives detection system into missing the presence of explosives by carefully placing them between other denser objects [KIT96].

A solution to this problem is to add another x-ray detector to obtain depth information. This detector provides an orthogonal x-ray view and can be used to better determine the thickness of the material. Furthermore, information will also be obtained about the distance of each object from the scatter detector. Distance measurements can be used to better interpret scatter image data [CON96].

Adding an orthogonal view to the current system requires some modifications. Assuming that the new detector can indeed be positioned in the x-ray tunnel, it will present another image source that must be controlled and collected. Therefore, a pre-amplifier digitizing board must be built. Although another AS&E pre-amplifier board can be used, developing a custom board with 12-bit resolution is greatly preferred. The new board must use the same AS&E interface, with the extensions discussed in Chapter 4 for 12-bit transfers. The DPIB logic can then remain unchanged.

The new image data must somehow be transferred to the host computer. The DPIB, due to ISA size limitations, can only support three data sources. Another DPIB board must therefore be used. This presents an additional problem of transferring data from two sources to the MCPCI, which only supports one Zee bus connection. A MORRPH board can be used to multiplex the two Zee bus signals onto a single output bus [DRA95a]. The output of the MORRPH can be directly connected to the MCPCI.

Adding an orthogonal view will raise the system cost, since new hardware must be installed. Also, luggage processing time will increase due to the increase in the amount of available data. However, the addition of the orthogonal view will improve the detection capabilities of the system and reduce the false alarm rate.

7.2 Active Control

In the current system configuration luggage is scanned once at each energy level. The same process is followed for every bag, regardless of whether it contains a threat or is completely innocent. Active control is the ability to dynamically scan certain areas of the luggage in detail, if the detection algorithms indicate a suspicious area. The luggage is held in the tunnel until a final decision is made. If it must be re-scanned, the conveyor belt is reversed and the luggage is returned to the front infrared sensor. The x-ray settings, such as voltage, current and integration time, as well as the conveyor belt speed are altered. Slowing down the conveyor belt increases the horizontal resolution of the image; increasing the integration time results in lower vertical resolution, but improves image quality, because the number of photons collected in the photo-multiplier is increased.

The current system can support most features required for active control with only minor software modifications. Image processing algorithms must be developed to register the different resolution images, but the control software only requires few changes. A communications protocol must also be developed between the graphical user interface and

the image processing algorithms to indicate when luggage must be re-scanned and at what settings. However, the major change in a system using active control will be a new motor controller. The conveyor belt controller used on the AS&E system provides only manual speed adjustment through a potentiometer. Although the controller can be researched and modified to allow computer control, the effort involved in such as activity is tremendous. Furthermore, the interface developed will be proprietary and will most likely be accessible only through the DPIB. A new motor controller should be used that allows settings to be altered through a serial communications port. This approach provides greater flexibility, a standardized interface and is more durable.

7.3 DPIB modifications

Currently, the DPIB exists only on wire-wrap boards. Before a printed circuit board is manufactured, the changes discussed here are recommended to improve the modularity of the hardware.

First, the 9-pin sensor signal connector can be replaced with a wider 15 or 25 pin connector. This will allow for more TTL level signals to enter (or exit) the DPIB. Clamping diodes and TTL buffers should be used with each pin, and some pins should use relays for isolation from external devices.

The application range of the DPIB can also be increased by substituting the 28-pin DIP sockets used for a memory ICs with support sockets [DRA95b]. A support socket provides a power and ground bus and a undefined array of pins that connect to the FPGA. The power and ground connections are made by physically adding a jumper from the appropriate bus to the pin. Pin sockets are placed in two columns and are spaced to hold a DIP sized IC. The advantages of this approach is that any size IC in a DIP package can be used, since the signal direction and function of the pins is defined in the FPGA. Therefore, the current prototype system design can still use the existing memory ICs, but a

different DPIB design can use another IC, such as a multiplier or a FIFO, by simply placing the IC on the socket and re-defining the FPGA pin assignment.

Chapter 8. Conclusions

The goal of the research project leading to this thesis was the development of a prototype inspection system for airport luggage. This system will aid in the creation of new image processing algorithms, as well as advanced materials characterization techniques. It will allow automatic detection of explosives and detonators in passenger luggage, and will serve as a basis for the development of a commercial system.

The prototype system is based on an American Science & Engineering 101ZZ airport security system. The 101ZZ was analyzed and modified to fit the purposes of the research project. The operation of the data collection and control hardware was documented. Features were added for computer control of all major functions of the system and all obsolete system hardware was removed. Only a minimum of the system electronics was maintained to provide raw image information.

Hardware was researched to interface to the existing system and obtain image data. A Differential Pair Interface Board (DPIB) was developed to connect to the system electronics and obtain raw images. The DPIB multiplexes images from three different sensors and outputs data through a high-speed bus to other image processing hardware. It performs synchronization of all data collection activities and controls the operation of the AS&E system. A library of hardware modules was also developed for the purposes of this research activity. These modules are implemented on the DPIB through the MORRPH Development System (MDS) and can interface to an existing library of components, to reduce development time. Modules specific to the AS&E system design, but also general purpose modules are provided to assist future hardware designers.

The DPIB was developed using Field Programmable Gate Arrays (FPGAs) and is highly re-configurable. It can be used to interface to most data sources using a differential pair bus, eliminating the need for additional hardware. The flexibility of the data connectors supports an 8-bit to 12-bit bi-directional data bus, while also providing signals

for the control of the external device. Furthermore, two or three data connectors can be combined to interface with a single device, allowing 32-bit data transfers.

Software was developed to control the prototype system and provide a user interface, but also to provide access to hardware devices. A sophisticated device driver for the Multiple Channel PCI board (MCPCI) was written for the Windows NT operating system. The device driver can fully configure and control the hardware. Plug-n-play capabilities are provided to automatically configure the MCPCI without user intervention. The driver can allocate and handle very large DMA buffers despite operating system limitations, and provides a robust interface to the MCPCI. It was also written to serve as a general purpose example of Windows NT device drivers for PCI hardware and can be used as a sample for development of other Windows NT drivers. A software library was created to group all functions used to access device drivers. The library has been documented and provides a common, simple interface to the hardware device, avoiding the confusion of using standard operating system functions.

A utility for the display of ELAS images on a PC running the Windows 95 or Windows NT operating system was also developed. This utility can display monochrome or color images, and takes advantage of hardware acceleration techniques and intelligent memory management to quickly draw and re-draw images. Utilities developed by other members of the Spatial Data Analysis Laboratory were modified and ported to operate under Windows NT. These utilities are used widely in any software development effort in the SDA Lab and have shifted the development platform from MS-DOS to the advanced environment of Windows NT.

Moreover, a graphical user interface was developed as the front-end of the prototype system. It is used to automate the data collection task and incorporate all the image processing and system control software in a single package. Another software library was also developed, to provide access to all the functions necessary for the computer control of the prototype system. Commonly used algorithms for the

initialization and control of the system are clearly documented and made available through a simplified interface.

Finally, methods of improving the current system operation were discussed. The modifications that should be performed to the existing hardware or software, as well as any additions were examined.

The prototype system is presently in operation in the Spatial Data Analysis Laboratory at Virginia Tech. It has been used to collect high quality x-ray images through the sophisticated data collection hardware, and to develop image processing algorithms for explosives detection. The system detection capabilities are being tested using explosives simulants and real airport luggage and will soon be evaluated by the Federal Aviation Administration Technical Center. Efforts continue for the improvement of the detection algorithms and the expansion of the explosives database, and to provide real-time computational capabilities.

References

[ANA91] "Data Conversion Reference Manual", Analog Devices, Inc, 1992, Vol. I

[ARE96] Arendtsz, N., Hussein, E., "Compton Scattering for Density Imaging", *Proceedings of the Second Explosives Technology Symposium and Aviation Security Technology Conference*, Nov. 1996, pp. 137-141

[ASE96] AS&E, Technology for Combating Illegal Drugs and Terrorism, American Science and Engineering, 1996 Annual Report

[BOR96] Borland, "Borland C++ User's Guide", Borland International, Inc, Version 5.0, 1996

[BOU94] Bouisset, J.-F., "Security Technologies and Techniques: Airport Security Systems", *Journal of Testing and Evaluation*, JTEVA, Vol. 22, No. 3, May 1994, pp. 247-250

[CHU96] Chutjian, A., Darrach, M.R., "Improved, Portable Reversal Electron Attachment (READ) Vapor Detection System for Explosives Detection", *Proceedings of the Second Explosives Detection Symposium & Aviation Security Conference*, Nov. 1996, pp. 176-180

[CON96] Conners, R.W., Abbott, A.L., et. al., "Smart Multiple x-ray Sensor System for Explosives Detection", *Proceedings of the Second Explosives Detection Technology Symposium & Aviation Security Technology Conference*, Nov. 1996, pp. 254-259

[DDK96] Microsoft Corp., "Device Driver Developer's Kit for Windows NT 4.0", Microsoft Corp, Professional MSDN Subscription, 1996

[DRA95a] Drayer, T.H., Tront, J.G., et. al., "A Modular and Reprogrammable Real-time Processing Hardware, MORRPH", *Proceedings of FCCM '95*, Napa, CA, April 1995, pp. 11-19

[DRA95b] Drayer, T.H., King W.E., et. al., "Using Multiple FPGA Architectures for Real-time Processing of Low-level Machine Vision Functions", *Proceedings of IECON*, Nov. 95 (to be published)

[DRA97a] Drayer, T.H., Lu, Q., et. al., “Prototype Multiple Sensor Luggage Inspection System for Explosives Detection”, Spatial Data Analysis Laboratory, 1997

[DRA97b] Drayer, T.H., “A Design Methodology for Creating Programmable Logic based Real-time Image Processing Hardware”, Ph.D. Dissertation, Bradley Department of Electrical and Computer Engineering, Virginia Tech, January 1997. Available at http://etd.vt.edu

[EGG91] Eggebrecht, L.C., Interfacing to the IBM Personal Computer, Second Edition, Macmillan Computer Publishing, 1991, pp. 74-297

[EIL92] Eilbert, R.F., Krug, K.D., “Aspects of Image Recognition in Vivid Technology’s dual-energy x-ray system for Explosives Detection”, SPIE 1824, 1992

[ELA89] National Aeronautics and Space Administration - John C. Stennis Space Science and Technology Laboratory, “ELAS”, Science and Technology Laboratory Applications Software Programmer Reference, Volume I, Report No. 183, May 1989, pp. 1-4, 20-22

[EUR96] Europ Scan, Inc, “Xcalibur, An innovative multi-energy x-ray Explosive Device Detection System”, *Proceedings of the Second Explosives Detection Symposium & Aviation Security Conference*, Nov. 1996, pp. 230-235

[FAI94] Fainberg, A., “Explosives Detection for Aviation Security”, Science, *American Association for the Advancement of Science*, vol. 255, pp. 1531-1537

[GOZ91] Gozani, T., “Principles of Nuclear-Based Explosive Detection Systems”, *Proceedings of the First International Symposium on Explosive Detection Technology*, Nov. 1991, pp. 27-55

[GUL95] Gulmay, Ltd., “MP-1 Technical Manual”, Gulmay Ltd., May 1995

[INT90] Intel Corporation, “MCS File Format”

[INV96] Invision Technologies, CTX 5000 Product Literature, 1996 (URL: http://www.invision-tech.com

[KIT96] Kitzinger, D., Cheung S., "High Resolution 3D Geometric Modeling for Improved Explosive Detection Simulation", *Proceedings of the Second Explosives Detection Technology Symposium & Aviation Security Technology Conference*, Nov. 1996, pp. 113-117

[LUM95] LumenX, "Gemini 2000 x-ray Controller System Manual", LumenX Company, 1995

[MIC93] Michette, A.G., and Buckley, C.J., x-ray Science and Technology, Institute of Physics Publushing, 1993, pp. 1-44

[MOT92] "Static Memory Databook", Motorola, Inc, 1992

[MOT93] "Linear/Interface ICs Device Data", Motorola, Inc., 1993, Vol. II, pp. 7-25 to 7-27

[NEW96] Newsday, "Troublesome Trial of a Bomb Detector", Sept. 3, 1997 (URL: http://www.newsday.com)

[NOV92] Novakoff, A.K., "FAA bulk technology overview for explosive detection", *Proceedings of the SPIE*, Vol. 1824, Nov. 1992, pp. 2-12

[NRC96] NRC, "Airline Passenger and Security Screening, New Technologies and Implementation Issues", National Research Council, Publication NMAB-482-1, National Academy Press, Washington D.C., 1996

[PCI93] PCI Special Interest Group, "PCI Local Bus Specification", Revision 2.0, April 30, 1993

[POL94] Polski, P.A., "International Aviation Security Research and Development", *Journal of Testing and Evaluation*, JTEVA, Vol. 22, No. 3, May 1994, pp. 267-274

[RAY96] Rayner, T., Thorson, B., et. al., "Explosives Detection Using Quadrupole Resonance Analysis", *Proceedings of the Second Explosives Detection Symposium & Aviation Security Conference*, Nov. 1996, pp. 275-280

[ROD91] Roder, F.L., "The Evolution of Computed Tomography (CT) as an Explosives Detection Modality", *Proceedings of the First International Symposium on Explosive Detection Technology*, Nov. 1991, pp. 297-308

[SCH91] Schafer, D., Annis, M., and Hacker, M., "New x-ray Technology for the Detection of Explosives", *Proceedings of the First International Symposium on Explosive Detection Technology*, Nov. 1991, pp. 269-281

[SDK96] Microsoft Corp., “Software Developer’s Kit for Win32”, Microsoft Corp., Professional MSDN Subscription, 1996

[VEL85] Velmex, Inc., “8300 Series Stepping Motor Controller/Driver User’s Guide”, Velmex, Inc, Jan. 1985

[VIV97] Vivid Technologies, “Corporate & Product Overview”, Vivid Technologies, May 1997

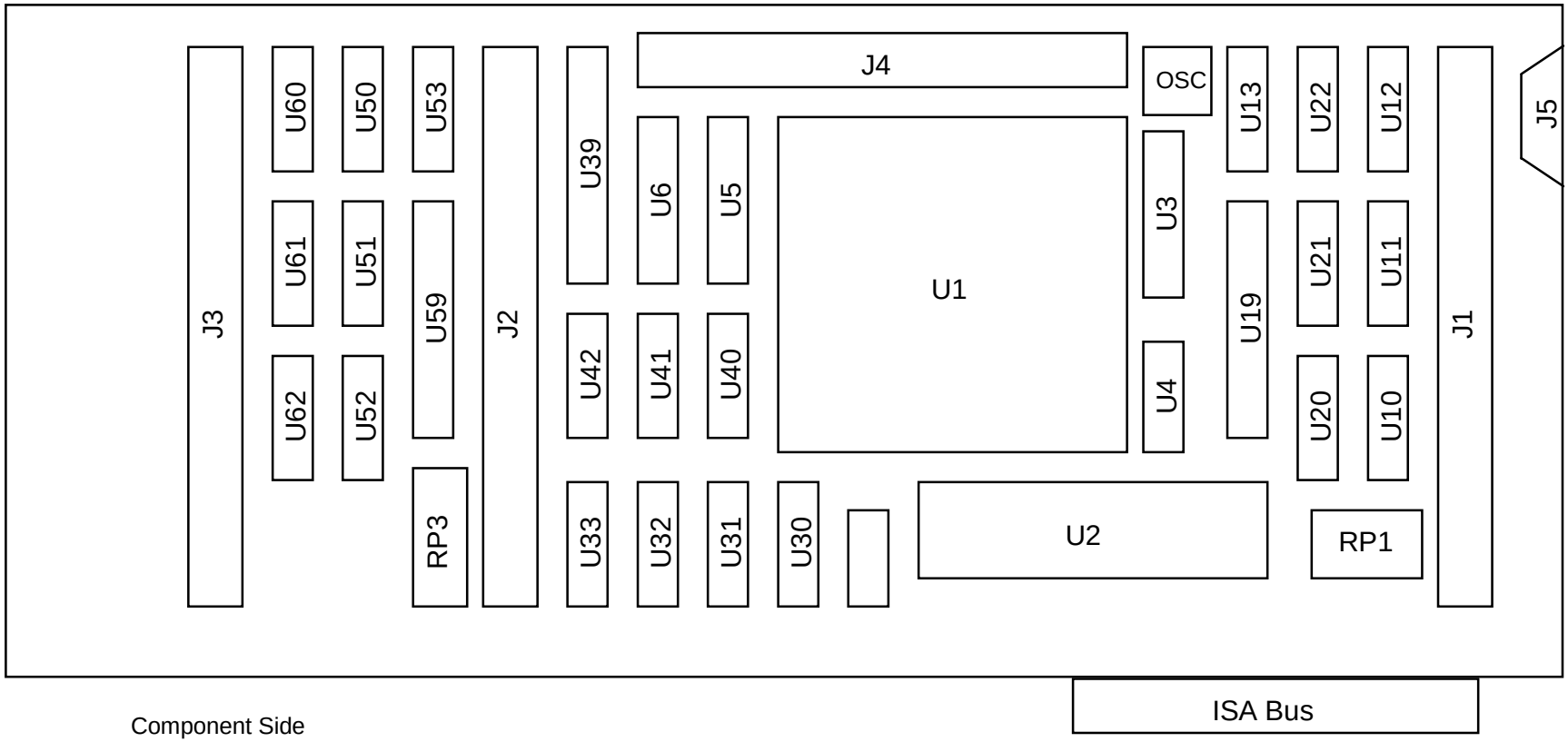
[VOU94] Vourvopoulos, G., “Methods for the detection of explosives and contraband”, *Chemistry & Industry*, April 18, 1994, pp. 297-300

[XIL94a] Xilinx, Inc., “The Programmable Logic Data Book”, Xilinx, Inc, April 1995

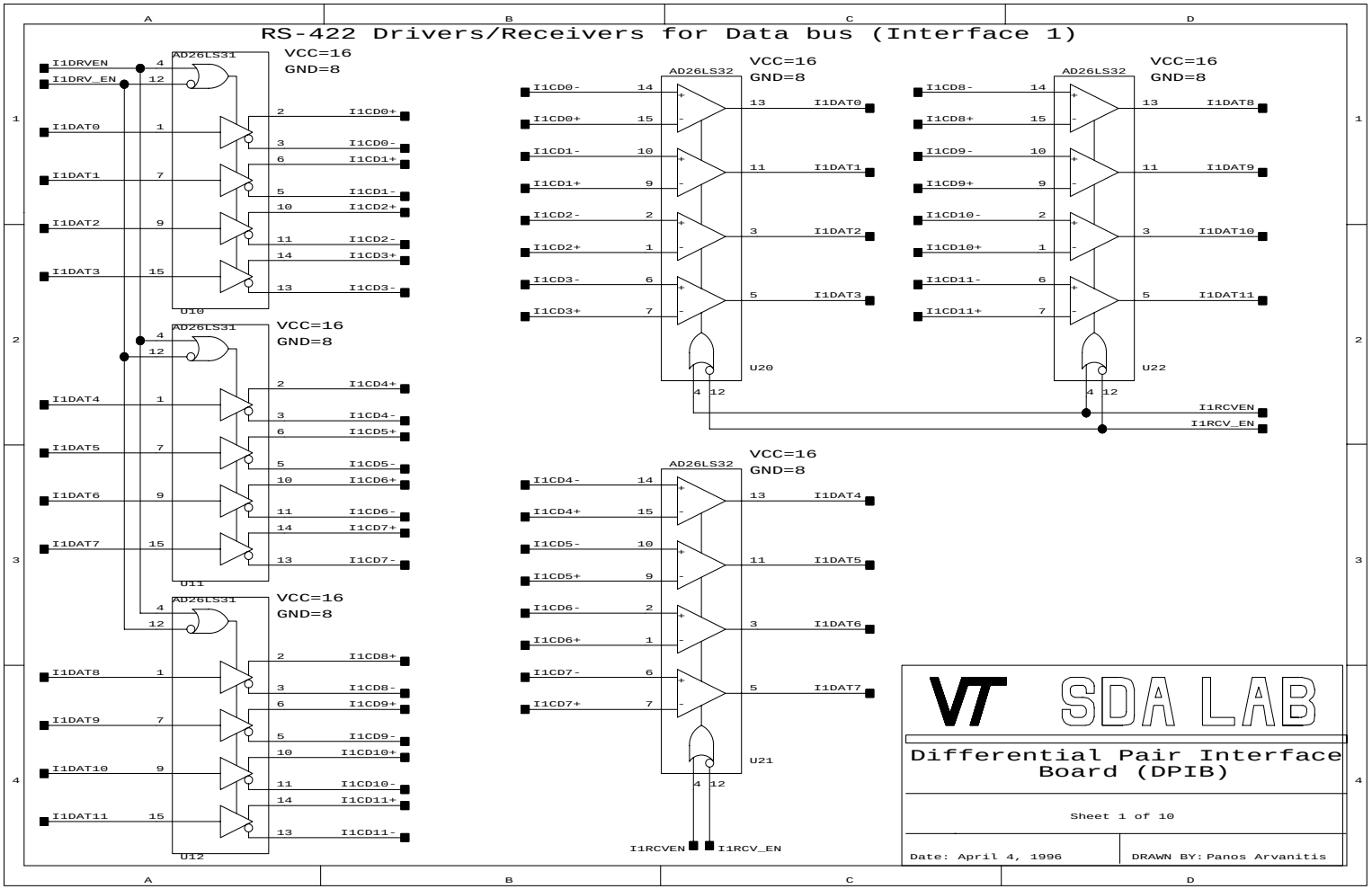
[XIL94b] Xilinx Corporation, Components Price List, Sales Literature, October 1994

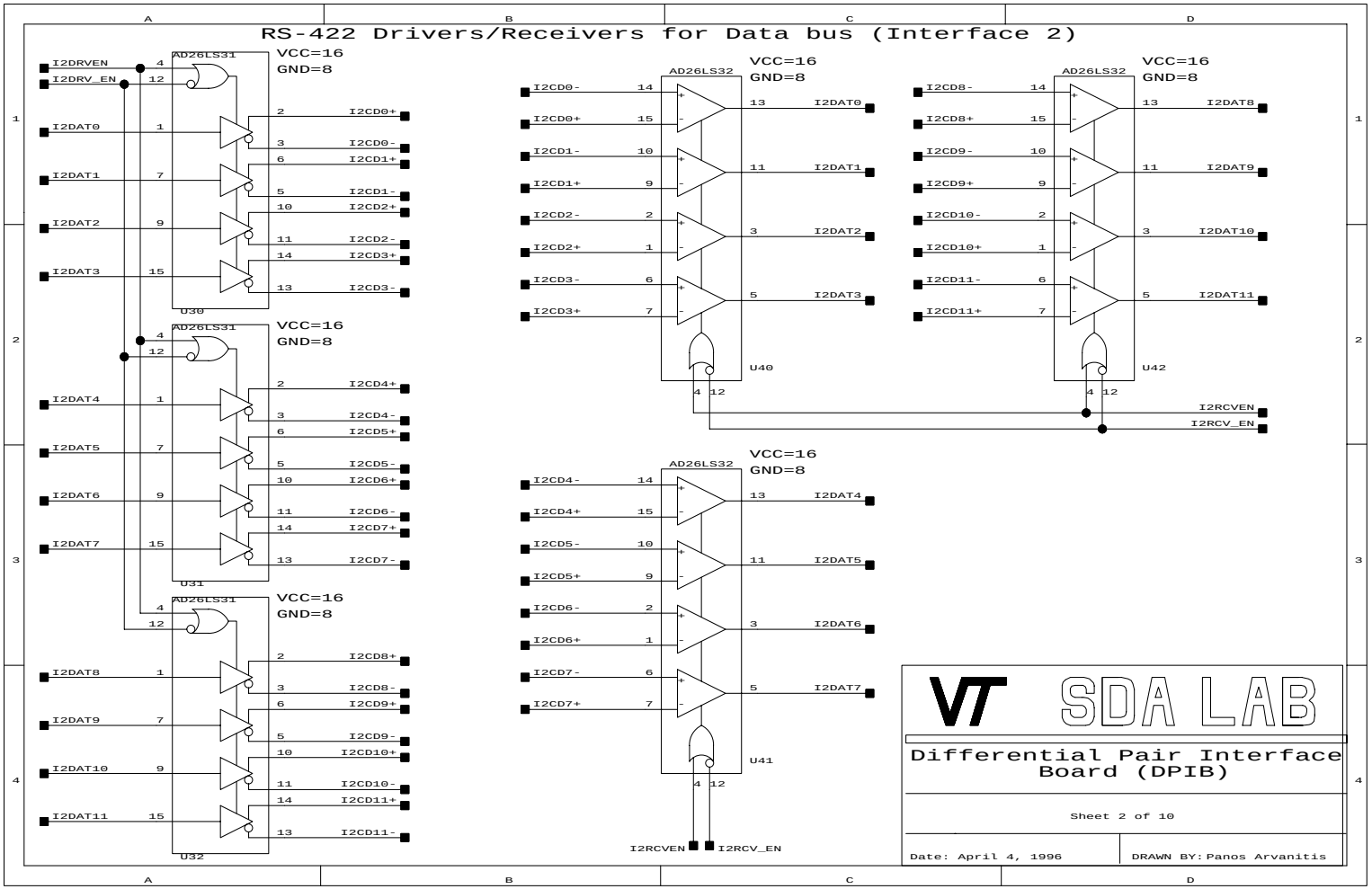
Appendix A. DPIB Board Level Schematics

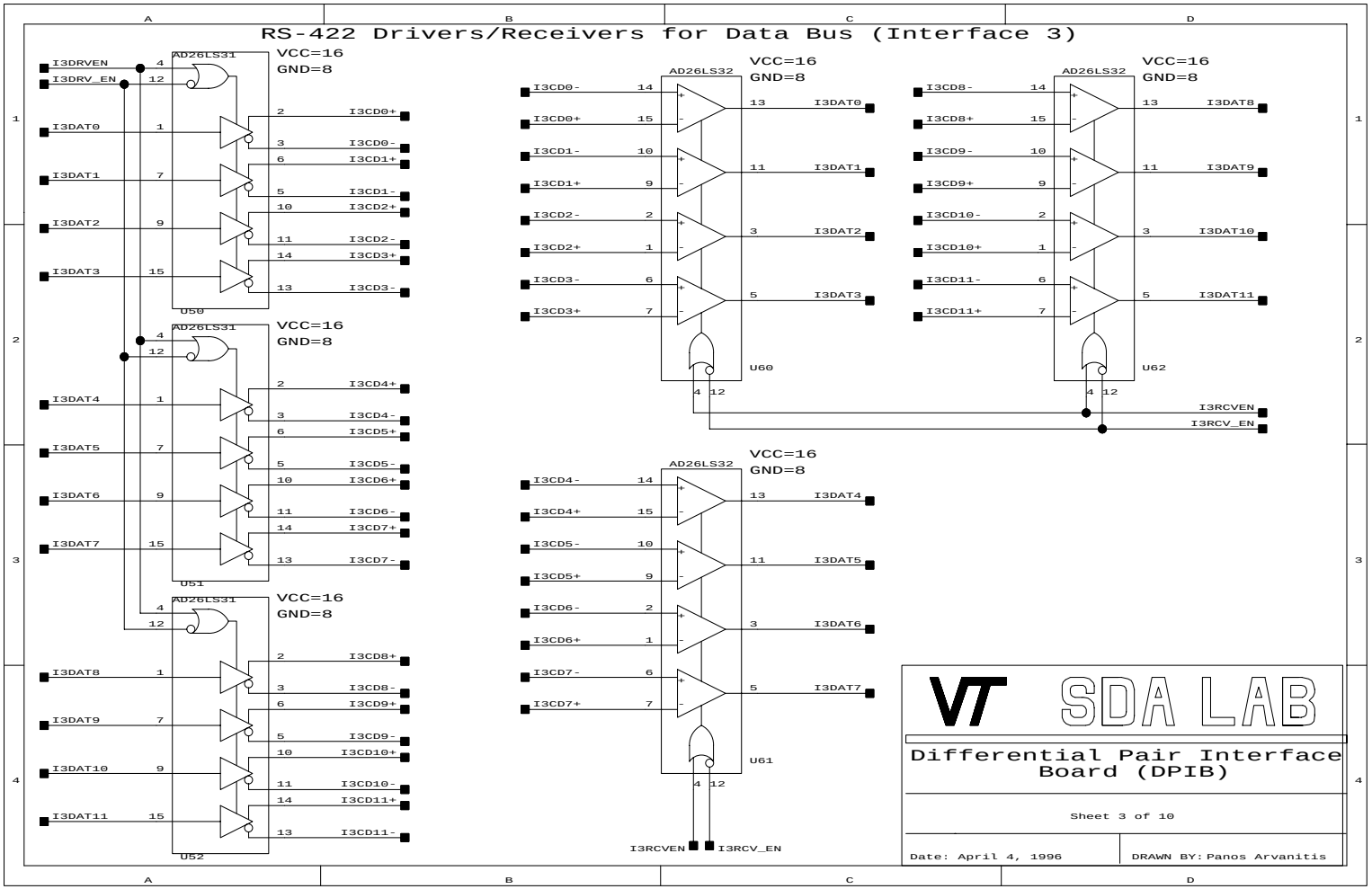
Appendix A contains the board level schematics of the Differential Pair Interface Board. A component location diagram is first shown, with unit numbers to identify each IC on the board. The DPIB schematics are then shown, with each component using the same unit number as on the component location diagram.

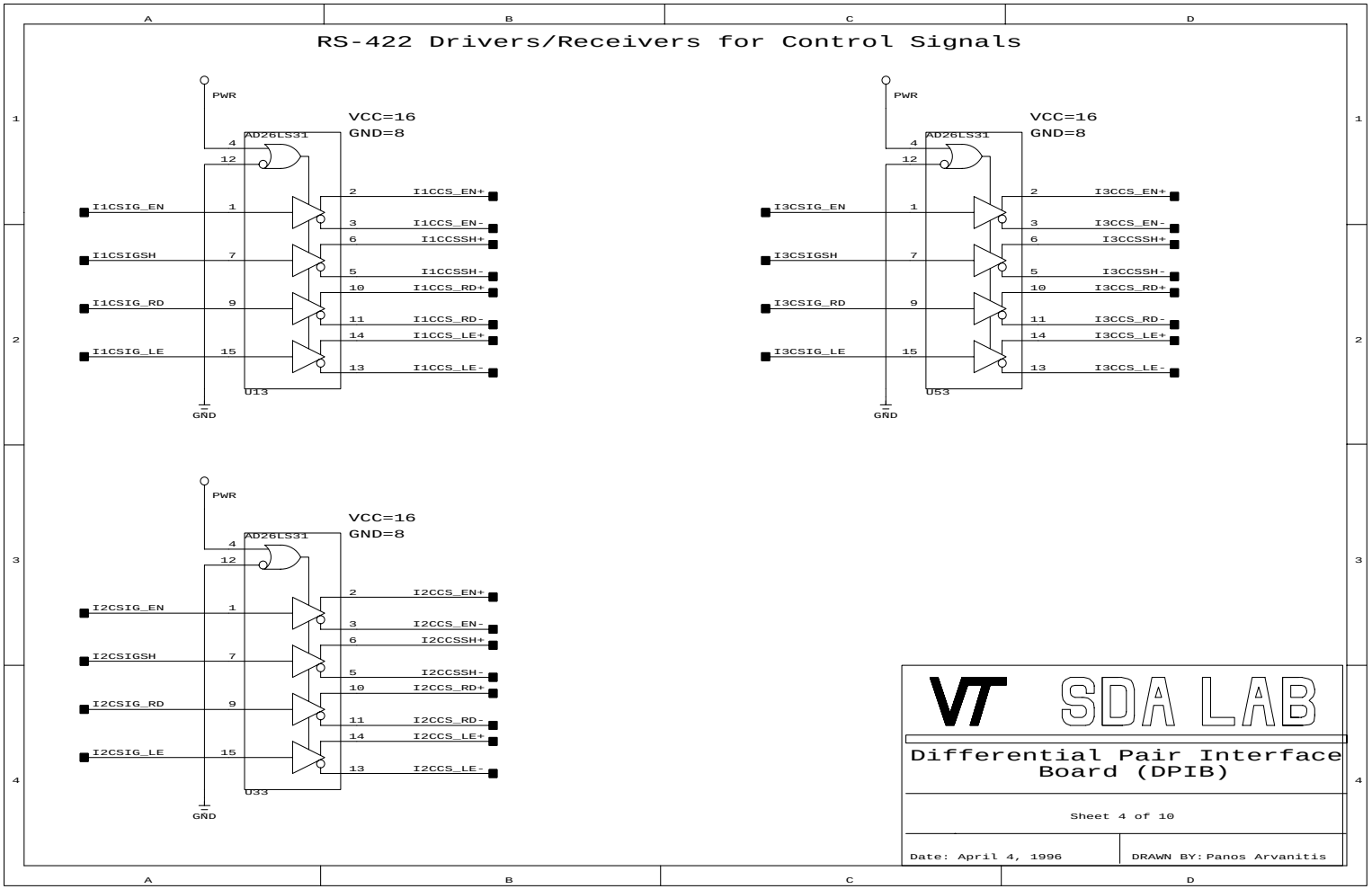


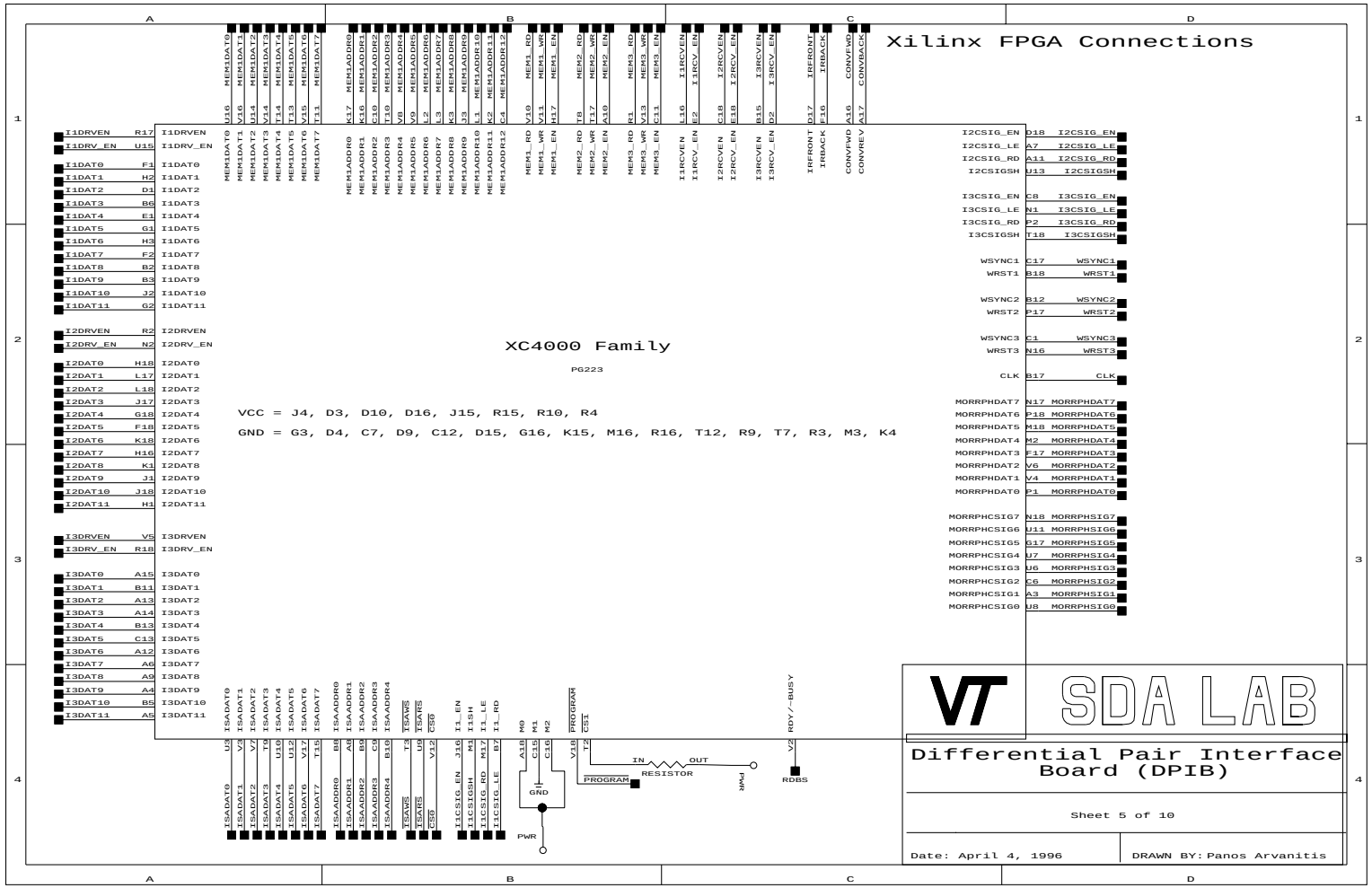
Differential Pair Interface Board Component Location Diagram

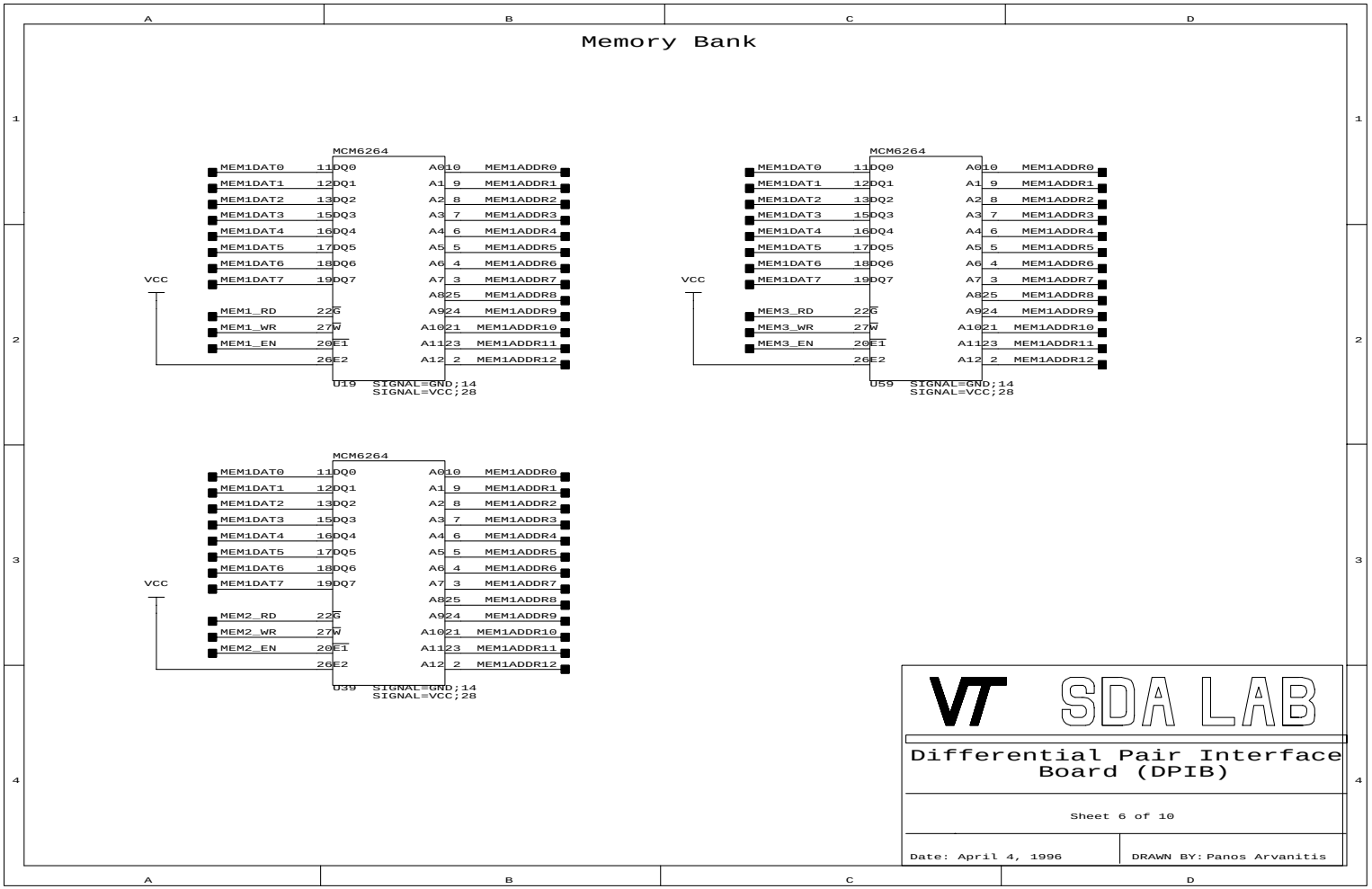

RS-422 Drivers/Receivers for Data bus (Interface 1)
AD26LS31
VCC=16
GND=8
I1DRVEN
I1DRV_EN
I1DAT0
I1DAT1
I1DAT2
I1DAT3
I1CD0+
I1CD0-
I1CD1+
I1CD1-
I1CD2+
I1CD2-
I1CD3+
I1CD3-
U10
I1DAT4
I1DAT5
I1DAT6
I1DAT7
I1CD4+
I1CD4-
I1CD5+
I1CD5-
I1CD6+
I1CD6-
I1CD7+
I1CD7-
U11
I1DAT8
I1DAT9
I1DAT10
I1DAT11
I1CD8+
I1CD8-
I1CD9+
I1CD9-
I1CD10+
I1CD10-
I1CD11+
I1CD11-
U12
AD26LS32
I1CD0-
I1CD0+
I1CD1-
I1CD1+
I1CD2-
I1CD2+
I1CD3-
I1CD3+
I1DAT0
I1DAT1
I1DAT2
I1DAT3
U20
I1CD4-
I1CD4+
I1CD5-
I1CD5+
I1CD6-
I1CD6+
I1CD7-
I1CD7+
I1DAT4
I1DAT5
I1DAT6
I1DAT7
U21
I1CD8-
I1CD8+
I1CD9-
I1CD9+
I1CD10-
I1CD10+
I1CD11-
I1CD11+
I1DAT8
I1DAT9
I1DAT10
I1DAT11
U22
I1RCVEN
I1RCV_EN
SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 1 of 10
Date: April 4, 1996
DRAWN BY: Panos Arvanitis

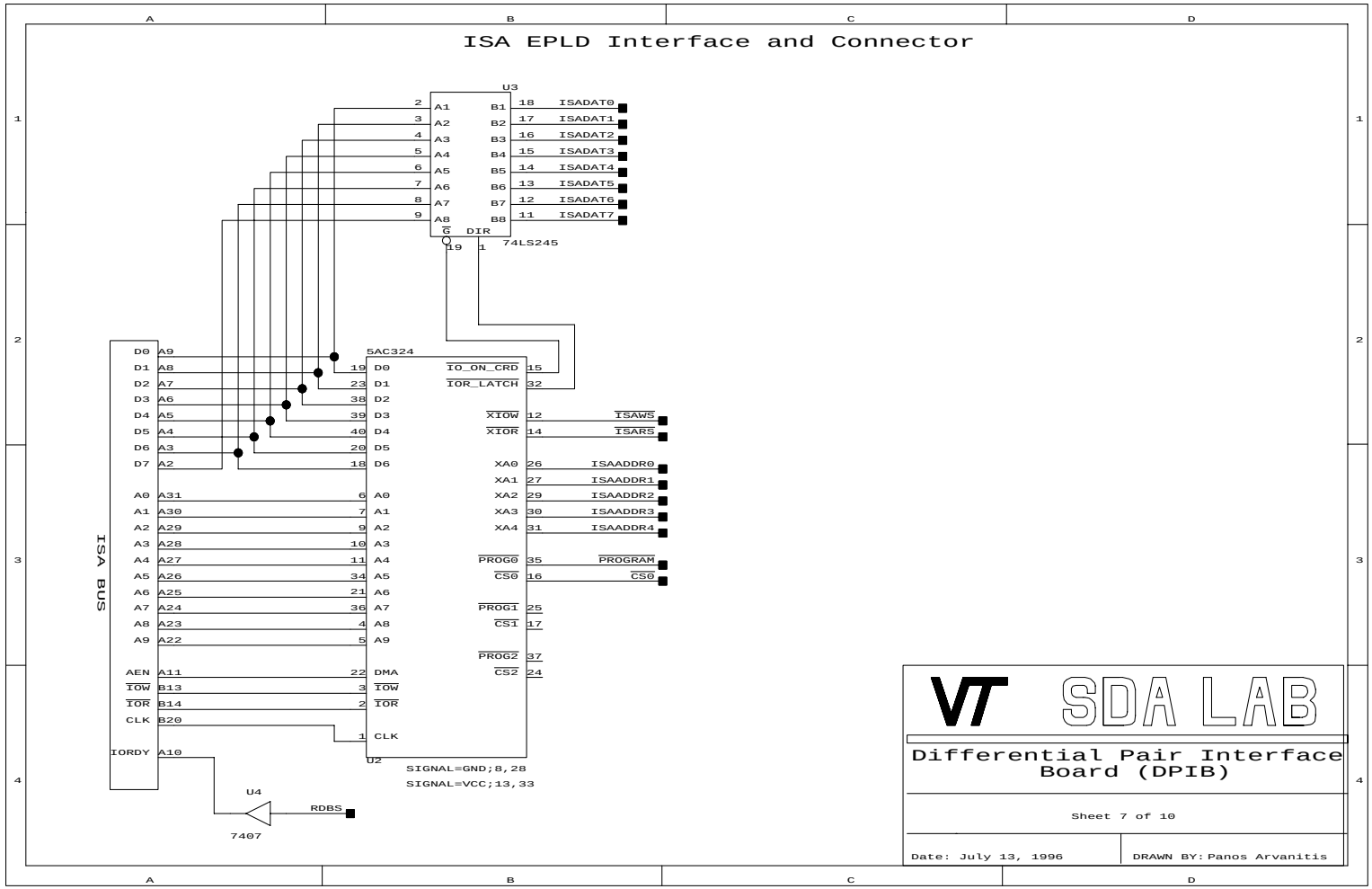

RS-422 Drivers/Receivers for Data bus (Interface 2)
AD26LS31
VCC=16
GND=8
U30
U31
U32
I2DRVEN
I2DRV_EN
I2DAT0
I2DAT1
I2DAT2
I2DAT3
I2DAT4
I2DAT5
I2DAT6
I2DAT7
I2DAT8
I2DAT9
I2DAT10
I2DAT11
I2CD0+
I2CD0-
I2CD1+
I2CD1-
I2CD2+
I2CD2-
I2CD3+
I2CD3-
I2CD4+
I2CD4-
I2CD5+
I2CD5-
I2CD6+
I2CD6-
I2CD7+
I2CD7-
I2CD8+
I2CD8-
I2CD9+
I2CD9-
I2CD10+
I2CD10-
I2CD11+
I2CD11-
AD26LS32
U40
U41
U42
I2RCVEN
I2RCV_EN
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 2 of 10
Date: April 4, 1996
DRAWN BY: Panos Arvanitis

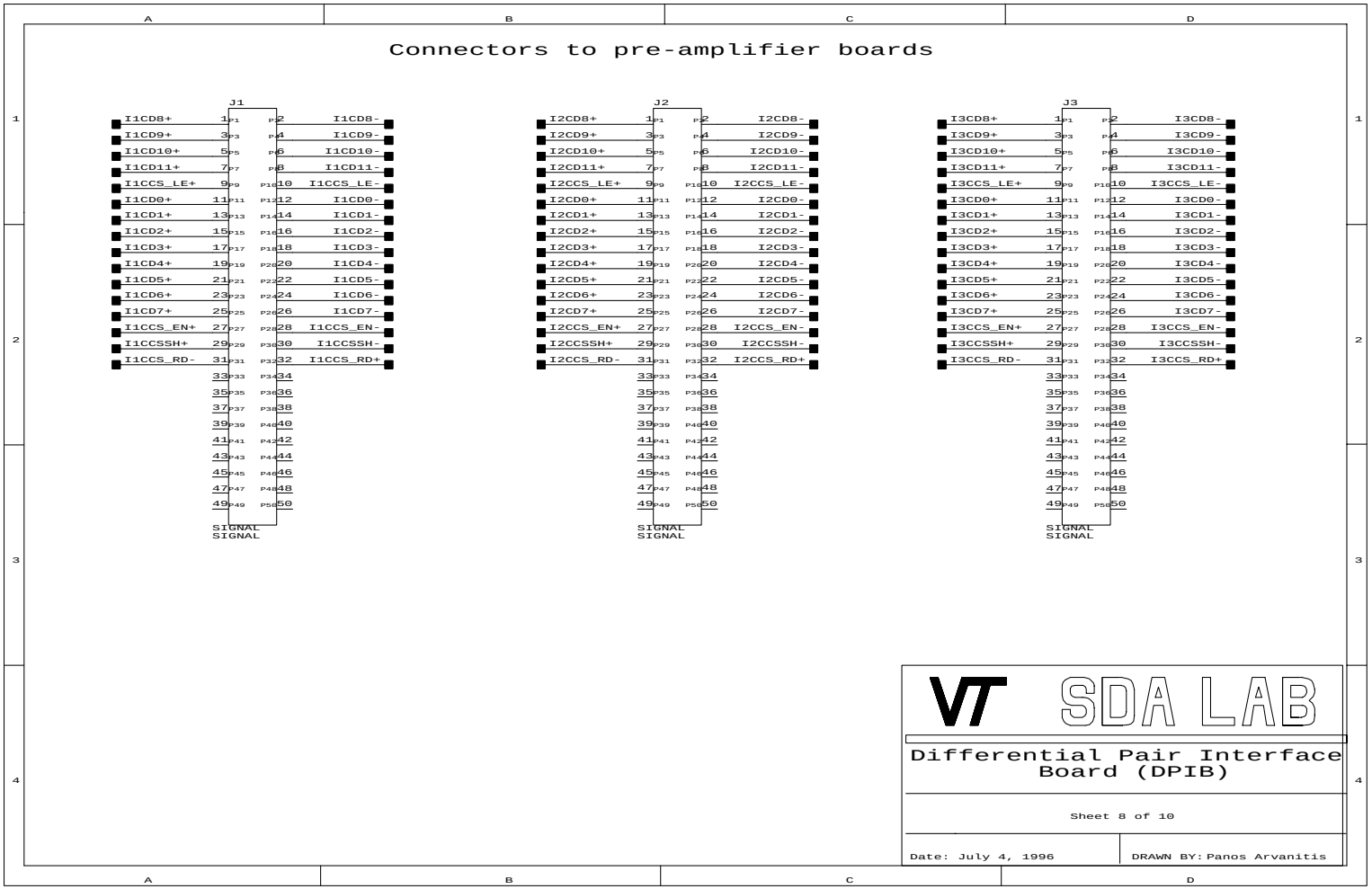

RS-422 Drivers/Receivers for Data Bus (Interface 3)
I3DRVEN
I3DRV_EN
I3DAT0
I3DAT1
I3DAT2
I3DAT3
AD26LS31
VCC=16
GND=8
I3CD0+
I3CD0-
I3CD1+
I3CD1-
I3CD2+
I3CD2-
I3CD3+
I3CD3-
U50
I3DAT4
I3DAT5
I3DAT6
I3DAT7
I3CD4+
I3CD4-
I3CD5+
I3CD5-
I3CD6+
I3CD6-
I3CD7+
I3CD7-
U51
I3DAT8
I3DAT9
I3DAT10
I3DAT11
I3CD8+
I3CD8-
I3CD9+
I3CD9-
I3CD10+
I3CD10-
I3CD11+
I3CD11-
U52
AD26LS32
I3CD0-
I3CD0+
I3CD1-
I3CD1+
I3CD2-
I3CD2+
I3CD3-
I3CD3+
I3DAT0
I3DAT1
I3DAT2
I3DAT3
U60
I3CD4-
I3CD4+
I3CD5-
I3CD5+
I3CD6-
I3CD6+
I3CD7-
I3CD7+
I3DAT4
I3DAT5
I3DAT6
I3DAT7
U61
I3CD8-
I3CD8+
I3CD9-
I3CD9+
I3CD10-
I3CD10+
I3CD11-
I3CD11+
I3DAT8
I3DAT9
I3DAT10
I3DAT11
U62
I3RCVEN
I3RCV_EN
SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 3 of 10
Date: April 4, 1996
DRAWN BY: Panos Arvanitis

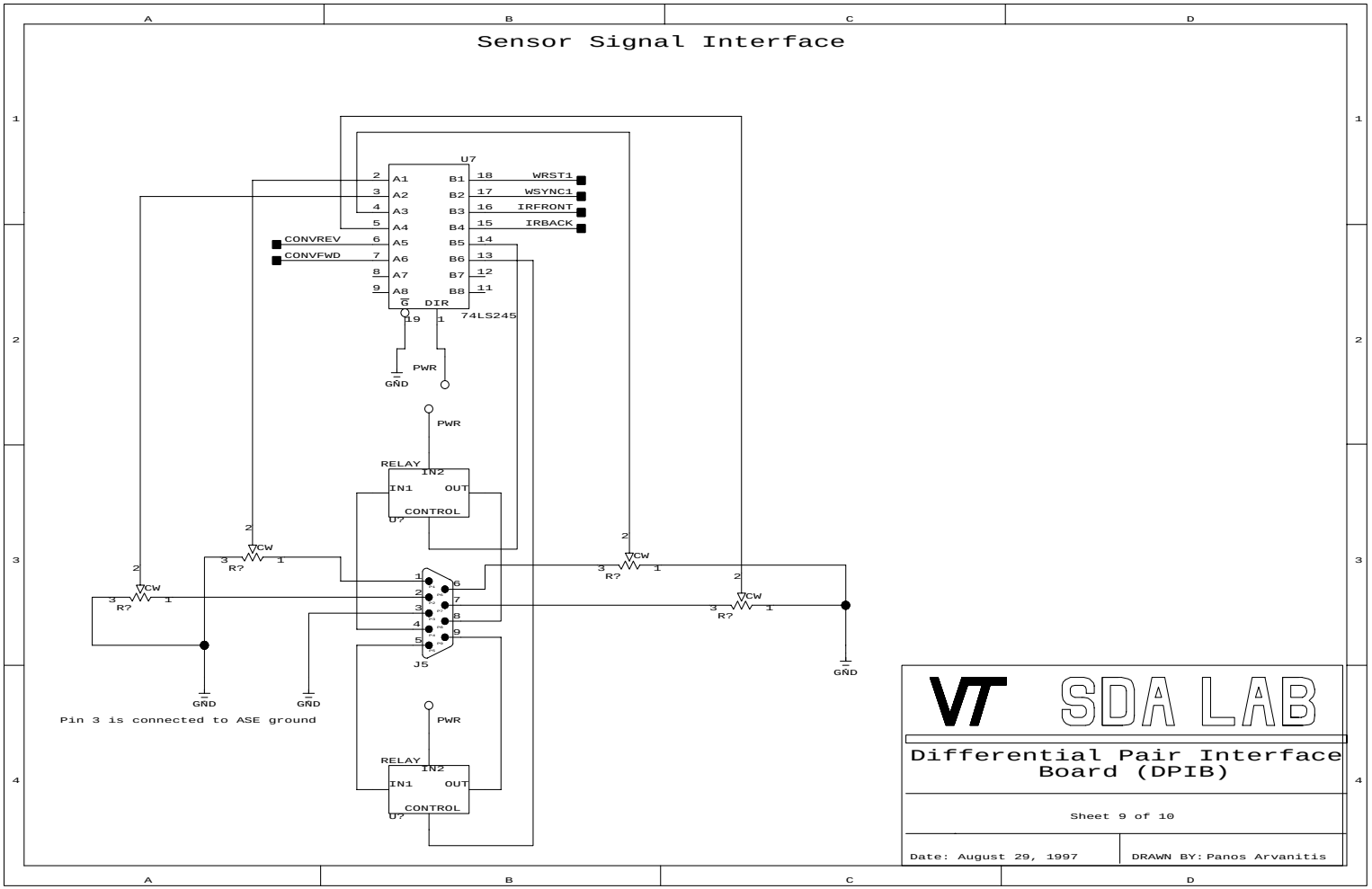

RS-422 Drivers/Receivers for Control Signals
PWR
VCC=16
GND=8
AD26LS31
U13
GND
I1CSIG_EN
I1CSIGSH
I1CSIG_RD
I1CSIG_LE
I1CCS_EN+
I1CCS_EN-
I1CCSSH+
I1CCSSH-
I1CCS_RD+
I1CCS_RD-
I1CCS_LE+
I1CCS_LE-
U53
I3CSIG_EN
I3CSIGSH
I3CSIG_RD
I3CSIG_LE
I3CCS_EN+
I3CCS_EN-
I3CCSSH+
I3CCSSH-
I3CCS_RD+
I3CCS_RD-
I3CCS_LE+
I3CCS_LE-
U33
I2CSIG_EN
I2CSIGSH
I2CSIG_RD
I2CSIG_LE
I2CCS_EN+
I2CCS_EN-
I2CCSSH+
I2CCSSH-
I2CCS_RD+
I2CCS_RD-
I2CCS_LE+
I2CCS_LE-
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 4 of 10
Date: April 4, 1996
DRAWN BY: Panos Arvanitis

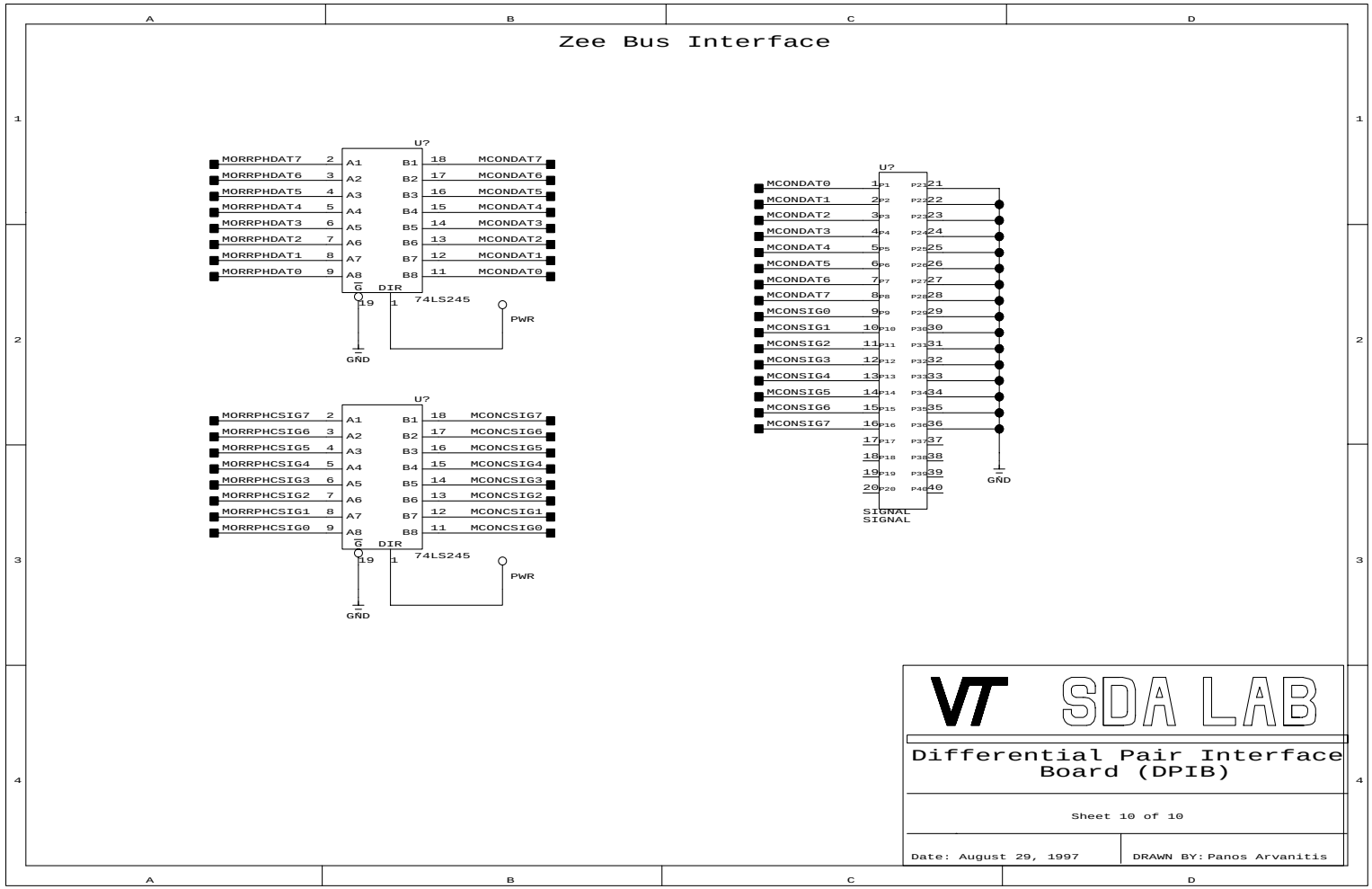

Xilinx FPGA Connections
XC4000 Family
PG223
VCC = J4, D3, D10, D16, J15, R15, R10, R4
GND = G3, D4, C7, D9, C12, D15, G16, K15, M16, R16, T12, R9, T7, R3, M3, K4
I1DRVEN R17 I1DRVEN
I1DRV_EN U15 I1DRV_EN
I1DAT0 F1 I1DAT0
I1DAT1 H2 I1DAT1
I1DAT2 D1 I1DAT2
I1DAT3 B6 I1DAT3
I1DAT4 E1 I1DAT4
I1DAT5 G1 I1DAT5
I1DAT6 H3 I1DAT6
I1DAT7 F2 I1DAT7
I1DAT8 B2 I1DAT8
I1DAT9 B3 I1DAT9
I1DAT10 J2 I1DAT10
I1DAT11 G2 I1DAT11
I2DRVEN R2 I2DRVEN
I2DRV_EN N2 I2DRV_EN
I2DAT0 H18 I2DAT0
I2DAT1 L17 I2DAT1
I2DAT2 L18 I2DAT2
I2DAT3 J17 I2DAT3
I2DAT4 G18 I2DAT4
I2DAT5 F18 I2DAT5
I2DAT6 K18 I2DAT6
I2DAT7 H16 I2DAT7
I2DAT8 K1 I2DAT8
I2DAT9 J1 I2DAT9
I2DAT10 J18 I2DAT10
I2DAT11 H1 I2DAT11
I3DRVEN V5 I3DRVEN
I3DRV_EN R18 I3DRV_EN
I3DAT0 A15 I3DAT0
I3DAT1 B11 I3DAT1
I3DAT2 A13 I3DAT2
I3DAT3 A14 I3DAT3
I3DAT4 B13 I3DAT4
I3DAT5 C13 I3DAT5
I3DAT6 A12 I3DAT6
I3DAT7 A6 I3DAT7
I3DAT8 A9 I3DAT8
I3DAT9 A4 I3DAT9
I3DAT10 B5 I3DAT10
I3DAT11 A5 I3DAT11
I2CSIG_EN D18 I2CSIG_EN
I2CSIG_LE A7 I2CSIG_LE
I2CSIG_RD A11 I2CSIG_RD
I2CSIGSH U13 I2CSIGSH
I3CSIG_EN C8 I3CSIG_EN
I3CSIG_LE N1 I3CSIG_LE
I3CSIG_RD P2 I3CSIG_RD
I3CSIGSH T18 I3CSIGSH
WSYNC1 C17 WSYNC1
WRST1 B18 WRST1
WSYNC2 B12 WSYNC2
WRST2 P17 WRST2
WSYNC3 C1 WSYNC3
WRST3 N16 WRST3
CLK B17 CLK
MORRPHDAT7 N17 MORRPHDAT7
MORRPHDAT6 P18 MORRPHDAT6
MORRPHDAT5 M18 MORRPHDAT5
MORRPHDAT4 M2 MORRPHDAT4
MORRPHDAT3 F17 MORRPHDAT3
MORRPHDAT2 V6 MORRPHDAT2
MORRPHDAT1 V4 MORRPHDAT1
MORRPHDAT0 P1 MORRPHDAT0
MORRPHCSIG7 N18 MORRPHSIG7
MORRPHCSIG6 U11 MORRPHSIG6
MORRPHCSIG5 G17 MORRPHSIG5
MORRPHCSIG4 U7 MORRPHSIG4
MORRPHCSIG3 U6 MORRPHSIG3
MORRPHCSIG2 C6 MORRPHSIG2
MORRPHCSIG1 A3 MORRPHSIG1
MORRPHCSIG0 U8 MORRPHSIG0
MEM1DAT0 U16 MEM1DAT1 V16 MEM1DAT2 U14 MEM1DAT3 V14 MEM1DAT4 T14 MEM1DAT5 T13 MEM1DAT6 V15 MEM1DAT7 T11
MEM1ADDR0 K17 MEM1ADDR1 K16 MEM1ADDR2 C10 MEM1ADDR3 T10 MEM1ADDR4 V8 MEM1ADDR5 V9 MEM1ADDR6 L2 MEM1ADDR7 K3 MEM1ADDR8 K3 MEM1ADDR9 J3 MEM1ADDR10 L1 MEM1ADDR11 K2 MEM1ADDR12 C4
MEM1_RD V10 MEM1_WR V11 MEM1_EN H17
MEM2_RD T8 MEM2_WR T17 MEM2_EN A10
MEM3_RD K1 MEM3_WR V13 MEM3_EN C11
I1RCVEN L16 I1RCV_EN F2
I2RCVEN C18 I2RCV_EN E18
I3RCVEN B15 I3RCV_EN D2
IRFRONT D17 IRBACK F16
CONVFWD A16 CONVREV A17
ISADAT0 U3 ISADAT1 V3 ISADAT2 V7 ISADAT3 T9 ISADAT4 U10 ISADAT5 U12 ISADAT6 V17 ISADAT7 T15
ISAADDR0 R8 ISAADDR1 A8 ISAADDR2 B8 ISAADDR3 C8 ISAADDR4 B10
ISAWS T4 ISARS U5 CS0 V12
I1_EN J16 I1SH M1 I1_LE M17 I1_RD B7
M0 A18 M1 C15 M2 C16
PROGRAM V18 CST T2
RDY/-BUSY V2
I1CSIG_EN J16 I1CSIGSH M1 I1CSIG_RD M17 I1CSIG_LE B7
GND
PWR
IN OUT RESISTOR PROGRAM PWR
RDBS
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 5 of 10
Date: April 4, 1996
DRAWN BY: Panos Arvanitis


Memory Bank
MCM6264
MEM1DAT0 11 DQ0
MEM1DAT1 12 DQ1
MEM1DAT2 13 DQ2
MEM1DAT3 15 DQ3
MEM1DAT4 16 DQ4
MEM1DAT5 17 DQ5
MEM1DAT6 18 DQ6
MEM1DAT7 19 DQ7
MEM1_RD 22 G
MEM1_WR 27 W
MEM1_EN 20 E1
26 E2
A0 10 MEM1ADDR0
A1 9 MEM1ADDR1
A2 8 MEM1ADDR2
A3 7 MEM1ADDR3
A4 6 MEM1ADDR4
A5 5 MEM1ADDR5
A6 4 MEM1ADDR6
A7 3 MEM1ADDR7
A8 25 MEM1ADDR8
A9 24 MEM1ADDR9
A10 21 MEM1ADDR10
A11 23 MEM1ADDR11
A12 2 MEM1ADDR12
VCC
U19 SIGNAL=GND;14
SIGNAL=VCC;28
MCM6264
MEM1DAT0 11 DQ0
MEM1DAT1 12 DQ1
MEM1DAT2 13 DQ2
MEM1DAT3 15 DQ3
MEM1DAT4 16 DQ4
MEM1DAT5 17 DQ5
MEM1DAT6 18 DQ6
MEM1DAT7 19 DQ7
MEM3_RD 22 G
MEM3_WR 27 W
MEM3_EN 20 E1
26 E2
A0 10 MEM1ADDR0
A1 9 MEM1ADDR1
A2 8 MEM1ADDR2
A3 7 MEM1ADDR3
A4 6 MEM1ADDR4
A5 5 MEM1ADDR5
A6 4 MEM1ADDR6
A7 3 MEM1ADDR7
A8 25 MEM1ADDR8
A9 24 MEM1ADDR9
A10 21 MEM1ADDR10
A11 23 MEM1ADDR11
A12 2 MEM1ADDR12
VCC
U59 SIGNAL=GND;14
SIGNAL=VCC;28
MCM6264
MEM1DAT0 11 DQ0
MEM1DAT1 12 DQ1
MEM1DAT2 13 DQ2
MEM1DAT3 15 DQ3
MEM1DAT4 16 DQ4
MEM1DAT5 17 DQ5
MEM1DAT6 18 DQ6
MEM1DAT7 19 DQ7
MEM2_RD 22 G
MEM2_WR 27 W
MEM2_EN 20 E1
26 E2
A0 10 MEM1ADDR0
A1 9 MEM1ADDR1
A2 8 MEM1ADDR2
A3 7 MEM1ADDR3
A4 6 MEM1ADDR4
A5 5 MEM1ADDR5
A6 4 MEM1ADDR6
A7 3 MEM1ADDR7
A8 25 MEM1ADDR8
A9 24 MEM1ADDR9
A10 21 MEM1ADDR10
A11 23 MEM1ADDR11
A12 2 MEM1ADDR12
VCC
U39 SIGNAL=GND;14
SIGNAL=VCC;28
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 6 of 10
Date: April 4, 1996
DRAWN BY: Panos Arvanitis


ISA EPLD Interface and Connector
U3
74LS245
DIR
ISADAT0
ISADAT1
ISADAT2
ISADAT3
ISADAT4
ISADAT5
ISADAT6
ISADAT7
5AC324
U2
ISA BUS
IO_ON_CRD
IOR_LATCH
XIOW
XIOR
ISAWS
ISARS
XA0
XA1
XA2
XA3
XA4
ISAADDR0
ISAADDR1
ISAADDR2
ISAADDR3
ISAADDR4
PROG0
CS0
PROGRAM
PROG1
CS1
PROG2
CS2
AEN
IOW
IOR
CLK
IORDY
DMA
SIGNAL=GND;8,28
SIGNAL=VCC;13,33
U4
7407
RDBS
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 7 of 10
Date: July 13, 1996
DRAWN BY: Panos Arvanitis

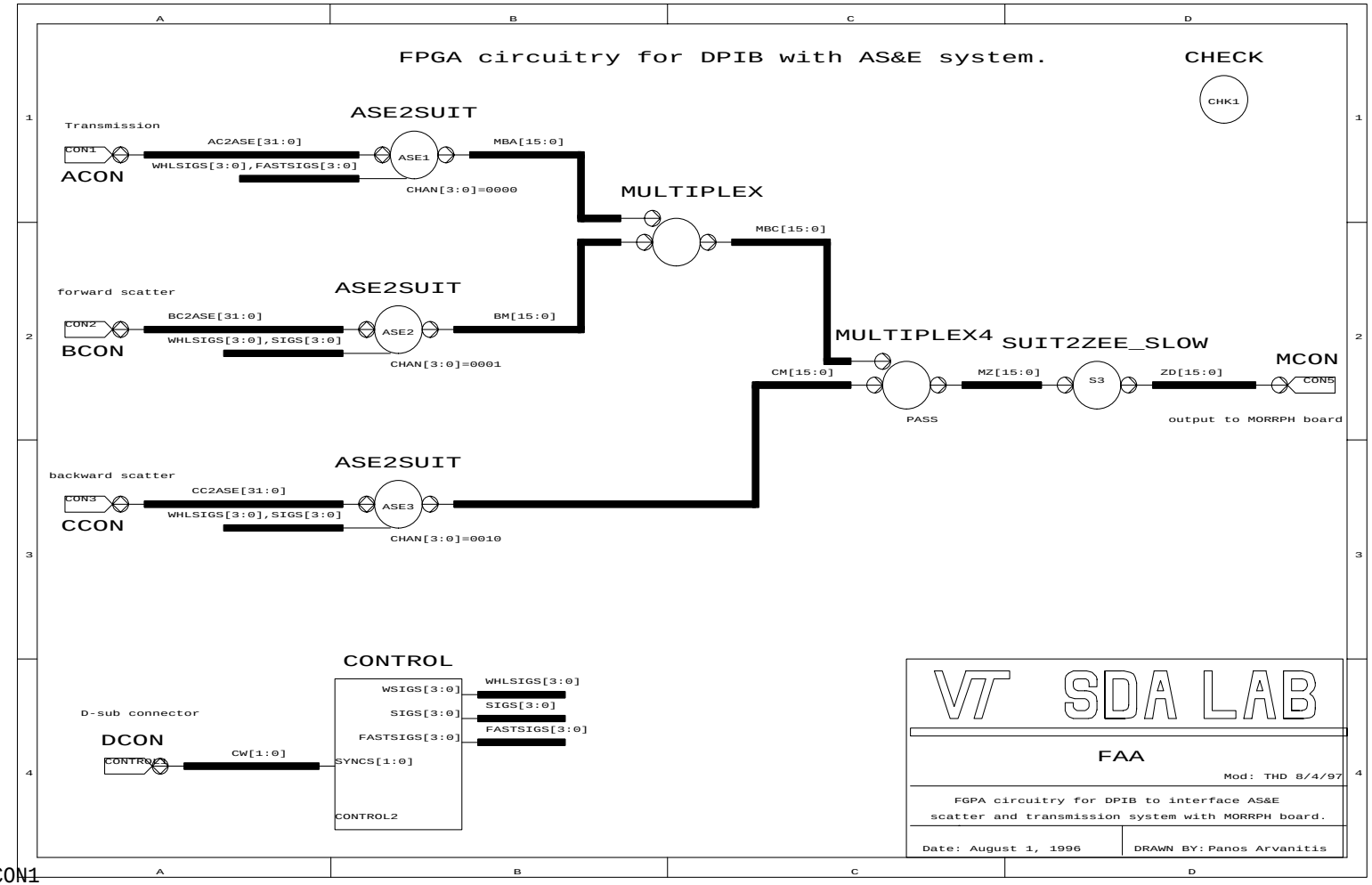

Connectors to pre-amplifier boards
J1
I1CD8+ I1CD8-
I1CD9+ I1CD9-
I1CD10+ I1CD10-
I1CD11+ I1CD11-
I1CCS_LE+ I1CCS_LE-
I1CD0+ I1CD0-
I1CD1+ I1CD1-
I1CD2+ I1CD2-
I1CD3+ I1CD3-
I1CD4+ I1CD4-
I1CD5+ I1CD5-
I1CD6+ I1CD6-
I1CD7+ I1CD7-
I1CCS_EN+ I1CCS_EN-
I1CCSSH+ I1CCSSH-
I1CCS_RD- I1CCS_RD+
SIGNAL
SIGNAL
J2
I2CD8+ I2CD8-
I2CD9+ I2CD9-
I2CD10+ I2CD10-
I2CD11+ I2CD11-
I2CCS_LE+ I2CCS_LE-
I2CD0+ I2CD0-
I2CD1+ I2CD1-
I2CD2+ I2CD2-
I2CD3+ I2CD3-
I2CD4+ I2CD4-
I2CD5+ I2CD5-
I2CD6+ I2CD6-
I2CD7+ I2CD7-
I2CCS_EN+ I2CCS_EN-
I2CCSSH+ I2CCSSH-
I2CCS_RD- I2CCS_RD+
SIGNAL
SIGNAL
J3
I3CD8+ I3CD8-
I3CD9+ I3CD9-
I3CD10+ I3CD10-
I3CD11+ I3CD11-
I3CCS_LE+ I3CCS_LE-
I3CD0+ I3CD0-
I3CD1+ I3CD1-
I3CD2+ I3CD2-
I3CD3+ I3CD3-
I3CD4+ I3CD4-
I3CD5+ I3CD5-
I3CD6+ I3CD6-
I3CD7+ I3CD7-
I3CCS_EN+ I3CCS_EN-
I3CCSSH+ I3CCSSH-
I3CCS_RD- I3CCS_RD+
SIGNAL
SIGNAL
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 8 of 10
Date: July 4, 1996
DRAWN BY: Panos Arvanitis


Sensor Signal Interface
U7
A1
A2
A3
A4
A5
A6
A7
A8
B1
B2
B3
B4
B5
B6
B7
B8
WRST1
WSYNC1
IRFRONT
IRBACK
CONVREV
CONVFWD
G
DIR
74LS245
PWR
GND
RELAY
IN2
IN1
OUT
CONTROL
CW
R?
J5
Pin 3 is connected to ASE ground
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 9 of 10
Date: August 29, 1997
DRAWN BY: Panos Arvanitis

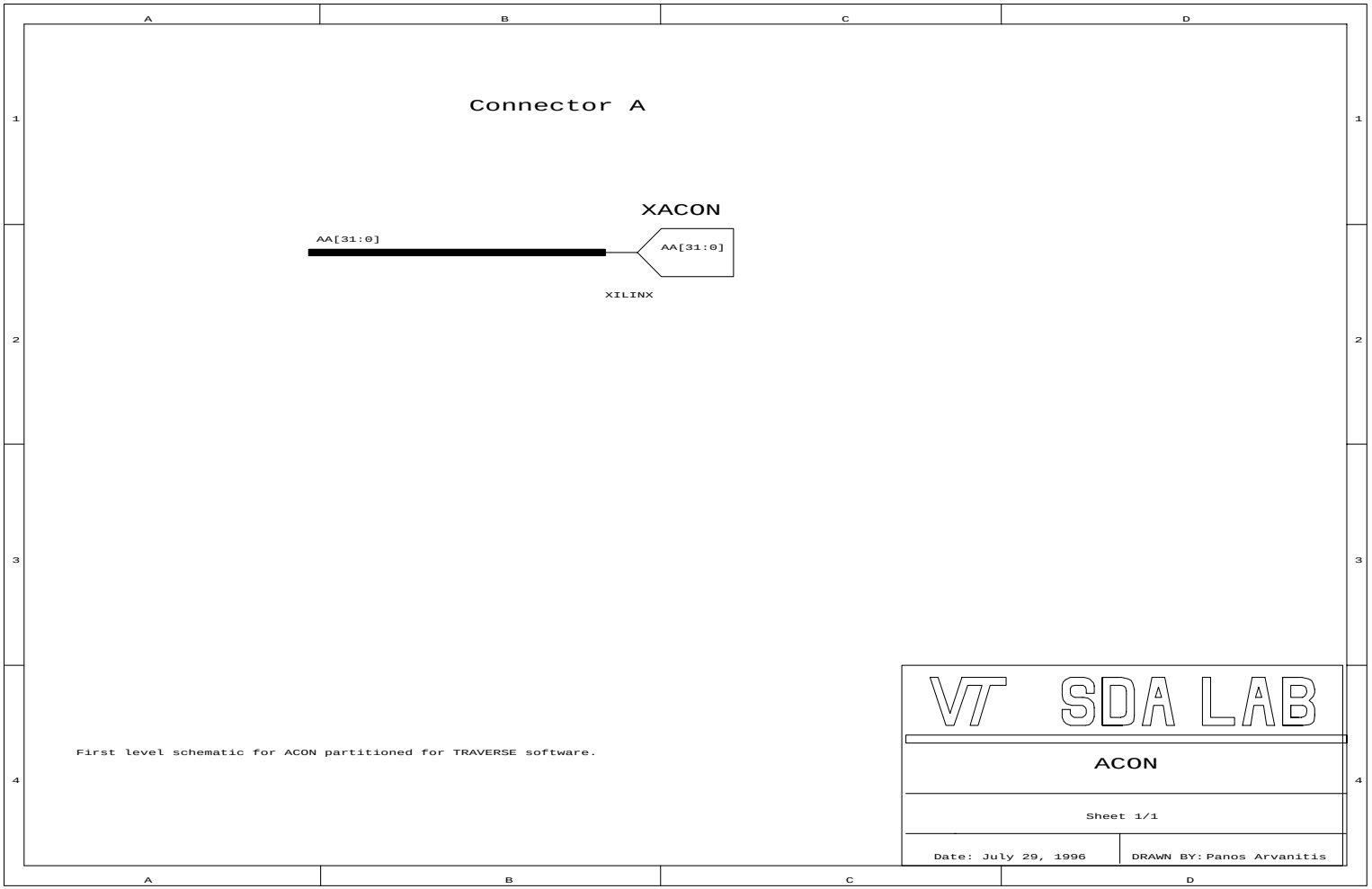

Zee Bus Interface
U?
MORRPHDAT7 2
MORRPHDAT6 3
MORRPHDAT5 4
MORRPHDAT4 5
MORRPHDAT3 6
MORRPHDAT2 7
MORRPHDAT1 8
MORRPHDAT0 9
A1 A2 A3 A4 A5 A6 A7 A8
B1 B2 B3 B4 B5 B6 B7 B8
18 MCONDAT7
17 MCONDAT6
16 MCONDAT5
15 MCONDAT4
14 MCONDAT3
13 MCONDAT2
12 MCONDAT1
11 MCONDAT0
G DIR
19 1 74LS245
PWR
GND
U?
MORRPHCSIG7 2
MORRPHCSIG6 3
MORRPHCSIG5 4
MORRPHCSIG4 5
MORRPHCSIG3 6
MORRPHCSIG2 7
MORRPHCSIG1 8
MORRPHCSIG0 9
18 MCONCSIG7
17 MCONCSIG6
16 MCONCSIG5
15 MCONCSIG4
14 MCONCSIG3
13 MCONCSIG2
12 MCONCSIG1
11 MCONCSIG0
74LS245
PWR
GND
U?
MCONDAT0 1
MCONDAT1 2
MCONDAT2 3
MCONDAT3 4
MCONDAT4 5
MCONDAT5 6
MCONDAT6 7
MCONDAT7 8
MCONSIG0 9
MCONSIG1 10
MCONSIG2 11
MCONSIG3 12
MCONSIG4 13
MCONSIG5 14
MCONSIG6 15
MCONSIG7 16
17
18
19
20
SIGNAL
SIGNAL
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
GND
VT SDA LAB
Differential Pair Interface Board (DPIB)
Sheet 10 of 10
Date: August 29, 1997
DRAWN BY: Panos Arvanitis

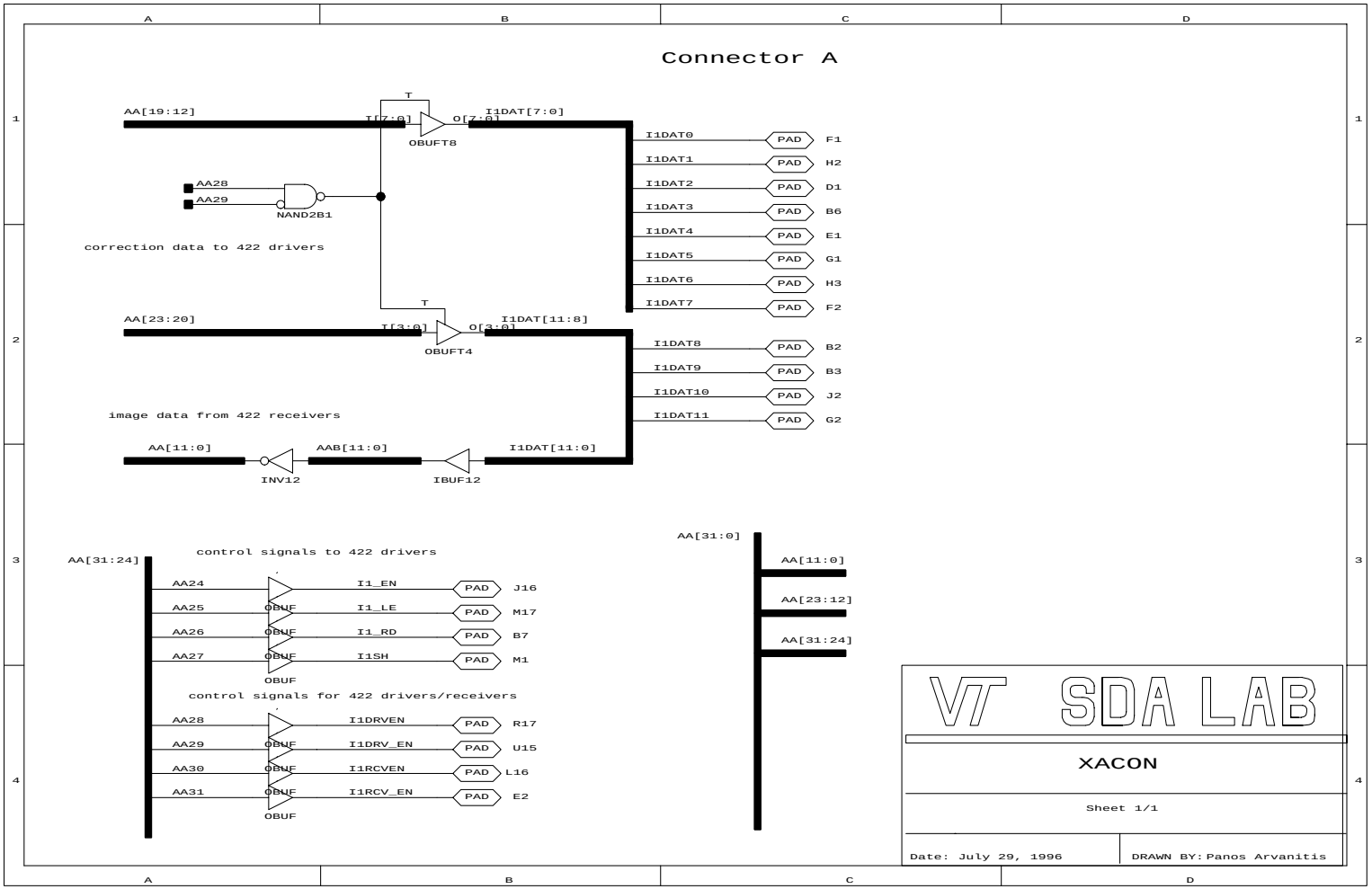
Appendix B. DPIB Logic (FPGA) Level Schematics

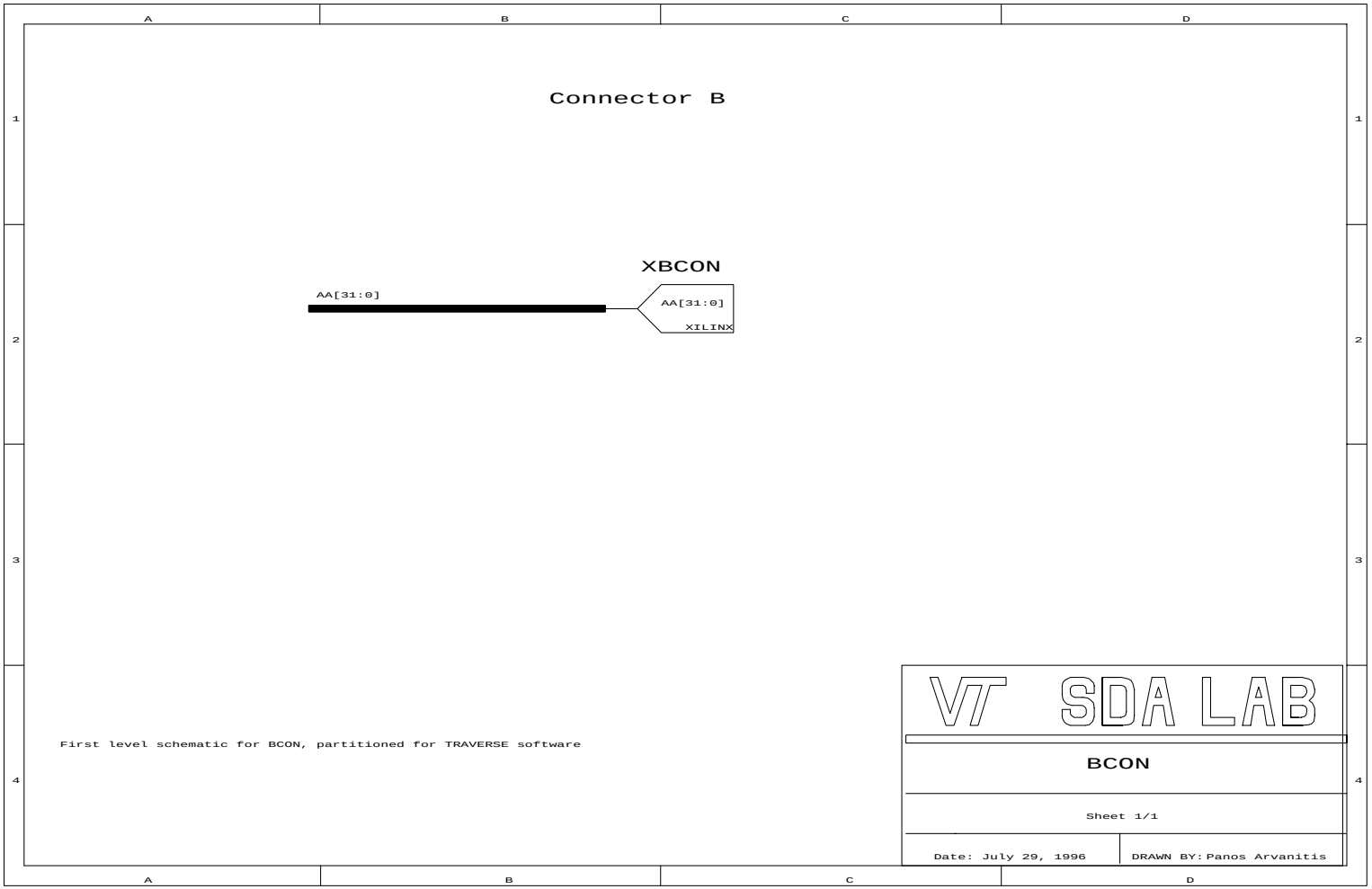
Appendix B contains the logic level schematics for the DPIB design. The modules shown here are used in the prototype system DPIB design. The MDS top level schematic is shown, followed by a more detailed schematic of each underlying symbol.

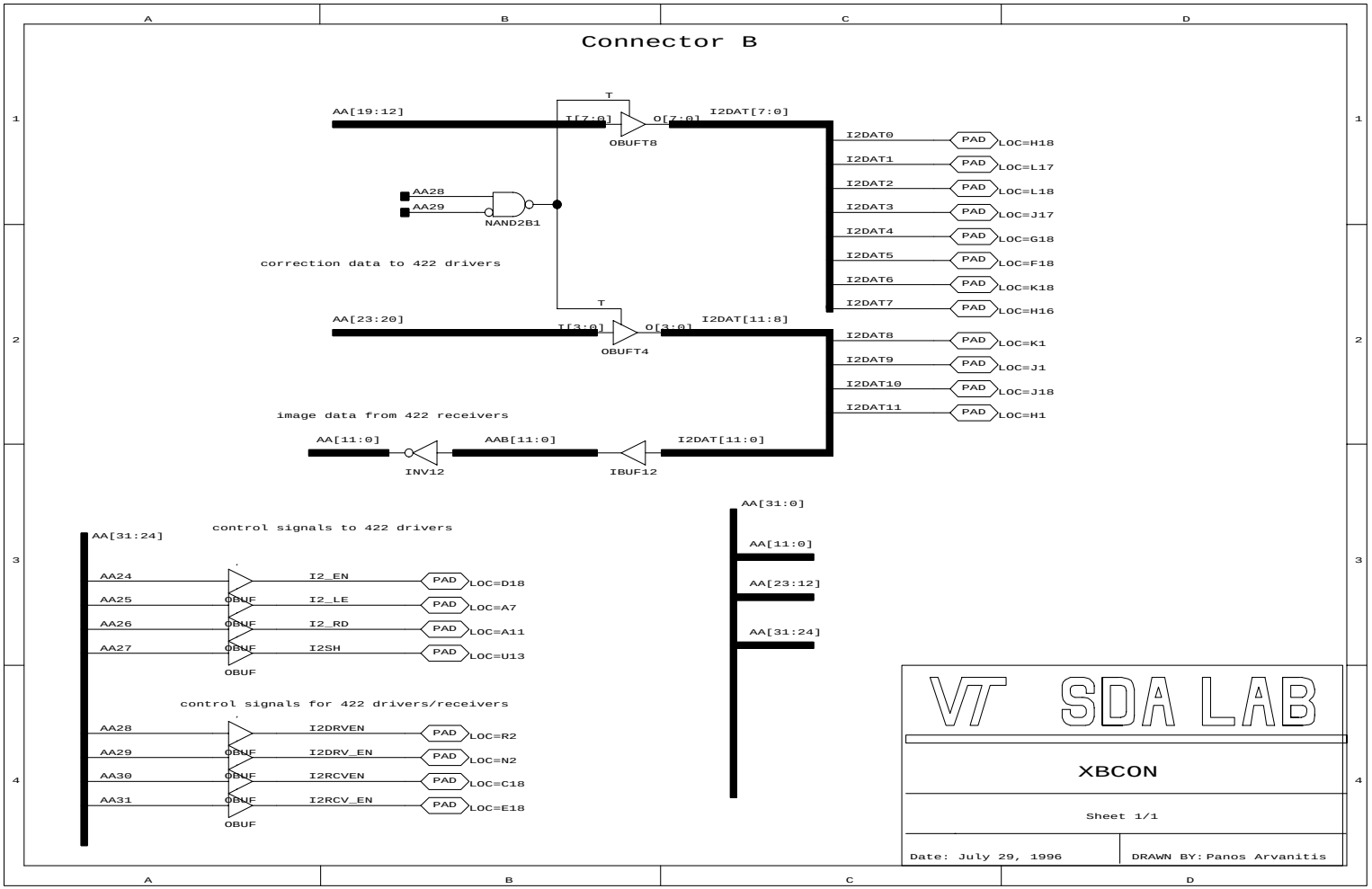

FPGA circuitry for DPIB with AS&E system.
CHECK
CHK1
ASE2SUIT
Transmission
CON1
ACON
AC2ASE[31:0]
WHLSIGS[3:0],FASTSIGS[3:0]
ASE1
MBA[15:0]
CHAN[3:0]=0000
MULTIPLEX
MBC[15:0]
forward scatter
ASE2SUIT
CON2
BCON
BC2ASE[31:0]
WHLSIGS[3:0],SIGS[3:0]
ASE2
BM[15:0]
CHAN[3:0]=0001
MULTIPLEX4
SUIT2ZEE_SLOW
MCON
CM[15:0]
MZ[15:0]
S3
ZD[15:0]
CON5
PASS
output to MORRPH board
backward scatter
ASE2SUIT
CON3
CCON
CC2ASE[31:0]
WHLSIGS[3:0],SIGS[3:0]
ASE3
CHAN[3:0]=0010
CONTROL
WSIGS[3:0]
SIGS[3:0]
FASTSIGS[3:0]
WHLSIGS[3:0]
SIGS[3:0]
FASTSIGS[3:0]
D-sub connector
DCON
CONTROL
CW[1:0]
SYNCS[1:0]
CONTROL2
VT SDA LAB
FAA
Mod: THD 8/4/97
FGPA circuitry for DPIB to interface AS&E
scatter and transmission system with MORRPH board.
Date: August 1, 1996
DRAWN BY: Panos Arvanitis
A
B
C
D
1
2
3
4

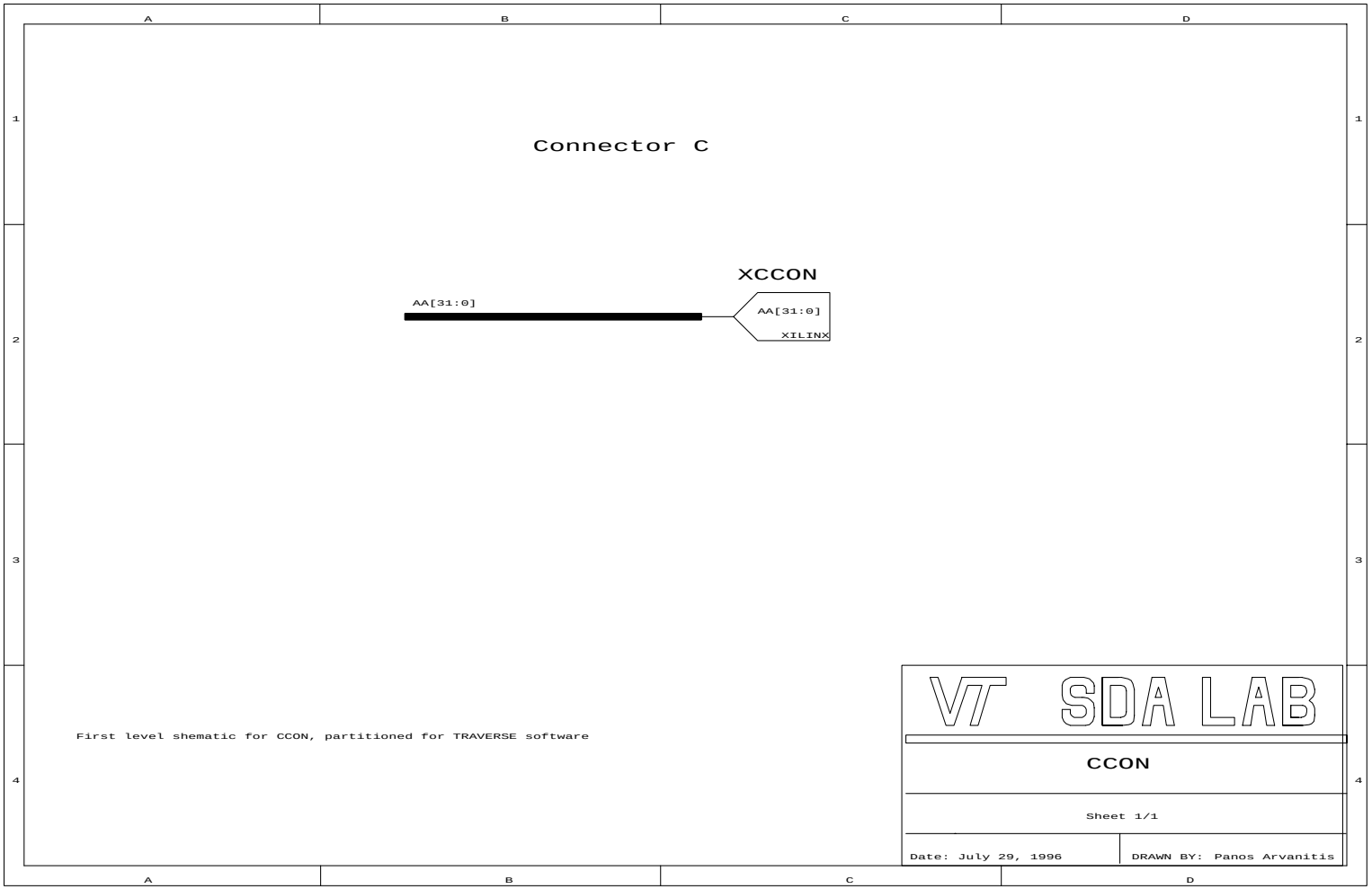
CON1

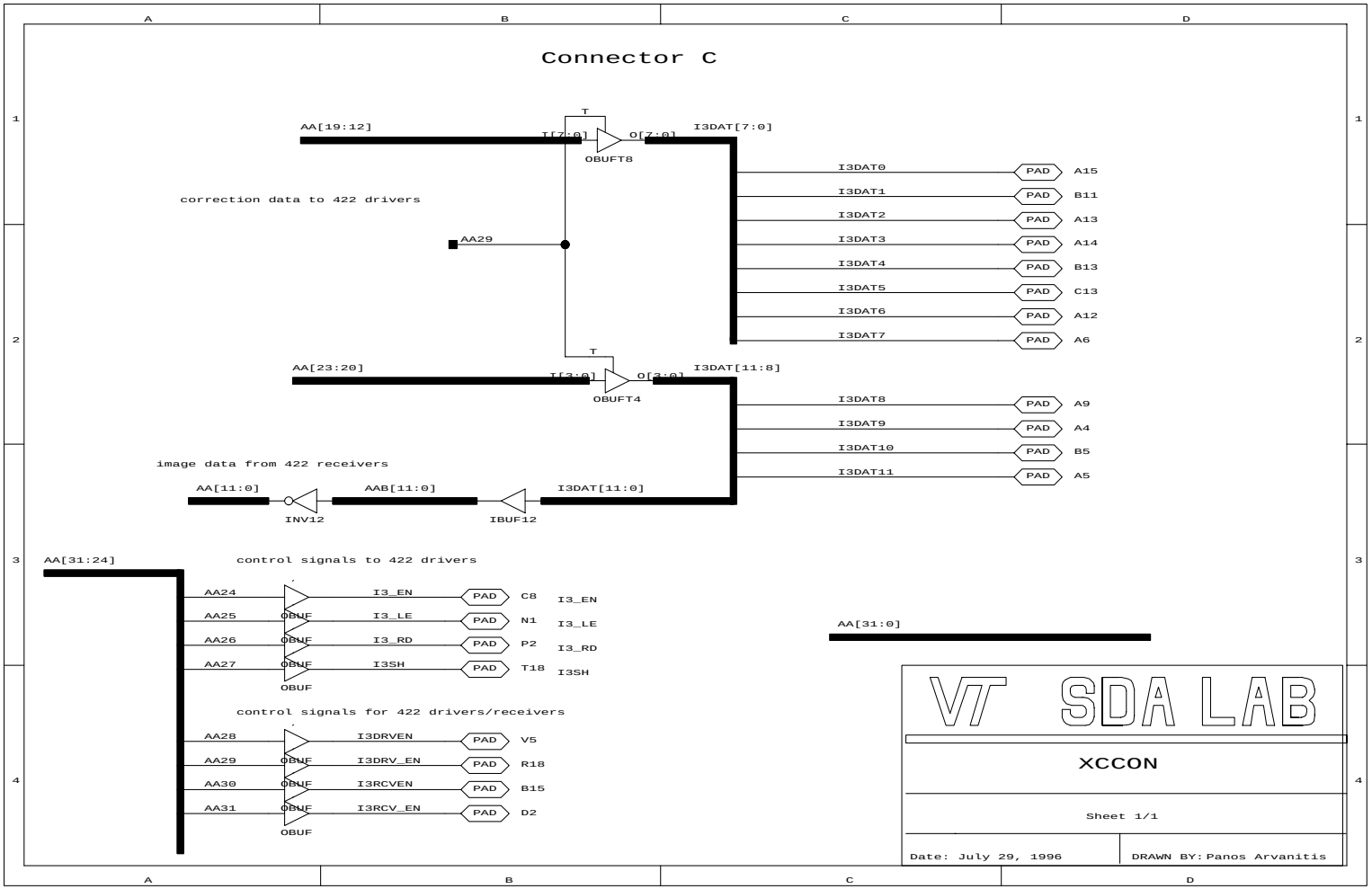

A
B
C
D
1
2
3
4
Connector A
XACON
AA[31:0]
AA[31:0]
XILINX
First level schematic for ACON partitioned for TRAVERSE software.
VT SDA LAB
ACON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis


Connector A
AA[19:12]
T
I[7:0]
O[7:0]
I1DAT[7:0]
OBUFT8
I1DAT0
I1DAT1
I1DAT2
I1DAT3
I1DAT4
I1DAT5
I1DAT6
I1DAT7
PAD
F1
H2
D1
B6
E1
G1
H3
F2
AA28
AA29
NAND2B1
correction data to 422 drivers
AA[23:20]
T
I[3:0]
O[3:0]
I1DAT[11:8]
OBUFT4
I1DAT8
I1DAT9
I1DAT10
I1DAT11
B2
B3
J2
G2
image data from 422 receivers
AA[11:0]
AAB[11:0]
I1DAT[11:0]
INV12
IBUF12
AA[31:0]
AA[11:0]
AA[23:12]
AA[31:24]
AA[31:24]
control signals to 422 drivers
AA24
AA25
AA26
AA27
OBUF
I1_EN
I1_LE
I1_RD
I1SH
J16
M17
B7
M1
control signals for 422 drivers/receivers
AA28
AA29
AA30
AA31
I1DRVEN
I1DRV_EN
I1RCVEN
I1RCV_EN
R17
U15
L16
E2
VT SDA LAB
XACON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis

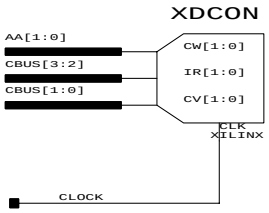

A
B
C
D
1
2
3
4
Connector B
XBCON
AA[31:0]
AA[31:0]
XILINX
First level schematic for BCON, partitioned for TRAVERSE software
VT SDA LAB
BCON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis


Connector B
AA[19:12]
T
I[7:0]
O[7:0]
OBUFT8
I2DAT[7:0]
I2DAT0
I2DAT1
I2DAT2
I2DAT3
I2DAT4
I2DAT5
I2DAT6
I2DAT7
PAD
LOC=H18
LOC=L17
LOC=L18
LOC=J17
LOC=G18
LOC=F18
LOC=K18
LOC=H16
AA28
AA29
NAND2B1
correction data to 422 drivers
AA[23:20]
T
I[3:0]
O[3:0]
OBUFT4
I2DAT[11:8]
I2DAT8
I2DAT9
I2DAT10
I2DAT11
LOC=K1
LOC=J1
LOC=J18
LOC=H1
image data from 422 receivers
AA[11:0]
INV12
AAB[11:0]
IBUF12
I2DAT[11:0]
AA[31:0]
AA[11:0]
AA[23:12]
AA[31:24]
control signals to 422 drivers
AA[31:24]
AA24
AA25
AA26
AA27
I2_EN
I2_LE
I2_RD
I2SH
OBUF
LOC=D18
LOC=A7
LOC=A11
LOC=U13
control signals for 422 drivers/receivers
AA28
AA29
AA30
AA31
I2DRVEN
I2DRV_EN
I2RCVEN
I2RCV_EN
LOC=R2
LOC=N2
LOC=C18
LOC=E18
VT SDA LAB
XBCON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis


A
B
C
D
1
2
3
4
Connector C
XCCON
AA[31:0]
AA[31:0]
XILINX
First level shematic for CCON, partitioned for TRAVERSE software
VT SDA LAB
CCON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis


Connector C
AA[19:12]
T
I[7:0]
O[7:0]
I3DAT[7:0]
OBUFT8
correction data to 422 drivers
AA29
I3DAT0
I3DAT1
I3DAT2
I3DAT3
I3DAT4
I3DAT5
I3DAT6
I3DAT7
PAD
A15
B11
A13
A14
B13
C13
A12
A6
AA[23:20]
I[3:0]
O[3:0]
I3DAT[11:8]
OBUFT4
I3DAT8
I3DAT9
I3DAT10
I3DAT11
A9
A4
B5
A5
image data from 422 receivers
AA[11:0]
AAB[11:0]
I3DAT[11:0]
INV12
IBUF12
AA[31:24]
control signals to 422 drivers
AA24
AA25
AA26
AA27
OBUF
I3_EN
I3_LE
I3_RD
I3SH
C8
N1
P2
T18
control signals for 422 drivers/receivers
AA28
AA29
AA30
AA31
I3DRVEN
I3DRV_EN
I3RCVEN
I3RCV_EN
V5
R18
B15
D2
AA[31:0]
VT SDA LAB
XCCON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis

Connector D



D-sub connector on right edge of ISA card

First level schematic for BCON, partitioned for TRAVERSE software

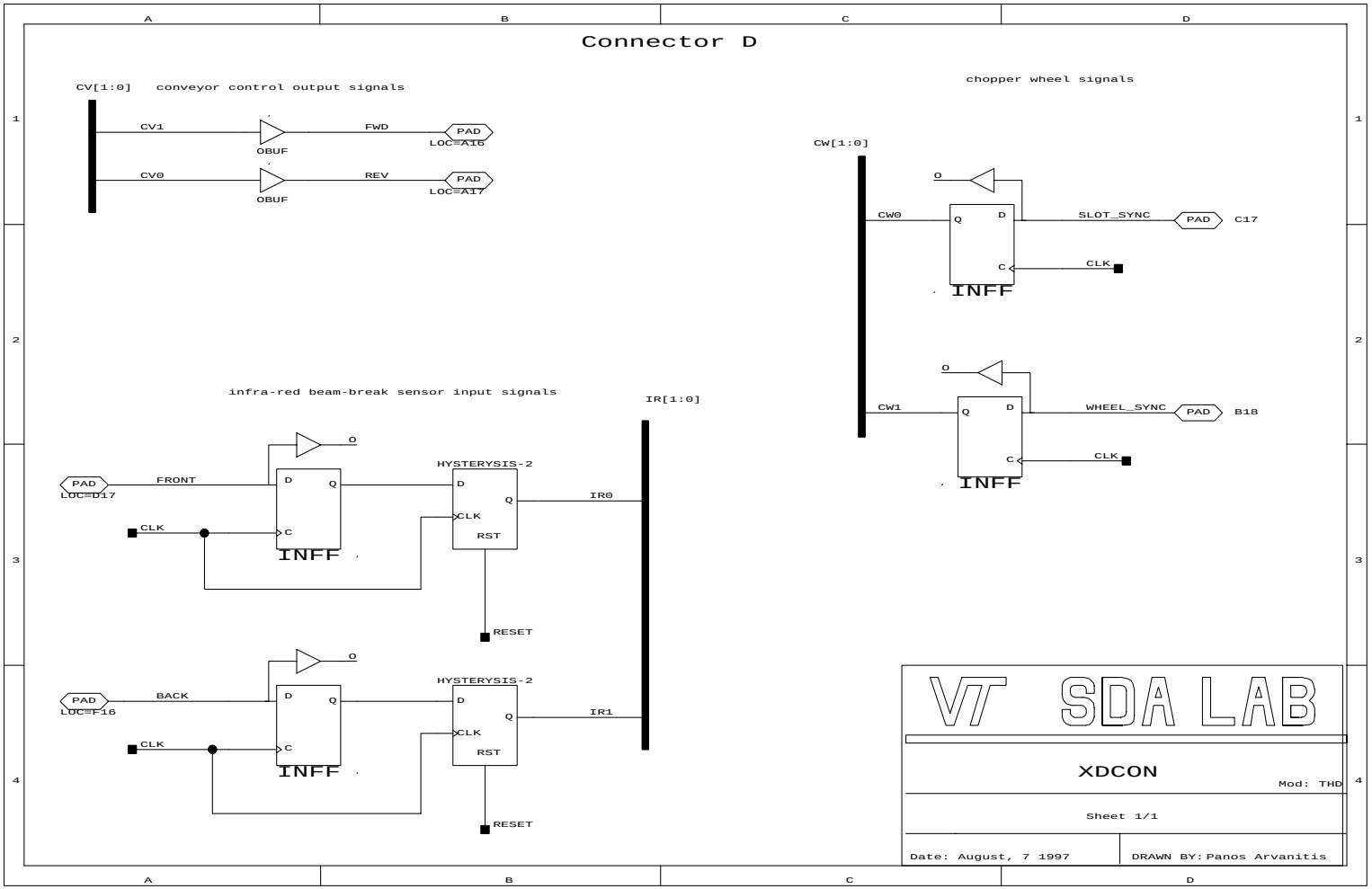
VT SDA LAB

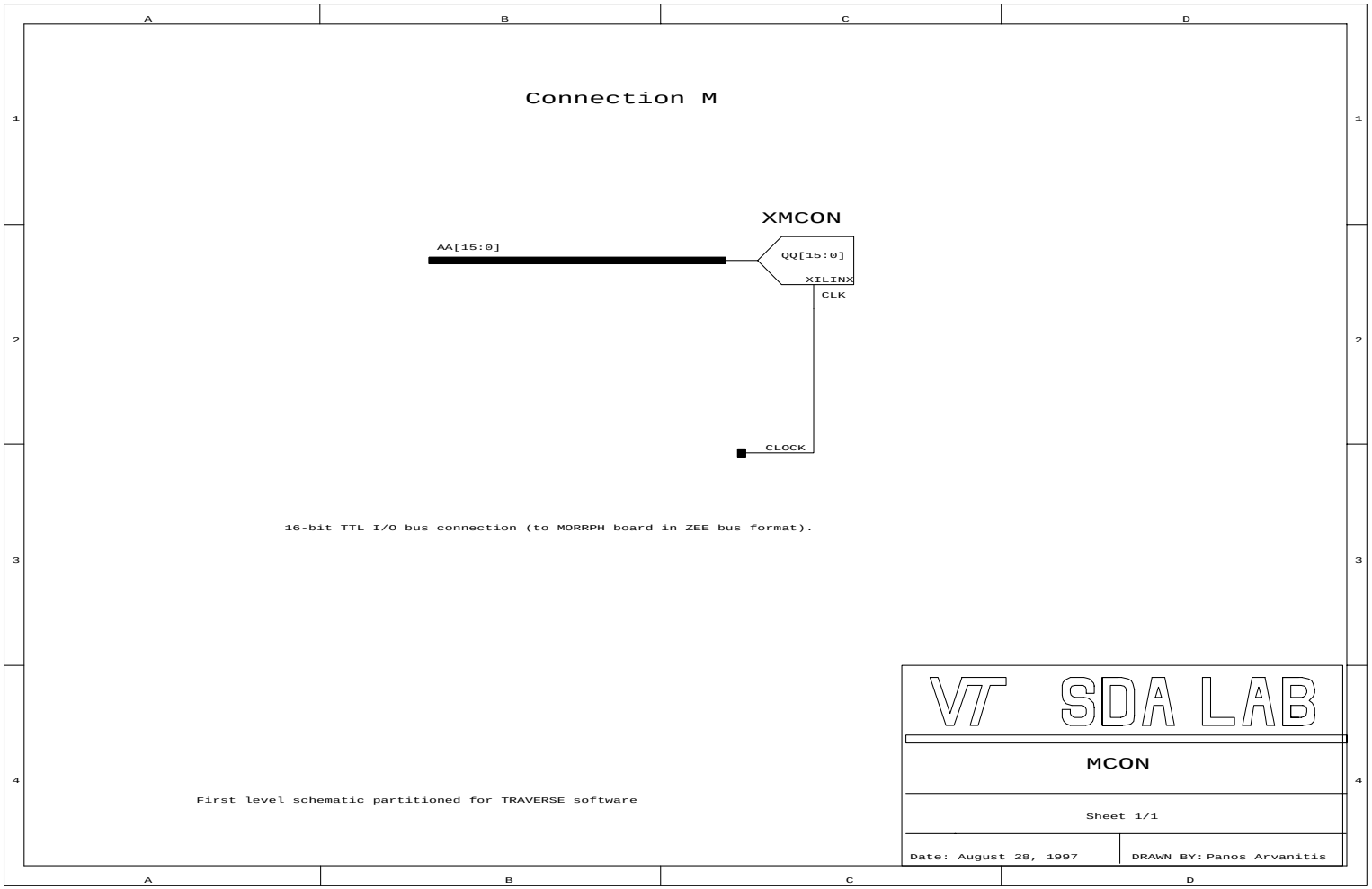
DCON

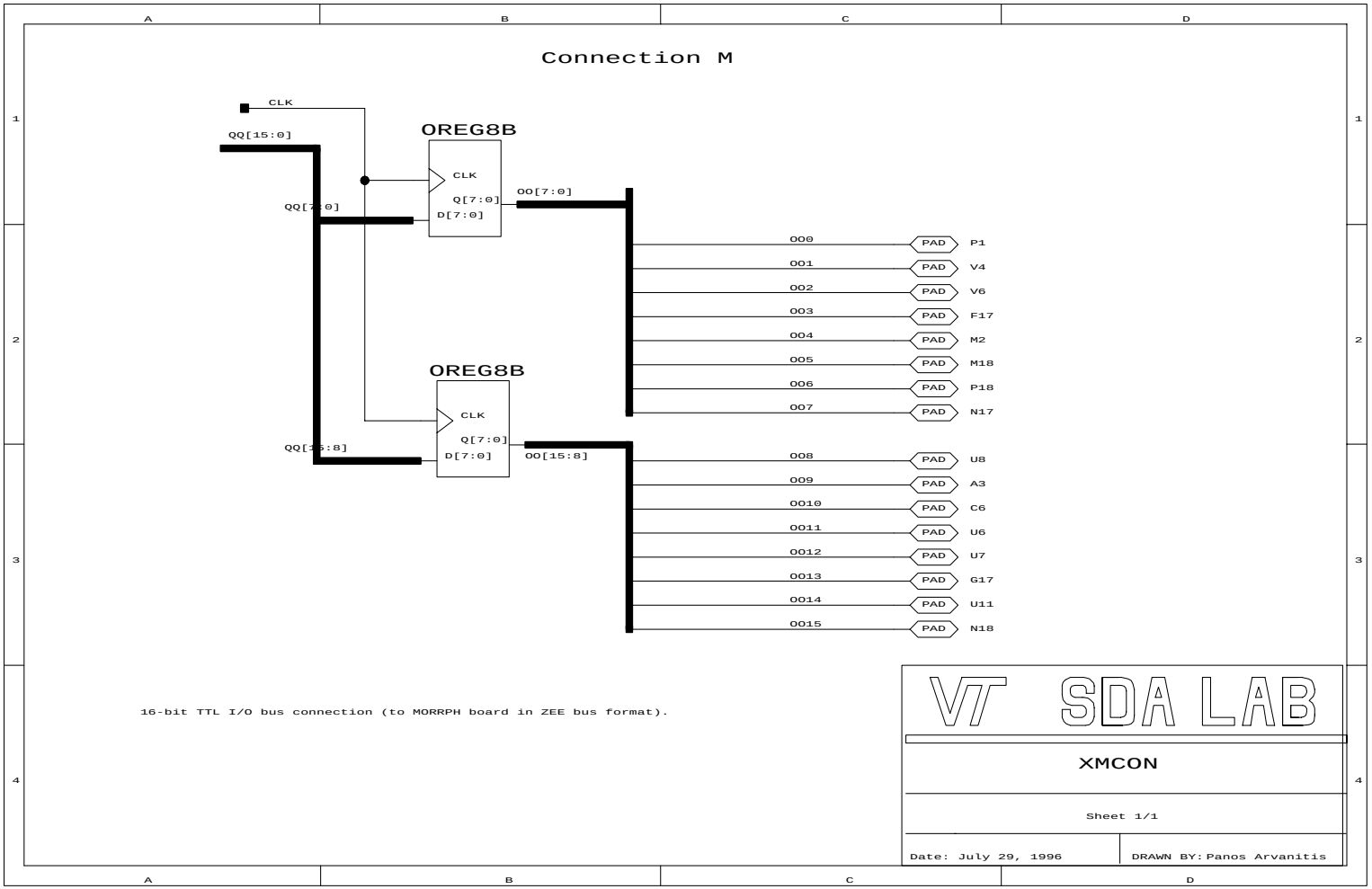
Mod: THD

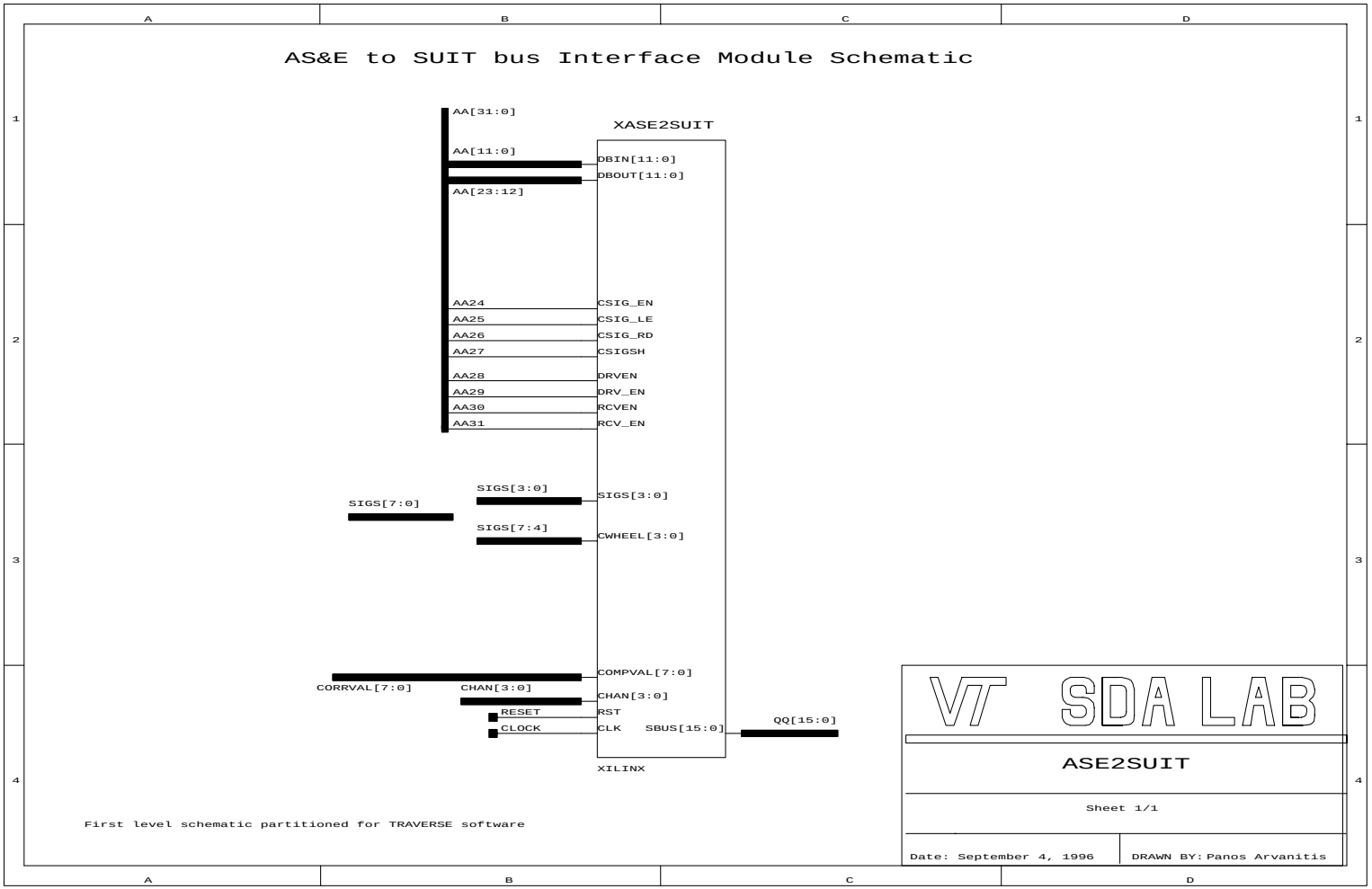
Sheet 1/1

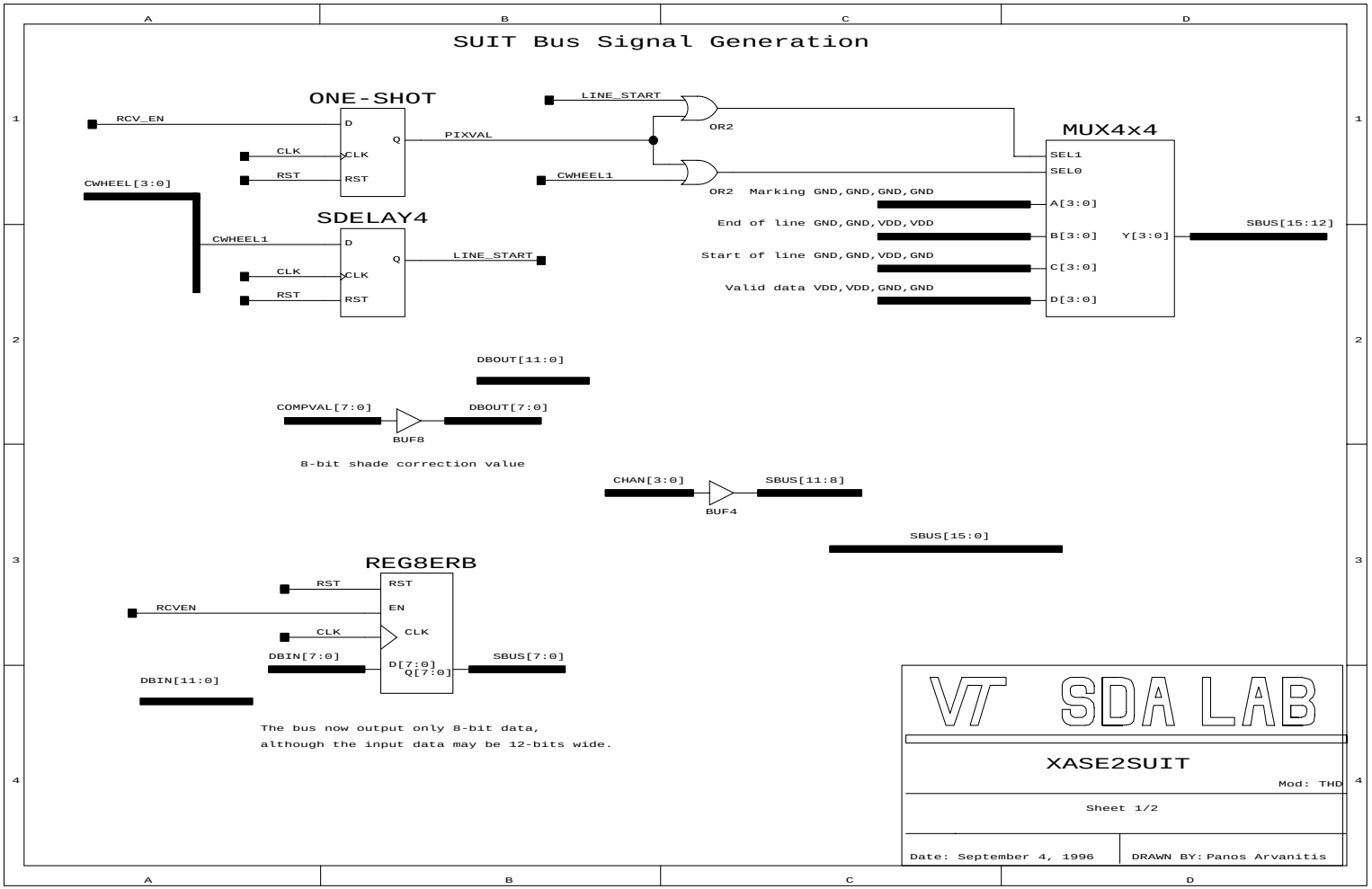
Date: August 9, 1997 | DRAWN BY:


Connector D
CV[1:0] conveyor control output signals
CV1
FWD
PAD
LOC=A16
OBUF
CV0
REV
PAD
LOC=A17
OBUF
chopper wheel signals
CW[1:0]
CW0
SLOT_SYNC
PAD
C17
CLK
INFF
CW1
WHEEL_SYNC
PAD
B18
CLK
INFF
infra-red beam-break sensor input signals
IR[1:0]
PAD
LOC=D17
FRONT
HYSTERYSIS-2
IR0
CLK
INFF
RESET
PAD
LOC=F16
BACK
HYSTERYSIS-2
IR1
CLK
INFF
RESET
VT SDA LAB
XDCON
Mod: THD
Sheet 1/1
Date: August, 7 1997
DRAWN BY: Panos Arvanitis


A
B
C
D
1
2
3
4
Connection M
XMCON
AA[15:0]
QQ[15:0]
XILINX
CLK
CLOCK
16-bit TTL I/O bus connection (to MORRPH board in ZEE bus format).
First level schematic partitioned for TRAVERSE software
VT SDA LAB
MCON
Sheet 1/1
Date: August 28, 1997
DRAWN BY: Panos Arvanitis

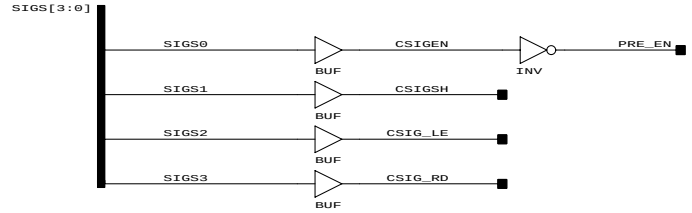

Connection M
CLK
QQ[15:0]
OREG8B
CLK
Q[7:0]
D[7:0]
QQ[7:0]
OO[7:0]
OO0
OO1
OO2
OO3
OO4
OO5
OO6
OO7
PAD
P1
V4
V6
F17
M2
M18
P18
N17
OREG8B
CLK
Q[7:0]
D[7:0]
QQ[15:8]
OO[15:8]
OO8
OO9
OO10
OO11
OO12
OO13
OO14
OO15
PAD
U8
A3
C6
U6
U7
G17
U11
N18
16-bit TTL I/O bus connection (to MORRPH board in ZEE bus format).
VT SDA LAB
XMCON
Sheet 1/1
Date: July 29, 1996
DRAWN BY: Panos Arvanitis


AS&E to SUIT bus Interface Module Schematic
XASE2SUIT
AA[31:0]
AA[11:0]
DBIN[11:0]
DBOUT[11:0]
AA[23:12]
AA24
AA25
AA26
AA27
AA28
AA29
AA30
AA31
CSIG_EN
CSIG_LE
CSIG_RD
CSIGSH
DRVEN
DRV_EN
RCVEN
RCV_EN
SIGS[3:0]
SIGS[7:0]
SIGS[7:4]
CWHEEL[3:0]
COMPVAL[7:0]
CORRVAL[7:0]
CHAN[3:0]
RESET
CLOCK
RST
CLK
SBUS[15:0]
QQ[15:0]
XILINX
First level schematic partitioned for TRAVERSE software
VT SDA LAB
ASE2SUIT
Sheet 1/1
Date: September 4, 1996
DRAWN BY: Panos Arvanitis


SUIT Bus Signal Generation
ONE-SHOT
RCV_EN
D
Q
CLK
CLK
RST
RST
PIXVAL
LINE_START
OR2
CWHEEL1
OR2
CWHEEL[3:0]
SDELAY4
CWHEEL1
D
Q
LINE_START
CLK
CLK
RST
RST
MUX4x4
SEL1
SEL0
Marking GND,GND,GND,GND
A[3:0]
End of line GND,GND,VDD,VDD
B[3:0]
Y[3:0]
SBUS[15:12]
Start of line GND,GND,VDD,GND
C[3:0]
Valid data VDD,VDD,GND,GND
D[3:0]
DBOUT[11:0]
COMPVAL[7:0]
DBOUT[7:0]
BUF8
8-bit shade correction value
CHAN[3:0]
SBUS[11:8]
BUF4
SBUS[15:0]
REG8ERB
RST
RST
RCVEN
EN
CLK
CLK
DBIN[7:0]
D[7:0]
Q[7:0]
SBUS[7:0]
DBIN[11:0]
The bus now output only 8-bit data,
although the input data may be 12-bits wide.
VT SDA LAB
XASE2SUIT
Mod: THD
Sheet 1/2
Date: September 4, 1996
DRAWN BY: Panos Arvanitis

Control Signal Generation

AS&E control signals to 422 drivers



These are the timing values for a 14.318MHz oscillator assumming that CLK is off the oscillator, and not halved.

SIGPER = 224

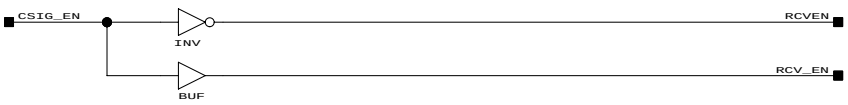
LTIM_EN = 215
HTIM_EN = 224

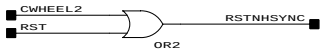
LTIMSH = 169
HTIMSH = 172

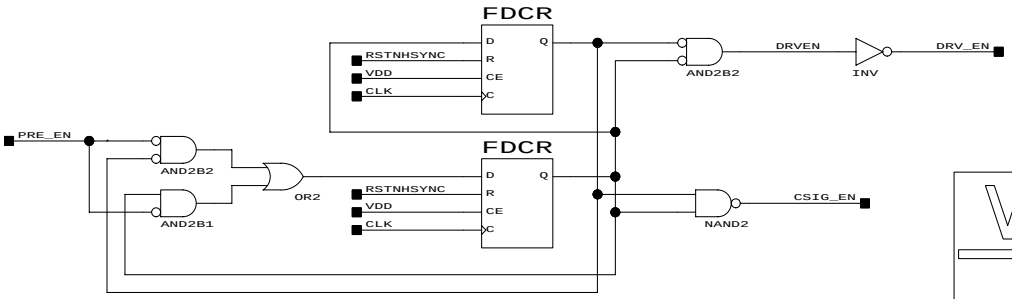
LTIM_LE = 182
HTIM_LE = 184

LTIM_RD = 176
HTIM_RD = 223

control signals for 422 drivers/receivers







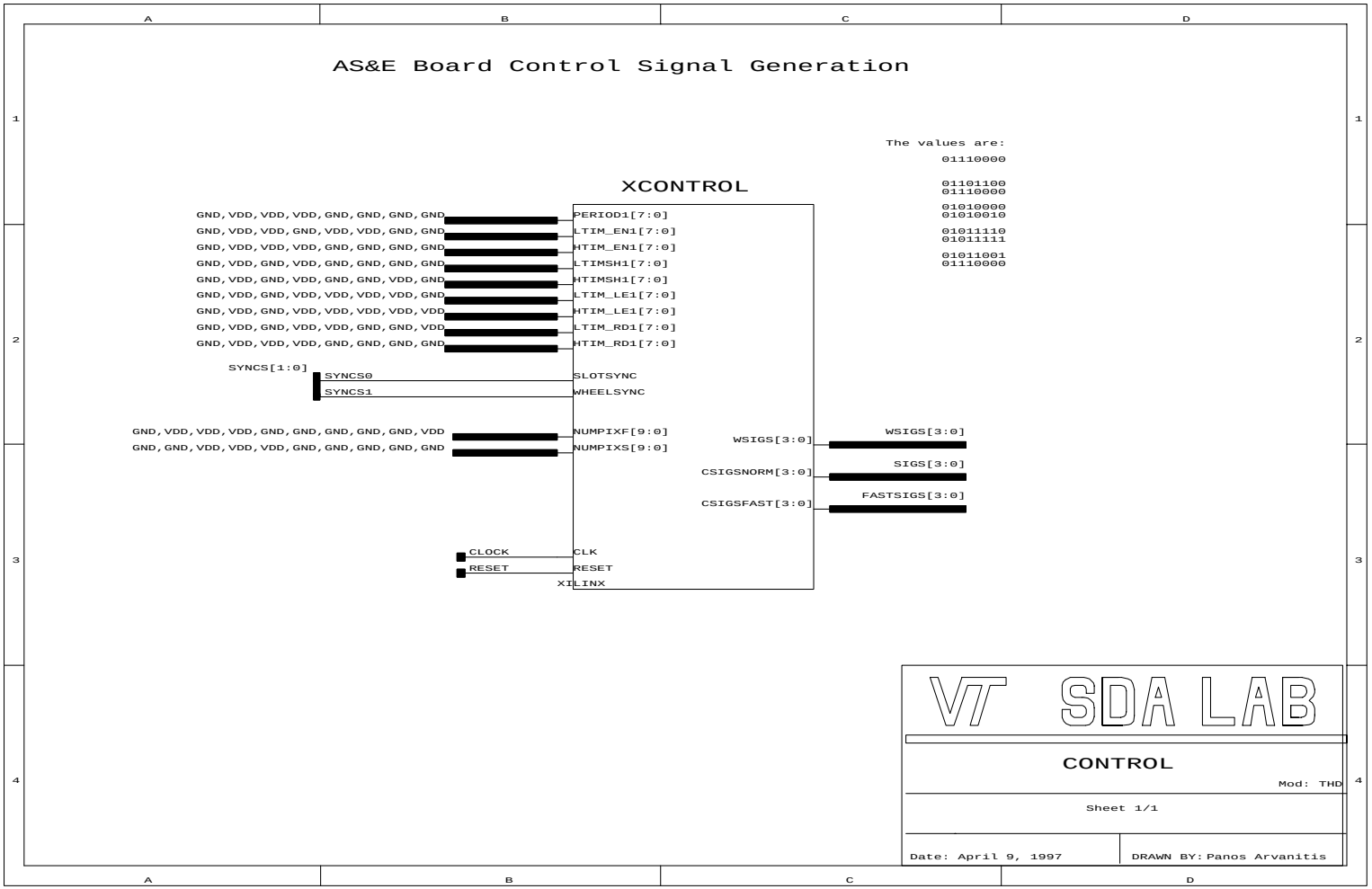
Introduce a time delay before 422 drivers are anabled, to avoid bus contention

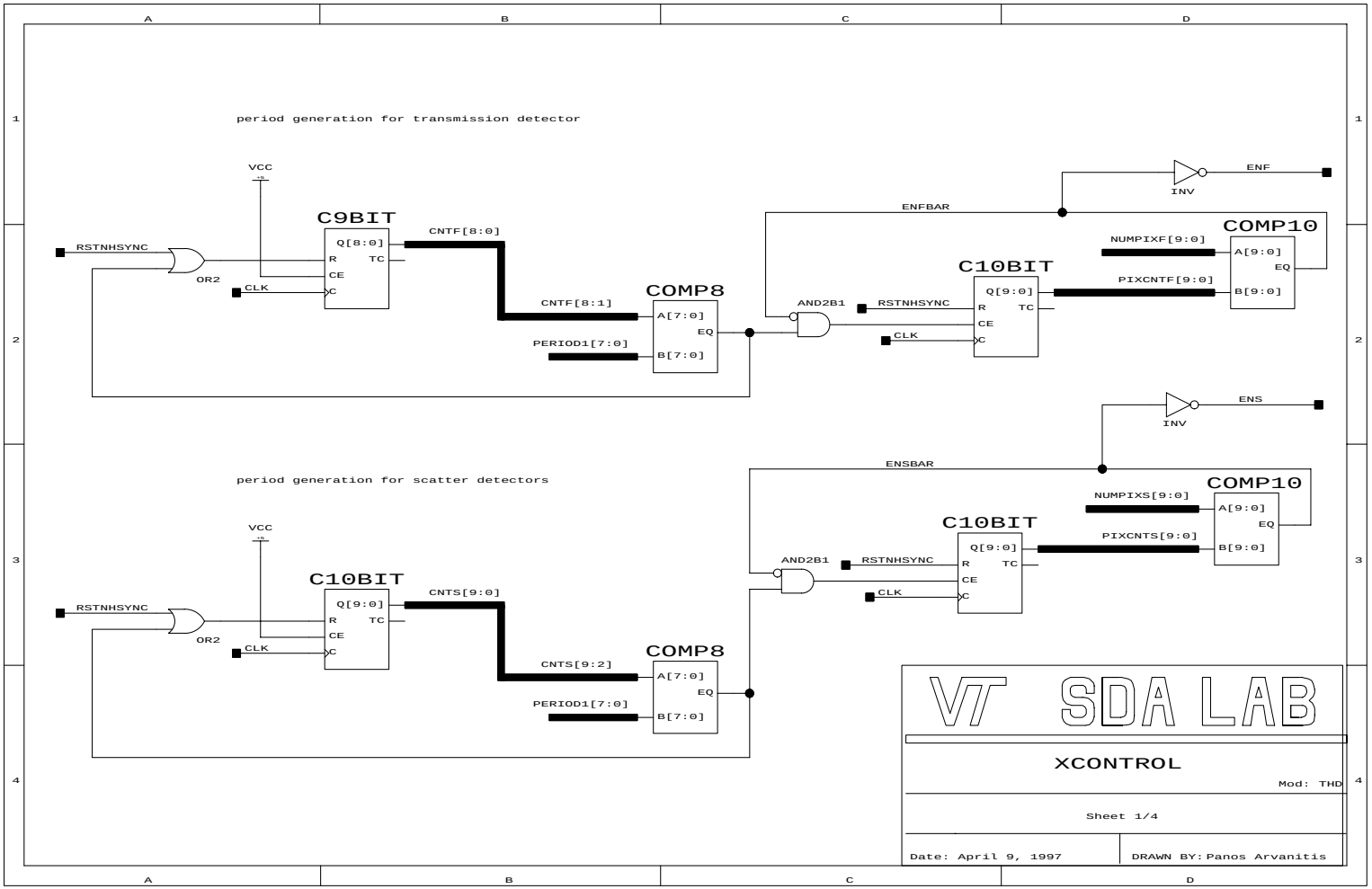
VT SDA LAB

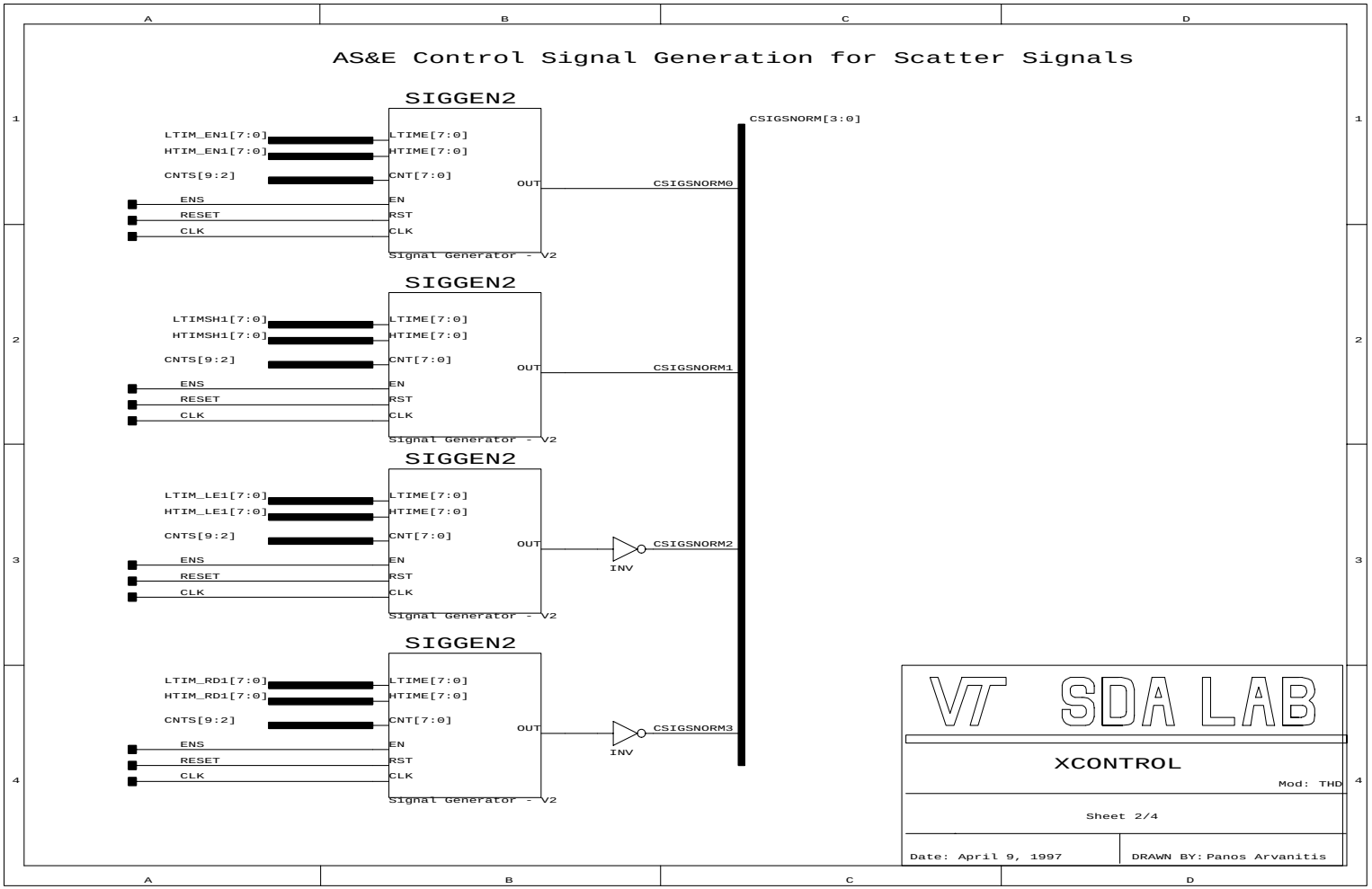
XASE2SUIT

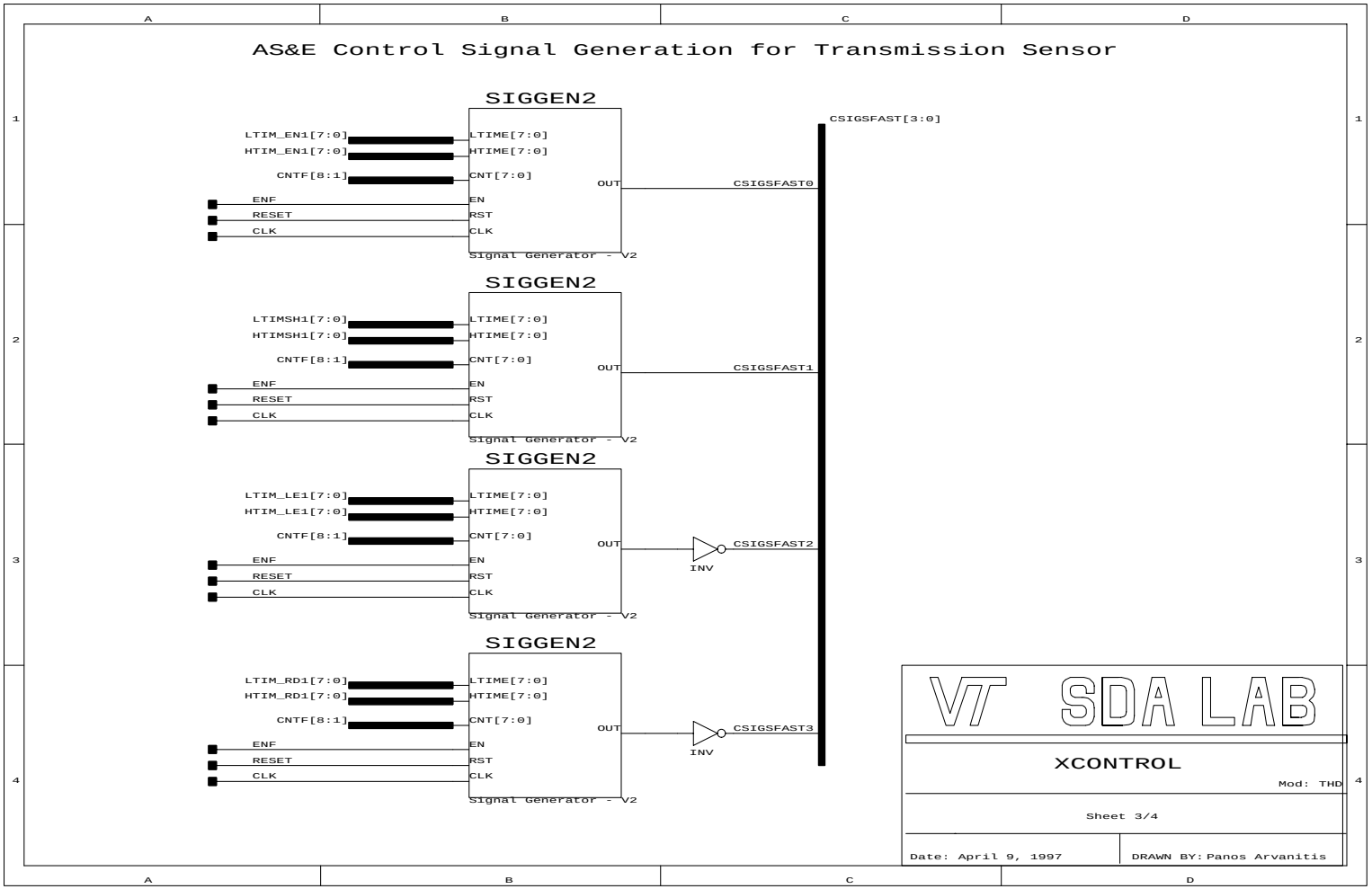
Sheet 2/2

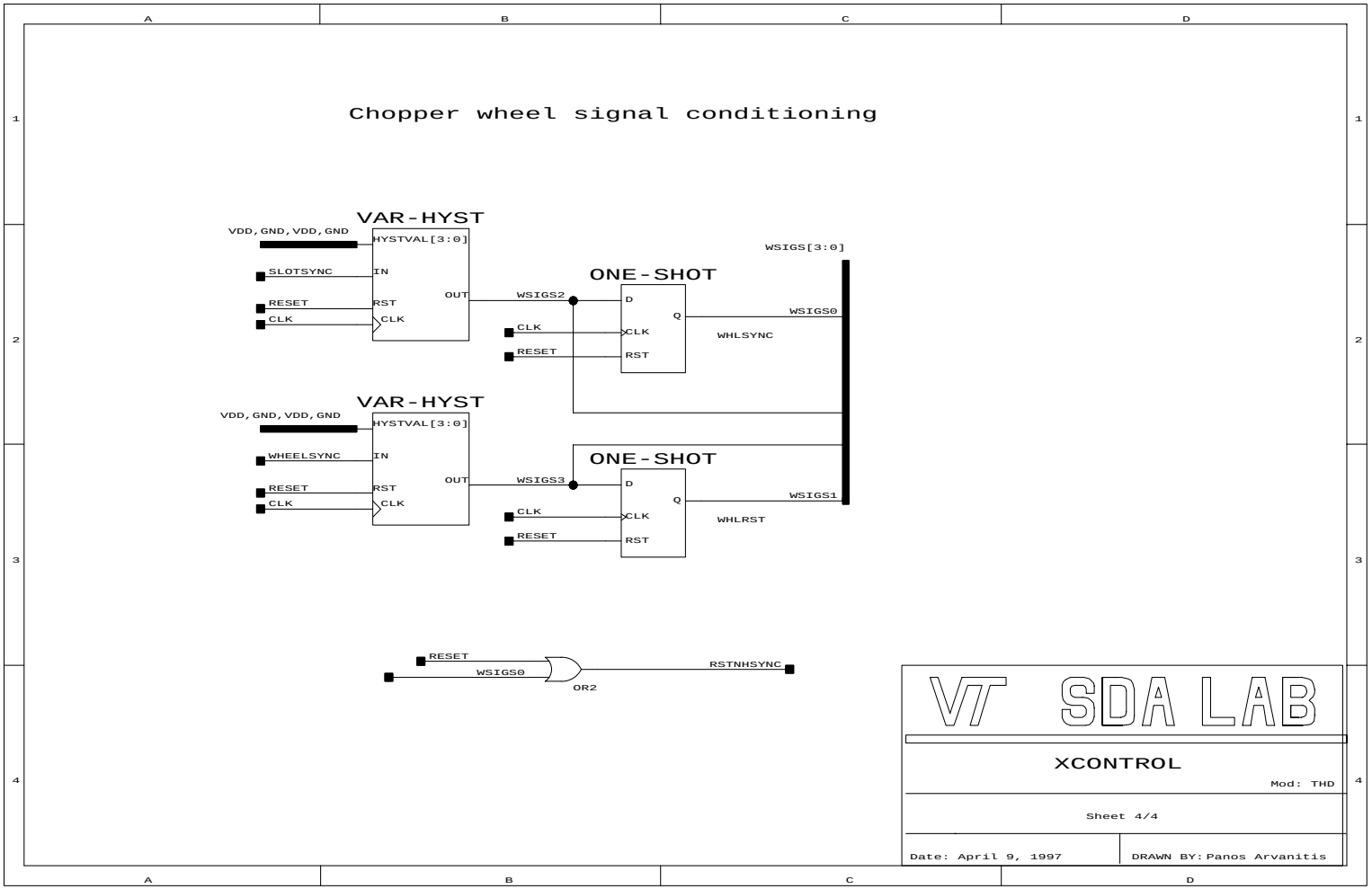
Date: September 15, 1996 | DRAWN BY: Panos Arvanitis

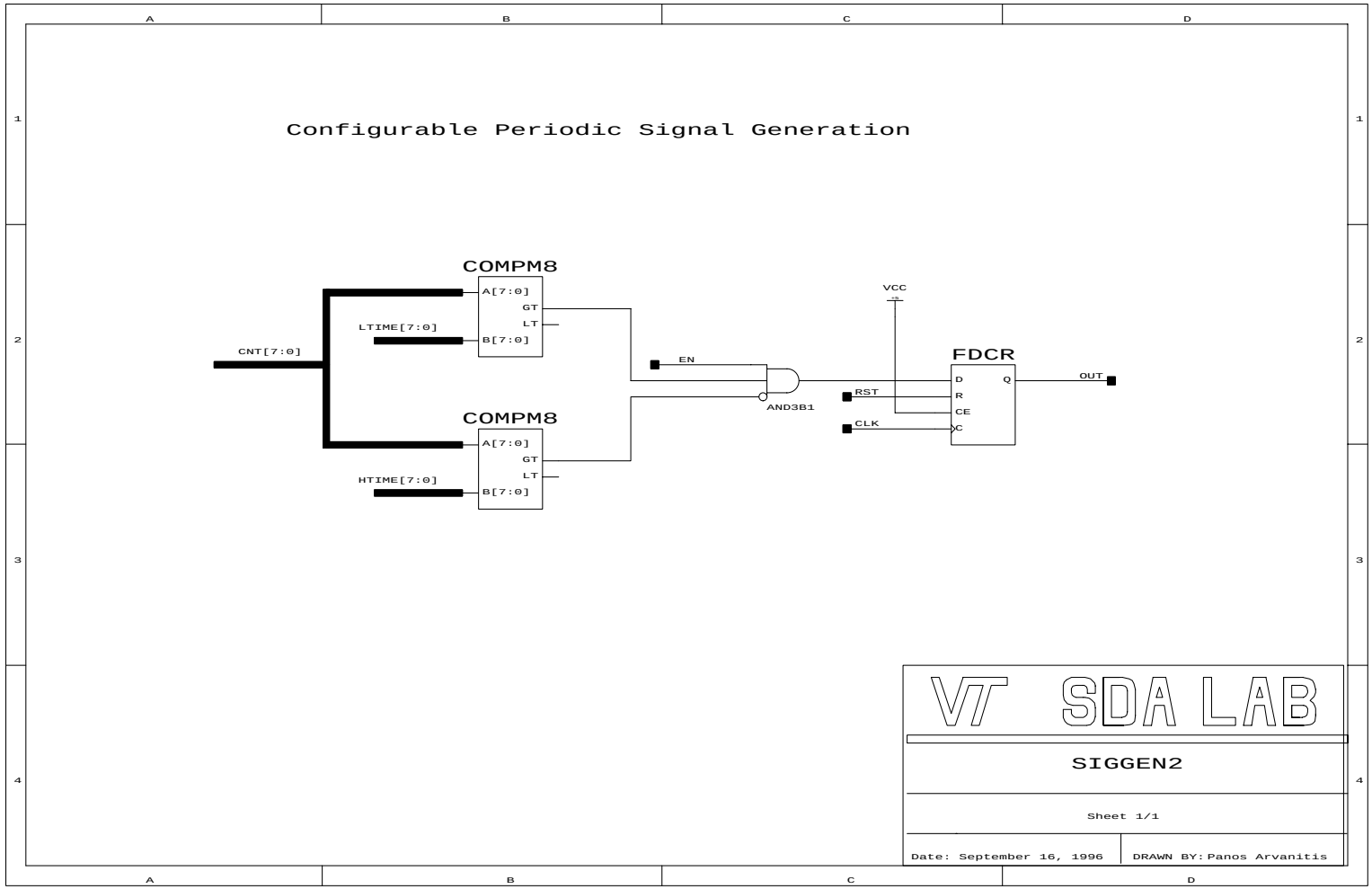

AS&E Board Control Signal Generation
XCONTROL
GND,VDD,VDD,VDD,GND,GND,GND,GND
PERIOD1[7:0]
GND,VDD,VDD,GND,VDD,VDD,GND,GND
LTIM_EN1[7:0]
GND,VDD,VDD,VDD,GND,GND,GND,GND
HTIM_EN1[7:0]
GND,VDD,GND,VDD,GND,GND,GND,GND
LTIMSH1[7:0]
GND,VDD,GND,VDD,GND,GND,VDD,GND
HTIMSH1[7:0]
GND,VDD,GND,VDD,VDD,VDD,VDD,GND
LTIM_LE1[7:0]
GND,VDD,GND,VDD,VDD,VDD,VDD,VDD
HTIM_LE1[7:0]
GND,VDD,GND,VDD,VDD,GND,GND,VDD
LTIM_RD1[7:0]
GND,VDD,VDD,VDD,GND,GND,GND,GND
HTIM_RD1[7:0]
SYNCS[1:0]
SYNCS0
SYNCS1
SLOTSYNC
WHEELSYNC
GND,VDD,VDD,VDD,GND,GND,GND,GND,GND,VDD
NUMPIXF[9:0]
GND,GND,VDD,VDD,VDD,GND,GND,GND,GND,GND
NUMPIXS[9:0]
WSIGS[3:0]
WSIGS[3:0]
CSIGSNORM[3:0]
SIGS[3:0]
CSIGSFAST[3:0]
FASTSIGS[3:0]
CLOCK
RESET
CLK
RESET
XILINX
The values are:
01110000
01101100
01110000
01010000
01010010
01011110
01011111
01011001
01110000
VT SDA LAB
CONTROL
Mod: THD
Sheet 1/1
Date: April 9, 1997
DRAWN BY: Panos Arvanitis


period generation for transmission detector
VCC
C9BIT
Q[8:0]
R
CE
C
TC
RSTNHSYNC
OR2
CLK
CNTF[8:0]
CNTF[8:1]
COMP8
A[7:0]
EQ
B[7:0]
PERIOD1[7:0]
AND2B1
RSTNHSYNC
C10BIT
Q[9:0]
R
CE
C
TC
CLK
ENFBAR
INV
ENF
COMP10
NUMPIXF[9:0]
A[9:0]
EQ
B[9:0]
PIXCNTF[9:0]
period generation for scatter detectors
VCC
C10BIT
Q[9:0]
R
CE
C
TC
RSTNHSYNC
OR2
CLK
CNTS[9:0]
CNTS[9:2]
COMP8
A[7:0]
EQ
B[7:0]
PERIOD1[7:0]
AND2B1
RSTNHSYNC
C10BIT
Q[9:0]
R
CE
C
TC
CLK
ENSBAR
INV
ENS
COMP10
NUMPIXS[9:0]
A[9:0]
EQ
B[9:0]
PIXCNTS[9:0]
VT SDA LAB
XCONTROL
Mod: THD
Sheet 1/4
Date: April 9, 1997
DRAWN BY: Panos Arvanitis


AS&E Control Signal Generation for Scatter Signals
SIGGEN2
LTIM_EN1[7:0]
HTIM_EN1[7:0]
CNTS[9:2]
ENS
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
CSIGSNORM0
Signal Generator - V2
SIGGEN2
LTIMSH1[7:0]
HTIMSH1[7:0]
CNTS[9:2]
ENS
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
CSIGSNORM1
Signal Generator - V2
SIGGEN2
LTIM_LE1[7:0]
HTIM_LE1[7:0]
CNTS[9:2]
ENS
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
INV
CSIGSNORM2
Signal Generator - V2
SIGGEN2
LTIM_RD1[7:0]
HTIM_RD1[7:0]
CNTS[9:2]
ENS
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
INV
CSIGSNORM3
Signal Generator - V2
CSIGSNORM[3:0]
VT SDA LAB
XCONTROL
Mod: THD
Sheet 2/4
Date: April 9, 1997
DRAWN BY: Panos Arvanitis

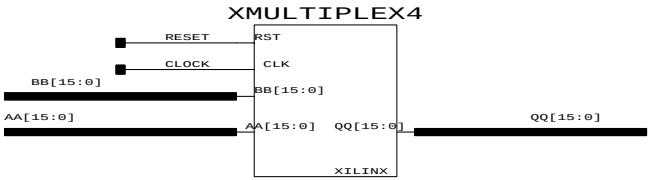

AS&E Control Signal Generation for Transmission Sensor
SIGGEN2
LTIM_EN1[7:0]
HTIM_EN1[7:0]
CNTF[8:1]
ENF
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
Signal Generator - V2
CSIGSFAST0
SIGGEN2
LTIMSH1[7:0]
HTIMSH1[7:0]
CNTF[8:1]
ENF
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
Signal Generator - V2
CSIGSFAST1
SIGGEN2
LTIM_LE1[7:0]
HTIM_LE1[7:0]
CNTF[8:1]
ENF
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
Signal Generator - V2
INV
CSIGSFAST2
SIGGEN2
LTIM_RD1[7:0]
HTIM_RD1[7:0]
CNTF[8:1]
ENF
RESET
CLK
LTIME[7:0]
HTIME[7:0]
CNT[7:0]
EN
RST
CLK
OUT
Signal Generator - V2
INV
CSIGSFAST3
CSIGSFAST[3:0]
VT SDA LAB
XCONTROL
Mod: THD
Sheet 3/4
Date: April 9, 1997
DRAWN BY: Panos Arvanitis


Chopper wheel signal conditioning
VAR-HYST
VDD,GND,VDD,GND
HYSTVAL[3:0]
SLOTSYNC
IN
RESET
RST
CLK
CLK
OUT
WSIGS2
ONE-SHOT
D
Q
CLK
RESET
CLK
RST
WSIGS[3:0]
WSIGS0
WHLSYNC
VAR-HYST
VDD,GND,VDD,GND
HYSTVAL[3:0]
WHEELSYNC
IN
RESET
RST
CLK
CLK
OUT
WSIGS3
ONE-SHOT
D
Q
CLK
RESET
CLK
RST
WSIGS1
WHLRST
RESET
WSIGS0
OR2
RSTNHSYNC
VT SDA LAB
XCONTROL
Mod: THD
Sheet 4/4
Date: April 9, 1997
DRAWN BY: Panos Arvanitis


Configurable Periodic Signal Generation
COMPM8
A[7:0]
GT
LT
B[7:0]
LTIME[7:0]
CNT[7:0]
COMPM8
A[7:0]
GT
LT
B[7:0]
HTIME[7:0]
EN
AND3B1
VCC
FDCR
D
Q
R
CE
C
RST
CLK
OUT
VT SDA LAB
SIGGEN2
Sheet 1/1
Date: September 16, 1996
DRAWN BY: Panos Arvanitis

This module multiplexes two input SUIT channels onto

a single SUIT output bus



compiled performance: 45.1 ns FF-FF

FG = 181
FF = 168
CLB = 197

.prt file

FG = 196
FF = 168
CLB = 132

Note: FIFOs are 2-words deep, input AA has priority.

VT SDA LAB

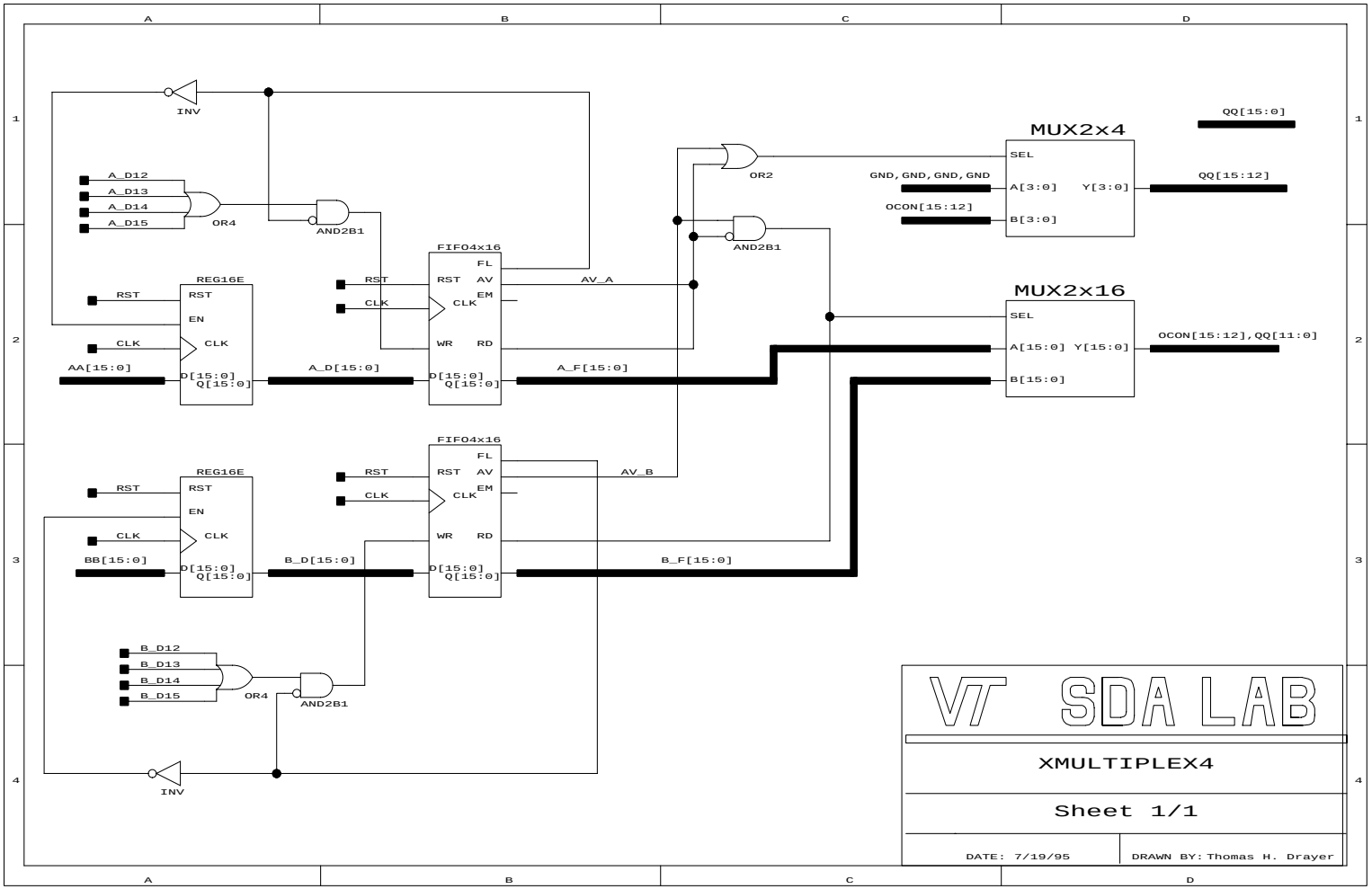
MULTIPLEX4

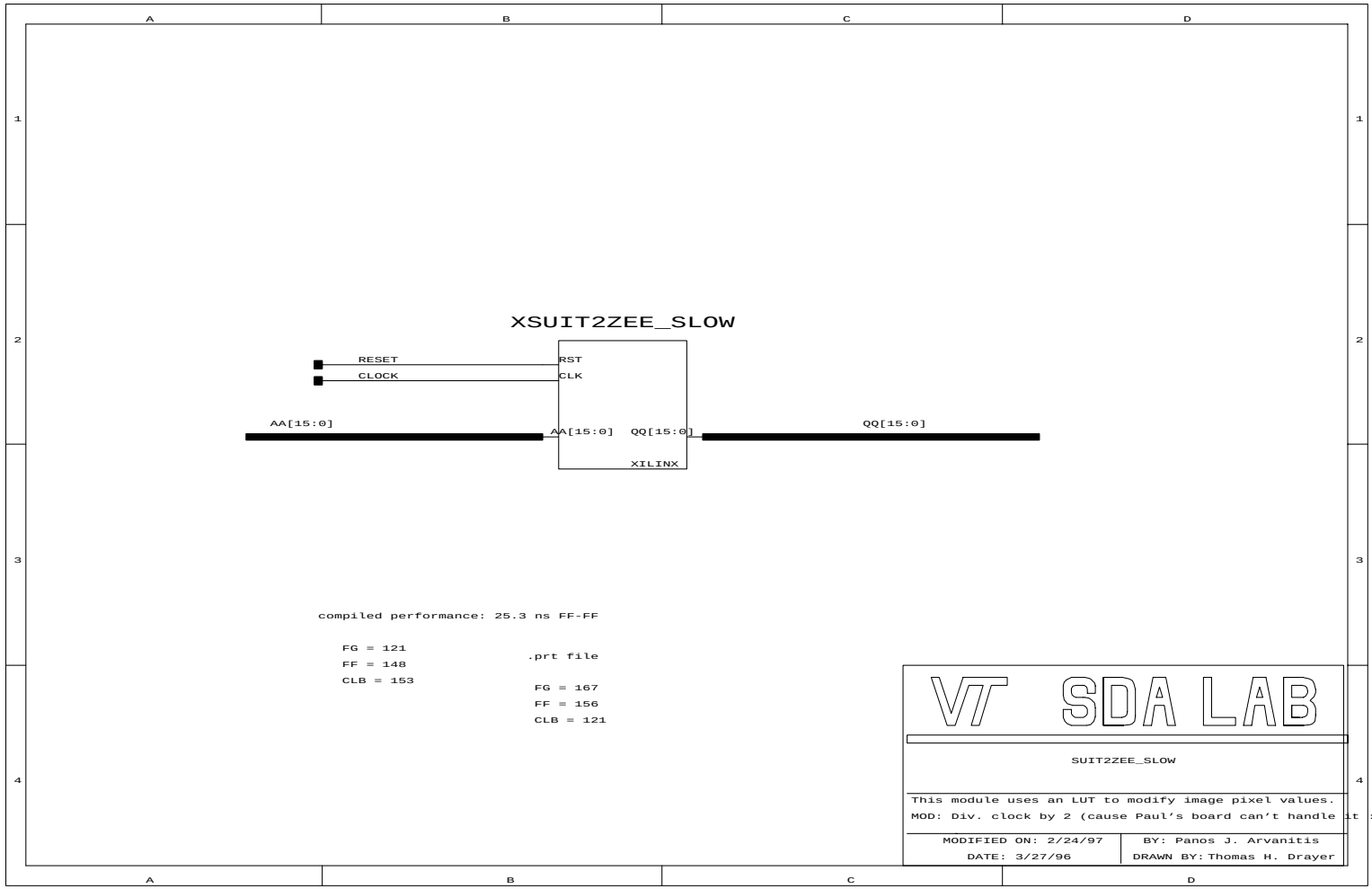
Sheet 1/1

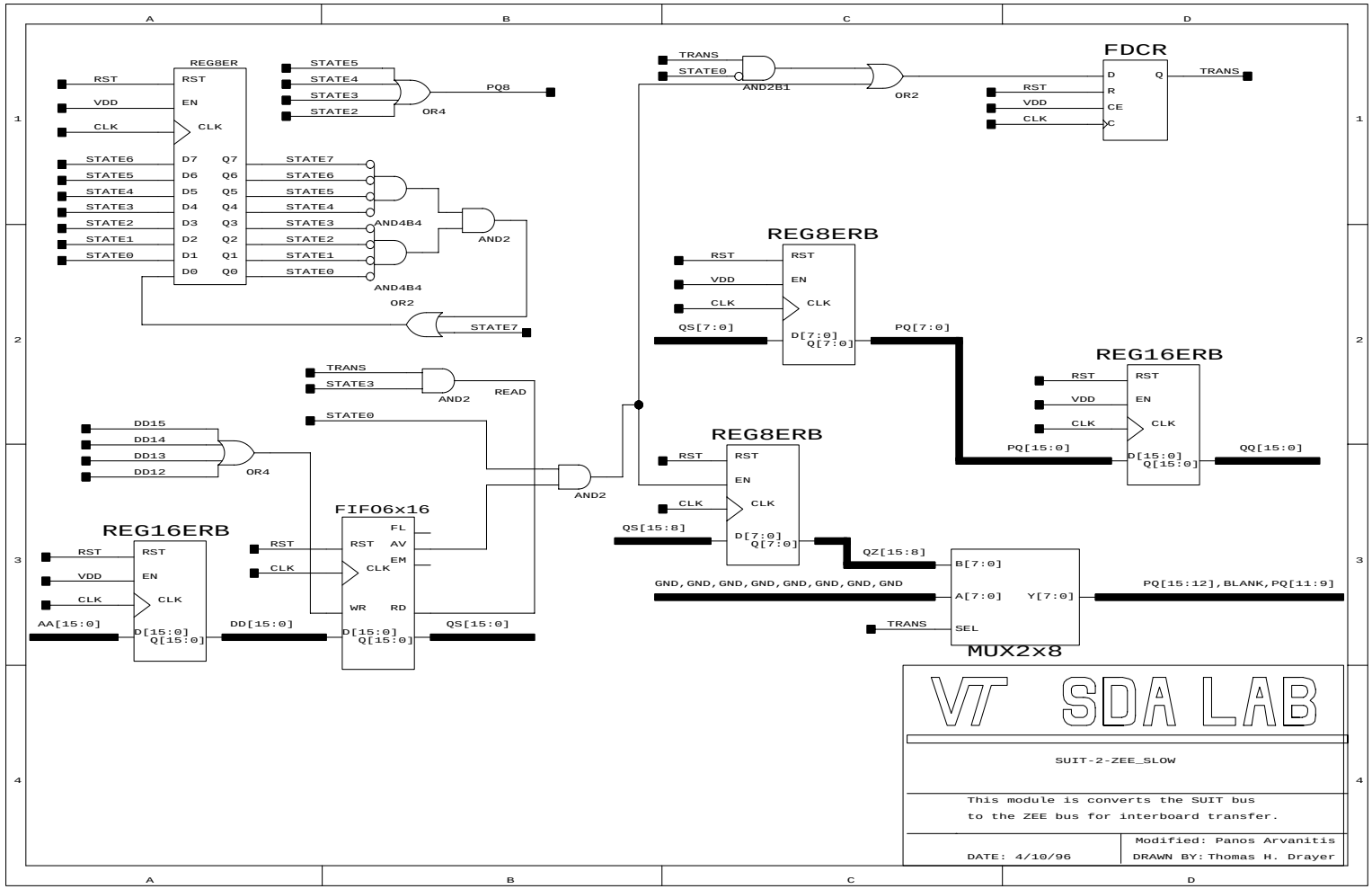
DATE: 7/19/95

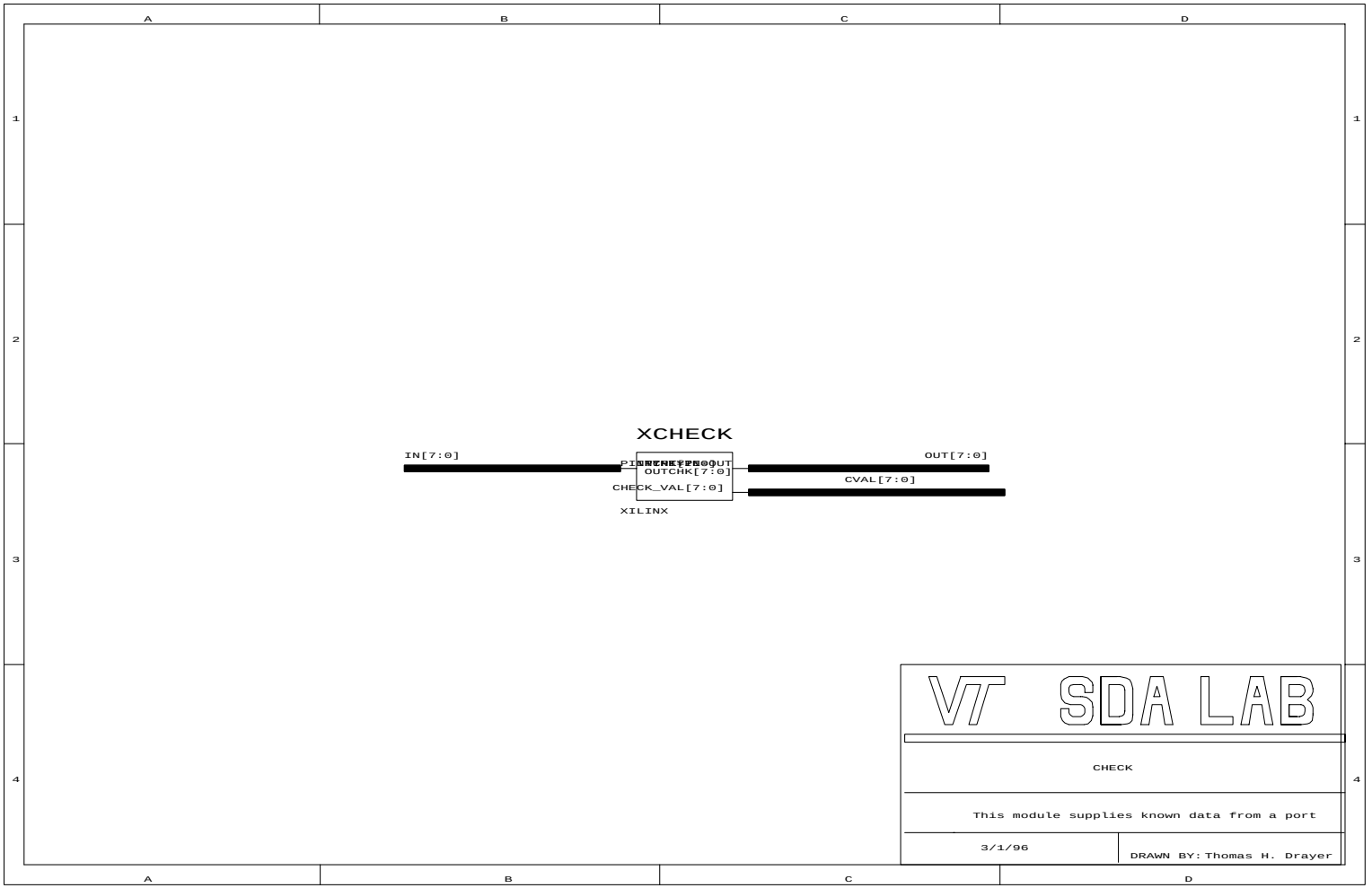
DRAWN BY: Thomas H. Drayer

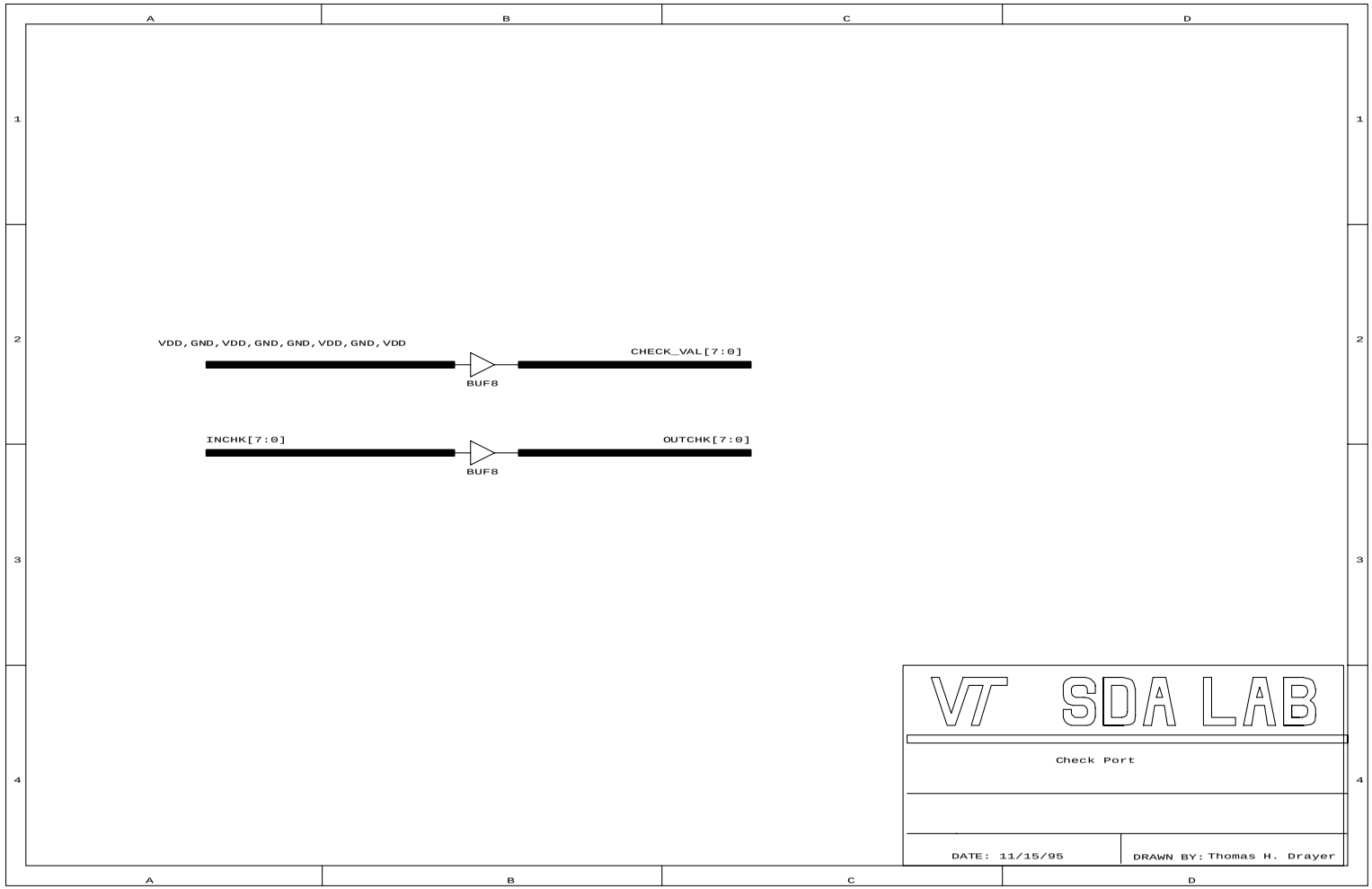
Location: /igor/hardware/morrph/suit2


A
B
C
D
1
2
3
4
INV
A_D12
A_D13
A_D14
A_D15
OR4
AND2B1
REG16E
RST
EN
CLK
AA[15:0]
D[15:0]
Q[15:0]
A_D[15:0]
FIFO4x16
FL
AV
EM
WR
RD
AV_A
A_F[15:0]
OR2
AND2B1
MUX2x4
SEL
A[3:0]
B[3:0]
Y[3:0]
GND,GND,GND,GND
OCON[15:12]
QQ[15:0]
QQ[15:12]
MUX2x16
A[15:0]
B[15:0]
Y[15:0]
OCON[15:12],QQ[11:0]
BB[15:0]
B_D[15:0]
AV_B
B_F[15:0]
B_D12
B_D13
B_D14
B_D15
VT SDA LAB
XMULTIPLEX4
Sheet 1/1
DATE: 7/19/95
DRAWN BY: Thomas H. Drayer


A
B
C
D
1
2
3
4
XSUIT2ZEE_SLOW
RESET
CLOCK
RST
CLK
AA[15:0]
AA[15:0]
QQ[15:0]
QQ[15:0]
XILINX
compiled performance: 25.3 ns FF-FF
FG = 121
FF = 148
CLB = 153
.prt file
FG = 167
FF = 156
CLB = 121
VT SDA LAB
SUIT2ZEE_SLOW
This module uses an LUT to modify image pixel values.
MOD: Div. clock by 2 (cause Paul's board can't handle it :))
MODIFIED ON: 2/24/97
BY: Panos J. Arvanitis
DATE: 3/27/96
DRAWN BY: Thomas H. Drayer


REG8ER
RST
VDD
CLK
EN
CLK
STATE6
STATE5
STATE4
STATE3
STATE2
STATE1
STATE0
D7 D6 D5 D4 D3 D2 D1 D0
Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0
STATE7
STATE6
STATE5
STATE4
STATE3
STATE2
STATE1
STATE0
STATE5
STATE4
STATE3
STATE2
OR4
PQ8
AND4B4
AND4B4
AND2
OR2
STATE7
TRANS
STATE0
AND2B1
OR2
FDCR
RST
VDD
CLK
D
R
CE
C
Q
TRANS
REG8ERB
RST
VDD
CLK
EN
CLK
QS[7:0]
D[7:0]
Q[7:0]
PQ[7:0]
REG16ERB
RST
VDD
CLK
EN
CLK
PQ[15:0]
D[15:0]
Q[15:0]
QQ[15:0]
TRANS
STATE3
AND2
READ
STATE0
DD15
DD14
DD13
DD12
OR4
AND2
REG8ERB
RST
CLK
EN
CLK
QS[15:8]
D[7:0]
Q[7:0]
QZ[15:8]
FIFO6x16
RST
CLK
FL
AV
EM
WR
RD
D[15:0]
Q[15:0]
REG16ERB
RST
VDD
CLK
EN
CLK
AA[15:0]
D[15:0]
Q[15:0]
DD[15:0]
DD[15:0]
QS[15:0]
GND,GND,GND,GND,GND,GND,GND,GND
B[7:0]
A[7:0]
Y[7:0]
SEL
TRANS
PQ[15:12],BLANK,PQ[11:9]
MUX2x8
VT SDA LAB
SUIT-2-ZEE_SLOW
This module is converts the SUIT bus
to the ZEE bus for interboard transfer.
DATE: 4/10/96
Modified: Panos Arvanitis
DRAWN BY: Thomas H. Drayer


A
B
C
D
1
2
3
4
XCHECK
IN[7:0]
OUTCHK[7:0]
CHECK_VAL[7:0]
OUT[7:0]
CVAL[7:0]
XILINX
VT SDA LAB
CHECK
This module supplies known data from a port
3/1/96
DRAWN BY: Thomas H. Drayer


A
B
C
D
1
2
3
4
VDD,GND,VDD,GND,GND,VDD,GND,VDD
CHECK_VAL[7:0]
BUF8
INCHK[7:0]
OUTCHK[7:0]
BUF8
VT SDA LAB
Check Port
DATE: 11/15/95
DRAWN BY: Thomas H. Drayer

Appendix C. Device driver source code

Appendix C includes the source code for the PCIDMA and DPIB device drivers. Any include files that are necessary to compile these drivers are also shown here.

PCIDMA.C

```
/*
 *--------------------------------------------------------------
 *
 * Module Name:
 *      pcidma.c
 *
 * Environment:
 *      Kernel mode
 *
 * Version :
 *  1.0
 *
 * Author:
 *      Panos Arvanitis, July 1996
 *
 *
 * Things to do:
 *      Add error logging with EventViewer.
 *
 *  Allow DMA buffer size to be read from registry entry and be
 *  be changeable by the user.
 *
 *  Fix dynamically loading and unloading of the driver (don't
 *  forget to free the DMA buffer and unmap and user memory)
 *
 *  Add the length of the buffer in the information returned by
 *  PPciDmaReturnMemoryInfo, so that the application can
 *  determined if it is adequate.
 *
 *  Portions of this source code were taken from the Skeleton
 *  PCI driver, the author of which could not be determined
 *--------------------------------------------------------------
 */


#include <ntddk.h>
#include <stdarg.h>

//Include header files
#include "ppcidma_ioctl.h"
#include "pcidma_dev.h"

//Static variables
static int nopens;
static int timeout_interval = 100;

//Function Prototypes
static NTSTATUS       CloseDevice(IN PDEVICE_OBJECT devObj,
                                  IN PFILE_OBJECT fileObj);
static NTSTATUS       Dispatch(IN PDEVICE_OBJECT devObj,
                                  IN PIRP Irp);
static NTSTATUS       OpenDevice(IN PDEVICE_OBJECT devObj,
                                  IN PFILE_OBJECT fileObj);
static NTSTATUS       ProbePCI(IN PDRIVER_OBJECT drvObj,
                                  IN PUNICODE_STRING regPath);
static BOOLEAN  ServiceInterrupt(IN PKINTERRUPT Interrupt,
                                  IN PVOID ServiceContext);
static VOID StartIo(IN PDEVICE_OBJECT devObj, IN PIRP Irp);
static VOID Unload(IN PDRIVER_OBJECT);

NTSTATUS PPciDmaMapBuffer(
       IN PSKELETON_DEVICE pLDI,
       IN PIRP pIrp,
```

```
        IN PIO_STACK_LOCATION IrpStack
        );

NTSTATUS PPciDmaUnMapBuffer(
        IN PSKELETON_DEVICE pLDI,
        IN PIRP pIrp,
        IN PIO_STACK_LOCATION IrpStack
        );

NTSTATUS PPciDmaIoctlReadPort(
        IN PSKELETON_DEVICE pLDI,
        IN PIRP pIrp,
        IN PIO_STACK_LOCATION IrpStack,
        IN ULONG IoctlCode
        );

NTSTATUS PPciDmaIoctlWritePort(
        IN PSKELETON_DEVICE pLDI,
        IN PIRP pIrp,
        IN PIO_STACK_LOCATION IrpStack,
        IN ULONG IoctlCode
        );

NTSTATUS PPciDmaReturnMemoryInfo(
        IN PSKELETON_DEVICE pLDI,
        IN PIRP pIrp,
        IN PIO_STACK_LOCATION IrpStack
        );

//Under NT, we can specify the parts of the code that are no
//longer needed after initialization with the pragma alloc_text.
#if 0
#ifdef ALLOC_PRAGMA
#    pragma alloc_text(INIT,DriverEntry)
#endif
#endif


/*
 *--------------------------------------------------------------------
 * DriverEntry --
 *
 *      This routine is called at system initialization time to
 *   initialize this driver.
 *
 * Arguments:
 *      DriverObject    - Supplies the driver object.
 *      RegistryPath    - Supplies the registry path for this
driver.
 *
 * Return Value:
 *      STATUS_SUCCESS          - Could initialize at least one
device.
 *      STATUS_NO_SUCH_DEVICE   - Could not initialize even one
device.
 *  STATUS_UNSUCCESSFUL     - For other errors?
 *--------------------------------------------------------------------
 */
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT drvObj, IN PUNICODE_STRING regPath)
{
  NTSTATUS status;
  KdPrint(("PPCIDMA.SYS : DriverEntry : Entering\n"));

  //Scan PCI bus for PCIDMA board
  status = ProbePCI(drvObj, regPath);
```

```
  if (NT_SUCCESS(status)) {
       //Create dispatch points for NT service routine
       drvObj->MajorFunction[IRP_MJ_CREATE]         = Dispatch;
       drvObj->MajorFunction[IRP_MJ_CLOSE]          = Dispatch;
       drvObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Dispatch;
       drvObj->MajorFunction[IRP_MJ_READ]           = Dispatch;
       drvObj->MajorFunction[IRP_MJ_WRITE]          = Dispatch;
       drvObj->DriverUnload                         = Unload;
       drvObj->DriverStartIo                        = StartIo;
    }

    KdPrint(("PPCIDMA.SYS : DriverEntry : Exiting\n"));

    return status;
}


/*
 *--------------------------------------------------------------------
 * Dispatch --
 *
 *      This routine handles all IRPs sent to this device.
 *
 * Arguments:
 *      devObj:        Pointer to the device object
 *      irp:           Pointer to an I/O request packet
 *
 * Results:
 *      Standard NT result
 *
 *--------------------------------------------------------------------
 */
```

```
static NTSTATUS
Dispatch(IN PDEVICE_OBJECT devObj, IN PIRP irp)
{
    PIO_STACK_LOCATION irpStack;
    PVOID              ioBuf;
    ULONG              inBufLen;
    ULONG              outBufLen;
    ULONG              ioctlCode;
    NTSTATUS           status;
    PSKELETON_DEVICE   skelDev;
    ULONG              key;

    skelDev = devObj->DeviceExtension;

    /*
     * enter device mutex  to ensure one request at a time
     */
    ExAcquireFastMutex(&skelDev->IrpMutex);

    irp->IoStatus.Status      = STATUS_SUCCESS;
    irp->IoStatus.Information = 0;

    irpStack = IoGetCurrentIrpStackLocation(irp);
    //This is the pointer to where the output data from the
apropriate device
    //control routine will be written
    //Only used in map and unmap requests.
    ioBuf = irp->AssociatedIrp.SystemBuffer;

    //Dispatch messages to appropriate routine
    switch (irpStack->MajorFunction) {

      //Create a new PCIDMA device
```

```
      case IRP_MJ_CREATE:
        KdPrint(("PPCIDMA.SYS: IRP_MJ_CREATE\n"));
            status = OpenDevice(devObj, irpStack->FileObject);
        break;

       //Close the PCIDMA device
      case IRP_MJ_CLOSE:
        KdPrint(("PPCIDMA.SYS: IRP_MJ_CLOSE\n"));
            status = CloseDevice(devObj, irpStack->FileObject);
        break;

       //Perform a control function, such as port read or write
      case IRP_MJ_DEVICE_CONTROL:
        ioctlCode= irpStack->Parameters. DeviceIoControl. \
                              IoControlCode;



        switch (ioctlCode) {
            case IOCTL_PPCIDMA_MAP_USER_PHYSICAL_MEMORY:
             status = PPciDmaMapBuffer(skelDev, irp, irpStack);
             break;

            case IOCTL_PPCIDMA_UNMAP_USER_PHYSICAL_MEMORY:
                       status = PPciDmaUnMapBuffer(skelDev,
                                                   irp,
                                                   irpStack
                                                    );
                       break;

            case IOCTL_PPCIDMA_RETURN_MEMORY_INFORMATION:
                       status = PPciDmaReturnMemoryInfo(skelDev,
                                                    irp,
irpStack );
            break;

      case IOCTL_PPCIDMA_READ_PORT_UCHAR:
      case IOCTL_PPCIDMA_READ_PORT_USHORT:
      case IOCTL_PPCIDMA_READ_PORT_ULONG:
            status = PPciDmaIoctlReadPort(
                                   skelDev,
                                   irp,
                                   irpStack,
                                   irpStack-> \
            Parameters.DeviceIoControl.IoControlCode
                                   );
            break;

      case IOCTL_PPCIDMA_WRITE_PORT_UCHAR:
      case IOCTL_PPCIDMA_WRITE_PORT_USHORT:
      case IOCTL_PPCIDMA_WRITE_PORT_ULONG:
            status = PPciDmaIoctlWritePort(
                                   skelDev,
                                   irp,
                                   irpStack,
                                   irpStack-> \
            Parameters.DeviceIoControl.IoControlCode
                                        );
      break;

      default:
        KdPrint(("PPCIDMA.SYS: unknown IRP_MJ_DEVICE_CONTROL\n"));
        status = STATUS_INVALID_PARAMETER;
      break;
      }
            break;
```

```
      default:
            KdPrint(("PPCIDMA.SYS: unknown Major Function\n"));
            status = STATUS_INVALID_PARAMETER;
    }

    /*
     * Don't get cute and try to use the status field of
     * the irp in the return status.  That IRP IS GONE as
     * soon as you call IoCompleteRequest.
     */
    if (status != STATUS_PENDING) {
               irp->IoStatus.Status = status;
               IoCompleteRequest(irp, IO_VIDEO_INCREMENT);
    } else {
               IoMarkIrpPending(irp);
               IoStartPacket(devObj, irp, &key, NULL);
    }

    ExReleaseFastMutex(&skelDev->IrpMutex);
    return status;
}



/*
 *---------------------------------------------------------------------
 * Unload --
 *
 *      Just delete the associated device and return
 *
 * Arguments:
 *      drvObj:        Pointer to the driver object
 *
 * Results:
 *      None
 *---------------------------------------------------------------------
 */
static VOID
Unload(IN PDRIVER_OBJECT drvObj)
{
    WCHAR              devLinkBuf[]  = L"\\DosDevices\\PPCIDMA0";
    UNICODE_STRING     devLinkUniStr;
    WCHAR              devNum;
    PDEVICE_OBJECT     devObj, nextDev;
    PSKELETON_DEVICE   skelDev;
    int                tmp;
    CM_RESOURCE_LIST   EmptyList;
    BOOLEAN            ResourceConflict;

    /*
     * For each device that is on the machine:
     *
     * 1. Delete the symbolic links
     * 2. Turn off the board interrupts and disconnect the
interrupt.
     * 3. Unmap the board memory from system space.
     * 4. Unreport the resources that were assigned by
HalAssignSlotResources
     * 5. Delete the device object
     */

    for (devNum = 0, devObj = drvObj->DeviceObject; devObj !=
NULL;
         devObj = nextDev, devNum++) {
        devLinkBuf[sizeof(devLinkBuf) - 1] = L'0' + devNum;
```

```
        RtlInitUnicodeString(&devLinkUniStr, devLinkBuf);
        IoDeleteSymbolicLink(&devLinkUniStr);

        skelDev = devObj->DeviceExtension;
        IoDisconnectInterrupt(skelDev->KIntrObj);
        //MmUnmapIoSpace(skelDev->FrameBase, skelDev->MemLength);
        /*HalFreeCommonBuffer(skelDev->AdaptorObj,
PPCIDMA_MAX_DMA_BUFFER_LENGTH, skelDev->LogicalAddress,
                                                    &skelDev-
>VirtualAddress, FALSE);*/

        /* un-report any resources used by the driver and the
device */
        EmptyList.Count = 0;
        IoReportResourceUsage(NULL, drvObj, &EmptyList,
sizeof(ULONG),
                              drvObj->DeviceObject, &EmptyList,
sizeof(ULONG),
                              FALSE, &ResourceConflict);

        nextDev = devObj->NextDevice;
        IoDeleteDevice(devObj);
    }
    KdPrint(("PPCIDMA.SYS: unloading\n"));
}
```

```
/*
 *--------------------------------------------------------------
 * StartTransferTimeout --
 *
 *      Starts a timer that can check on the DMA operation.
Hopefully,
 *      it never goes off
 *
 * Results:
 *      None
 *--------------------------------------------------------------
 */
static void
StartTransferTimeout(IN PDEVICE_OBJECT devObj, int msTimeout,
PVOID Ignore)
{
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;

        KdPrint(("PPCIDMA.SYS : UhOh : StartTransferTimeOut
called.\n"));
    skelDev->TransferDone = FALSE;
    /*
     * Timer is in 100 ns units.
     */
    KeSetTimer(&skelDev->DeviceCheckTimer,
               RtlConvertLongToLargeInteger(-msTimeout * 10000),
&skelDev->TimerDpc);
    skelDev->TimerStarted = TRUE;
}
```

```
/*
 *----------------------------------------------------------------
 * CancelTransferTimeout --
 *
 *      Remove a previously set DMA timeout timer.
 *
 * Results:
 *      None
 *----------------------------------------------------------------
 */
static void
CancelTransferTimeout(PVOID Context)
{
    PDEVICE_OBJECT devObj = Context;
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;

        KdPrint(("PPCIDMA.SYS : UhOh : CancelTransferTimeOut
called.\n"));

    skelDev->TransferDone = TRUE;
}



/*
 *----------------------------------------------------------------
 * ProgramDMAUtil
 *
 *      Utility routine that starts the DMA transfer
 *
 * Results:
 *      TRUE
 *----------------------------------------------------------------
 */
static BOOLEAN
ProgramDMAUtil(IN PVOID Context)
{
    PDEVICE_OBJECT devObj = Context;
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;
    ULONG bufLen;
    BOOLEAN writeOp;
    ULONG toMapLen, mappedLen;
    PHYSICAL_ADDRESS physAddr;
    PVOID virtualAddr;
    PIRP irp;

        KdPrint(("PPCIDMA.SYS : UhOh : ProgramDmaUtil
called.\n"));

    irp = (PIRP) skelDev->syncParam1;
    bufLen = skelDev->IrpBufLen;
    virtualAddr = skelDev->VirtualAddress;

    writeOp = (skelDev->OperationType == IRP_MJ_READ) ? FALSE :
TRUE;

    toMapLen = bufLen;
    while (toMapLen > 0) {
        mappedLen = (toMapLen >= 4096) ? 4096 : toMapLen;
        physAddr = IoMapTransfer(NULL, irp->MdlAddress,
                                 skelDev->MapRegisterBase,
                                 virtualAddr, &mappedLen, writeOp);

        /*
         * XXX: set addr on the board.  This will be different
per board.
```

```
         * Maybe you don't even have to do anything.
         */
#if 0
        board_write(start_address) =
LITTLE_ENDIAN_32(physAddr.u.LowPart);
#endif
        toMapLen -= mappedLen;
        virtualAddr = (((char *) virtualAddr) + mappedLen);
    }
    return TRUE;
}



/*
 *----------------------------------------------------------------
 * ProgramDMA
 *
 *      This routine gets called back by NT when an adapter
channel
 *      is free to use.  It then uses ProgramDMAUtil to start the
 *      actual transfer
 *
 * Results:
 *
 *----------------------------------------------------------------
 */
static IO_ALLOCATION_ACTION
ProgramDMA(IN PDEVICE_OBJECT devObj, IN PIRP irp,
           IN PVOID MapRegisterBase, IN PVOID Context)
{
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;

        KdPrint(("PPCIDMA.SYS : UhOh : ProgramDma called.\n"));

    skelDev->MapRegisterBase = MapRegisterBase;
    skelDev->syncParam1 = (ULONG) irp;
    KeSynchronizeExecution(skelDev->KIntrObj, ProgramDMAUtil,
devObj);
    StartTransferTimeout(devObj, timeout_interval, NULL);

    /*
     * return a value that says we want to keep the map
registers.
     */
    return DeallocateObjectKeepRegisters;
}



/*
 *----------------------------------------------------------------
 * StartIo --
 *
 *      This gets called when we are about to start a DMA
operation.
 *      This can occur because another DMA operation just
completed,
 *      or it can occur because this is the first DMA operation.
Either
 *      way, we don't expect anything to interfere with its
operation.
 *
 * Results:
 *      None
 *----------------------------------------------------------------
 */
```

```
static VOID
StartIo(IN PDEVICE_OBJECT devObj, IN PIRP irp)
{
    PIO_STACK_LOCATION  irpStack;
    PVOID               ioBuf;
    ULONG               inBufLen;
    ULONG               outBufLen;
    ULONG               ioctlCode;
    NTSTATUS            status;
    BOOLEAN             writeOp;
    PSKELETON_DEVICE    skelDev;

    skelDev = devObj->DeviceExtension;
    irpStack = IoGetCurrentIrpStackLocation(irp);

    KdPrint(("PPCIDMA.SYS(StartIo): Beginning irp %p\n", irp));

    switch (irpStack->MajorFunction) {
      case IRP_MJ_READ:
      case IRP_MJ_WRITE:
        break;

      case IRP_MJ_DEVICE_CONTROL:
        ioBuf     = irp->AssociatedIrp.SystemBuffer;
        inBufLen  = irpStack-
>Parameters.DeviceIoControl.InputBufferLength;
        outBufLen = irpStack-
>Parameters.DeviceIoControl.OutputBufferLength;
        ioctlCode = irpStack-
>Parameters.DeviceIoControl.IoControlCode;
         switch (ioctlCode) {
          default:
            KdPrint(("PPCIDMA.SYS(StartIo): unexpected
IRP_MJ_DEVICE_CONTROL\n"));
            status = STATUS_INVALID_PARAMETER;
            break;
         }

        if (status != STATUS_PENDING) {
            irp->IoStatus.Status = status;
            IoCompleteRequest(irp, IO_VIDEO_INCREMENT);
            IoStartNextPacket(devObj, TRUE);
        }
        return;

      default:
        KdPrint(("PPCIDMA.SYS(StartIo): unexpected major
function\n"));
        irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        IoCompleteRequest(irp, IO_NO_INCREMENT);
        IoStartNextPacket(devObj, TRUE);
        return;
    }

    skelDev->OperationType = irpStack->MajorFunction;
    skelDev->IrpSystemBuffer = irp->AssociatedIrp.SystemBuffer;
    if (skelDev->OperationType == IRP_MJ_READ) {
        skelDev->IrpBufLen = irpStack->Parameters.Read.Length;
    } else {
        skelDev->IrpBufLen  = irpStack->Parameters.Write.Length;
    }
    if (skelDev->IrpBufLen == 0 || irp->MdlAddress == NULL) {
        irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        IoCompleteRequest(irp, IO_NO_INCREMENT);
        IoStartNextPacket(devObj, TRUE);
```

```
    }

    skelDev->VirtualAddress = MmGetMdlVirtualAddress(irp-
>MdlAddress);

    if (skelDev->TimerStarted) {
        KeCancelTimer(&skelDev->DeviceCheckTimer);
        skelDev->TimerStarted = FALSE;
    }

    writeOp = (skelDev->OperationType == IRP_MJ_READ) ? FALSE :
TRUE;

    KeFlushIoBuffers(irp->mdlAddress, !writeOp, TRUE);
    status = IoAllocateAdapterChannel(skelDev->AdaptorObj,
devObj,
                                      skelDev->DmaMapRegisters,
ProgramDMA, devObj);

    KdPrint(("PPCIDMA.SYS(StartIo): Exiting irp %p\n", irp));
    if (!NT_SUCCESS(status)) {
        KdPrint(("PPCIDMA.SYS: Unable to allocate adaptor channel
for DMA\n"));
        irp->IoStatus.Status = status;
        IoCompleteRequest(irp, IO_NO_INCREMENT);
        return;
    }
}
```

```
/*
 *--------------------------------------------------------------
 * TransferDPC --
 *
 *      This routine is called at DISPATCH_LEVEL by the system at
the
 *      ServiceInterrupt().
 *
 *      This routine is protected against interrupts since it was
queued
 *      by an interrupt, and the next DMA related interrupt won't
occur
 *      until something else happens.
 *
 *      This routine is called when a DMA transfer has not been
completed.
 *      It sets everything up to continue the tranfer.
 *--------------------------------------------------------------
 */
static VOID
TransferDPC(IN PKDPC Dpc, IN PVOID Context, IN PVOID Arg1, IN
PVOID Arg2)
{
    PDEVICE_OBJECT devObj = Context;
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;
    PIRP irp;
    BOOLEAN writeOp;

    KdPrint(("PPCIDMA.SYS(TransferDPC): Finished irp %p\n",
             devObj->CurrentIrp));
    CancelTransferTimeout(devObj);

    irp = devObj->CurrentIrp;
```

```
    writeOp = (skelDev->OperationType == IRP_MJ_WRITE) ? TRUE :
FALSE;

    IoFlushAdapterBuffers(NULL, irp->MdlAddress,
                          skelDev->MapRegisterBase,
                          skelDev->VirtualAddress, skelDev-
>IrpBufLen, writeOp);

    IoFreeMapRegisters(skelDev->AdaptorObj, skelDev-
>MapRegisterBase,
                       skelDev->DmaMapRegisters);

    if (skelDev->OperationType == IRP_MJ_READ) {
        KeFlushIoBuffers(irp->MdlAddress, TRUE, TRUE);
    }
    irp->IoStatus.Status = skelDev->IrpStatus;
    if (skelDev->IrpStatus == STATUS_SUCCESS) {
        irp->IoStatus.Information = skelDev->IrpBytesTransferred;
    }
    IoCompleteRequest(irp, IO_VIDEO_INCREMENT);
    IoStartNextPacket(devObj, TRUE);
    skelDev->DpcRequested = FALSE;

    return;
}



/*
 *----------------------------------------------------------------
 * ServiceTimeoutUtil --
 *
 *      Utility routine for ServiceTimeout.  Runs code that is
 *      sensitive to interrupts.
 *----------------------------------------------------------------
 */
static BOOLEAN
ServiceTimeoutUtil(IN PCONTEXT Context)
{
    PSKELETON_DEVICE skelDev = (PSKELETON_DEVICE) Context;

        KdPrint(("PPCIDMA.SYS : UhOh : ServiceTimeoutUtil
called.\n"));

    return TRUE;
}



/*
 *----------------------------------------------------------------
 * ServiceTimeout --
 *
 *      Service a timeout.  Is this a routine to check on the
board
 *      if nothing happens after a little while?  If so,
 *      ddk/src/multimedia/soundlib/wave.c does something
similar.
 *
 * Results:
 *      None
 *----------------------------------------------------------------
 */
void
ServiceTimeout(PDEVICE_OBJECT devObj)
{
```

```
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;

        KdPrint(("PPCIDMA.SYS : UhOh : ServiceTimeout
called.\n"));

    KeSynchronizeExecution(skelDev->KIntrObj, ServiceTimeoutUtil,
skelDev);

    skelDev->IrpStatus = STATUS_UNSUCCESSFUL;
    skelDev->IrpBytesTransferred = 0L;
    skelDev->RequestDpc = TRUE;

    KdPrint(("PPCIDMA.SYS: ServiceTimeout calling
TransferDPC\n"));
    TransferDPC(NULL, devObj, NULL, NULL);
}



/*
 *---------------------------------------------------------------
 * TimeoutDPC --
 *
 *      This routine gets called when a timeout occurs.  We then
 *      need to check plxDev->TransferDone to see if the transfer
 *      finished before this timer went off.  If is did, then we
 *      can just ignore this DPC call.  If not, we need to clear
 *      everything up, fail the request, and move on.
 *
 * Results:
 *      None
 *---------------------------------------------------------------
 */
static VOID
TimeoutDPC(IN PKDPC Dpc, IN PVOID Context, IN PVOID Param1, IN
PVOID Param2)
{
    PDEVICE_OBJECT devObj = Context;
    PSKELETON_DEVICE skelDev = devObj->DeviceExtension;

        KdPrint(("PPCIDMA.SYS : UhOh : TimeOutDPC called.\n"));

    skelDev->TimerStarted = FALSE;
    if (! skelDev->TransferDone) {
        /*
         * XXX: Clean up the hardware here if necessary.
         */
        ServiceTimeout(devObj);
    }
}



/*
 *---------------------------------------------------------------
 * OpenDevice --
 *
 *      Open the device.  We will allow multiple opens to the
device.
 *
 * Results:
 *      A standard NT result
 *---------------------------------------------------------------
 */
static NTSTATUS
OpenDevice(IN PDEVICE_OBJECT devObj, IN PFILE_OBJECT fileObj)
```

```
{
    PSKELETON_DEVICE skelDev;

    KdPrint(("PPCIDMA.SYS: OpenDevice called\n"));

    skelDev = devObj->DeviceExtension;
    ++nopens;                 /* inc global open */
    return STATUS_SUCCESS;
}



/*
 *------------------------------------------------------------------
 * CloseDevice --
 *
 *     Close up device and free resources allocated by
OpenDevice
 *
 * Results:
 *     A standard NT result
 *------------------------------------------------------------------
 */
static NTSTATUS
CloseDevice(IN PDEVICE_OBJECT devObj, IN PFILE_OBJECT fileObj)
{
    PSKELETON_DEVICE skelDev;

    skelDev = devObj->DeviceExtension;
    nopens--;                 /* decrement global open */
    return STATUS_SUCCESS;
}
```

```
/*********************************************************
* PPciDmaUnmapBuffer
*
*/
NTSTATUS PPciDmaUnMapBuffer(
                      IN PSKELETON_DEVICE pLDI,
                      IN PIRP pIrp,
                      IN PIO_STACK_LOCATION IrpStack
                      )
{
        PULONG  pIOBuffer;
        ULONG   InBufferSize;

        InBufferSize = IrpStack-
>Parameters.DeviceIoControl.InputBufferLength;
        pIOBuffer = (PULONG) pIrp->AssociatedIrp.SystemBuffer;

        if (InBufferSize < sizeof(PVOID)) {
                KdPrint(("PPCIDMA.SYS : UnMapBuffer : Buffer too
small.\n"));
                return(STATUS_BUFFER_TOO_SMALL);
        }

        KdPrint(("PPCIDMA.SYS : Trying to unmap at %x.\n",
*pIOBuffer));
        return ZwUnmapViewOfSection((HANDLE) -1, *((PVOID *)
pIOBuffer));
}
```

```
/*
 *--------------------------------------------------------------
 * ResetBoard --
 *
 *     Does a hard reset of the board
 *
 *--------------------------------------------------------------
 */
static VOID
ResetBoard(PSKELETON_DEVICE skelDev)
{
    PUCHAR                          base;

    /*
     * Reset the board
     */

        KdPrint(("PPCIDMA.SYS : UhOh : ResetBoard called.\n"));

    base = skelDev->FrameBase;
    *((unsigned long *)(base + 0x7f0000)) = 0;

    KeStallExecutionProcessor(500);

    /*
     * Enable the board
     */
    *((unsigned long *)(base + 0x7f0000)) = LITTLE_ENDIAN_32(1);
}
```

```
/*
 *--------------------------------------------------------------
 * CreateDevice --
 *
 *     Create a Skeleton device
 *
 *--------------------------------------------------------------
 */
static NTSTATUS
CreateDevice(IN PDRIVER_OBJECT drvObj, IN PUNICODE_STRING
regPath,
            ULONG busId, ULONG slotId, IN PPCI_COMMON_CONFIG
pciData)
{
    PDEVICE_OBJECT              devObj = NULL;
    WCHAR                       devNameBuf[] =
L"\\Device\\PPCIDMA0";
    UNICODE_STRING              devNameUniStr;
    WCHAR                       devLinkBuf[] =
L"\\DosDevices\\PPCIDMA0";
    UNICODE_STRING              devLinkUniStr;
    NTSTATUS                    status;
    INTERFACE_TYPE              interfaceType;
    ULONG                       busNumber;
    PCM_RESOURCE_LIST           resourceList;
    PCM_PARTIAL_RESOURCE_LIST   partialResourceList;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR partialDescriptor;
    PHYSICAL_ADDRESS            xlatedAddr;
    PHYSICAL_ADDRESS            start;
    ULONG                       length;
    ULONG                       addressSpace;
    PSKELETON_DEVICE            skelDev;
    ULONG                       i, j;
```

```
    BOOLEAN                     b;
    int                         found;
    DEVICE_DESCRIPTION          devDesc;
    static WCHAR                devNum = 0;


    devNameBuf[sizeof(devNameBuf) - 2] = L'0' + devNum;
    devLinkBuf[sizeof(devLinkBuf) - 2] = L'0' + devNum;
    devNum++;

    RtlInitUnicodeString(&devNameUniStr, devNameBuf);

    status = IoCreateDevice(drvObj, sizeof(SKELETON_DEVICE),
&devNameUniStr,
                        FILE_DEVICE_SKELETON, 0, FALSE,
&devObj);

    devObj->Flags |= DO_DIRECT_IO;

    /*
     * 1. Create dispatch points for device control, create,
close.
     * 2. Create a symbolic link, i.e. a name that a Win32 app
can
     *    specify to open the device.  If this fails, delete the
     *    device object.
     */
    if (! NT_SUCCESS(status)) {
        KdPrint(("PPCIDMA.SYS: IoCreateDevice failed\n"));
       return status;
    }

    RtlInitUnicodeString(&devLinkUniStr, devLinkBuf);
    status = IoCreateSymbolicLink(&devLinkUniStr,
&devNameUniStr);

    if (!NT_SUCCESS(status)) {
        KdPrint(("PPCIDMA.SYS: IoCreateSymbolicLink failed\n"));
        goto error;
    }

       KdPrint(("PPCIDMA.SYS : Entering
HalAssignSlotResources.\n"));

    resourceList = NULL;
    status = HalAssignSlotResources(regPath, NULL, drvObj,
devObj,
                                    PCIBus, busId, slotId,
                                    &resourceList);
    if (!NT_SUCCESS(status)) {
        KdPrint(("PPCIDMA.SYS: HalAssignSlotResources
failed\n"));
        goto error;
    }

       KdPrint(("PPCIDMA.SYS : HalAssignSlotResources was
successful.\n"));

    /*
     * Now, we hopefully have an address for the card on the bus,
     * but who knows for sure.  We need to translate the returned
     * address and map the address space into kernel space.
     */

    skelDev = devObj->DeviceExtension;
    skelDev->BusId = busId;
```

```
    skelDev->SlotId = slotId;

    KdPrint(("PPCIDMA.SYS : resourceList->Count = %x\n",
resourceList->Count));

    found = 0;
    for (i = 0; i < resourceList->Count; i++) {
         //Get the interface type (PCIBus) from the resource
list
         interfaceType = resourceList->List[i].InterfaceType;
         //Get the bus number from the resource list
         busNumber = resourceList->List[i].BusNumber;
         //Get the ith partial resource list
         partialResourceList = &resourceList-
>List[i].PartialResourceList;

         KdPrint(("PPCIDMA.SYS : partialResourceList->Count =
%x\n",
                 partialResourceList->Count));

         //Go through each of the partial resource lists
         for (j = 0; j < partialResourceList->Count; j++) {
              //Get the jth partial resource descriptor
           partialDescriptor = &partialResourceList-
>PartialDescriptors[j];

              //Tell me what type of resource this is
              KdPrint(("PPCIDMA.SYS : partialDescriptor->Type =
"));
              switch (partialDescriptor->Type)
              {
              case CmResourceTypePort                    :
       KdPrint(("Port.\n"));
                    break;
              case CmResourceTypeInterrupt  :
       KdPrint(("Interrupt.\n"));

                    break;
              case CmResourceTypeMemory            :
       KdPrint(("Memory.\n"));

                    break;
              case CmResourceTypeDma               :
       KdPrint(("DMA.\n"));

                    break;
              case CmResourceTypeDeviceSpecific    :
       KdPrint(("Device specific.\n"));

                           break;
              }

/*
           if (partialDescriptor->Type == CmResourceTypeMemory)
{

                KdPrint(("PPCIDMA.SYS : partialDescriptorType =
CmResourceMemory\n"));

                addressSpace = 0; // Memory
                start = partialDescriptor->u.Memory.Start;
                length = partialDescriptor->u.Memory.Length;

                //Translate bus address to system logical
address
```

```
                b = HalTranslateBusAddress(interfaceType,
busNumber,
                                           start,
                                           &skelDev->FrameMemType,
                                           &xlatedAddr);

                if (!b) {
                  KdPrint(("PPCIDMA.SYS: HalTranslateBusAddress
failed\n"));
                  status = STATUS_UNSUCCESSFUL;
                  goto error;
                }

                skelDev->MemStart = xlatedAddr;
                skelDev->MemLength = length;

                if (skelDev->FrameMemType == 0) {
                  skelDev->FrameBase = MmMapIoSpace(xlatedAddr,
length, FALSE);
                } //else {
                  //skelDev->FrameBase = (PUCHAR)
xlatedAddr.LowPart;
                  //}

              found++;

           } else*/ if (partialDescriptor->Type ==
CmResourceTypeInterrupt) {

              /*
              * Get the system interrupt vector for
IoConnectInterrupt
              */
              ULONG level    = partialDescriptor-
>u.Interrupt.Level;
              ULONG vector   = partialDescriptor-
>u.Interrupt.Vector;
              ULONG affinity = partialDescriptor-
>u.Interrupt.Affinity;

              KdPrint(("PPCIDMA.SYS : partialDescriptorType =
CmResourceTypeInterrupt\n"));

              //Get mapped system interrupt vector for
IoConnectInterrupt
              skelDev->KIntrVector =
                  HalGetInterruptVector(PCIBus, busId, level,
vector,
                                        &skelDev->KIrql,
                                        &skelDev->KIntrAffinity);

              devDesc.Version = DEVICE_DESCRIPTION_VERSION;
              devDesc.Master = TRUE;
              devDesc.ScatterGather = FALSE;
              devDesc.DemandMode = FALSE;
              devDesc.AutoInitialize = FALSE;
              devDesc.Dma32BitAddresses = TRUE;
              devDesc.IgnoreCount = FALSE;
              devDesc.Reserved1 = FALSE;
              devDesc.Reserved2 = FALSE;
              devDesc.BusNumber = busId;
              devDesc.DmaChannel = 0; /* ? */
              devDesc.InterfaceType = PCIBus;
              devDesc.DmaWidth = Width32Bits;
              devDesc.DmaSpeed = Compatible;
```

```
            devDesc.MaximumLength =
PPCIDMA_MAX_DMA_BUFFER_LENGTH;
            devDesc.DmaPort = 0;

            //You only get one interrupt vector, so it's OK to
put this here.
            //Get pointer to DMA adapter object with given
description
            skelDev->AdaptorObj =
                HalGetAdapter(&devDesc, &skelDev-
>DmaMapRegisters);

            if (skelDev->AdaptorObj == NULL)
                   KdPrint(("PPCIDMA.SYS : HalGetAdapter
failed.\n"));

            //Allocate common buffer for DMA.
            //This must be done during initialization to have
a chance at getting
            //a large buffer.
            KdPrint(("PPCIDMA.SYS : Will attempt to allocate
buffer size %d bytes.\n",
                   devDesc.MaximumLength));



            /* This stuff can probably go away for a PCI card
with no 24-bit address limitation
            //Failed to get enough DMA map registers, i.e.
enough memory for the buffer
            if (skelDev->DmaMapRegisters*4096 <
devDesc.MaximumLength)
            {
                   KdPrint(("PPCIDMA.SYS : Not enough DMA map
registers available.\n"));
                   status = STATUS_INSUFFICIENT_RESOURCES;
                   goto error;
            }
            */



            //Allocate the common DMA buffer
            //I don't know what the virtual address returned
is used for, other than MDLs
            //and map registers, so I believe we can ignore
it.
            skelDev->VirtualAddress =
                   HalAllocateCommonBuffer(skelDev-
>AdaptorObj, devDesc.MaximumLength,
                                                 &skelDev-
>LogicalAddress, FALSE);

            skelDev->BufferLength = devDesc.MaximumLength;



            //Failed to allocate contiguous memory for DMA
buffer
            if (skelDev->VirtualAddress == NULL)
            {
                   KdPrint(("PPCIDMA.SYS :
HalAllocateCommonBuffer failed.\n"));
                   status = STATUS_INSUFFICIENT_RESOURCES;
                   goto error;
            }
```

```
            KdPrint(("PPCIDMA.SYS : Common buffer allocated at
virtual address %x.\n", skelDev->VirtualAddress));
            KdPrint(("PPCIDMA.SYS : Common buffer allocated at
physical (logical) address %x.\n", skelDev->LogicalAddress));

            /* This stuff can probably go away for PCI
            skelDev->Pmdl = IoAllocateMdl(skelDev-
>VirtualAddress, devDesc.MaximumLength,
                                                  FALSE,
FALSE, 0);

            if (skelDev->Pmdl == NULL)
            {
                  KdPrint(("PPCIDMA.SYS : IoAllocateMdl
failed.\n"));
                  status = STATUS_INSUFFICIENT_RESOURCES;
                  goto error;
            }


            MmBuildMdlForNonPagedPool(skelDev->Pmdl);
            */

            found++;

            } else if (partialDescriptor->Type ==
CmResourceTypePort) {

                  skelDev->FrameMemType = 1;    //Use I/O
space
                  start  = partialDescriptor->u.Port.Start;
                  length = partialDescriptor->u.Port.Length;

                  KdPrint(("PPCIDMA.SYS : Entering
HalTranslateBusAddress with start = %x\n", start));

                  b = HalTranslateBusAddress(interfaceType,
busNumber,
                                             start,
                                             &skelDev-
>FrameMemType,
                                             &xlatedAddr);

                  KdPrint(("PPCIDMA.SYS : Returned from
HalTranslateBusAddress.\n"));

                  if (!b) {
                        KdPrint(("PPCIDMA.SYS:
HalTranslateBusAddress failed\n"));
                        status = STATUS_UNSUCCESSFUL;
                        goto error;
                  }

               skelDev->FrameBase = xlatedAddr.LowPart;
                  skelDev->PortCount = length;
                  KdPrint(("PPCIDMA.SYS : FrameBase (xlated)
= %x\n", skelDev->FrameBase));

                  found++;

            } //else if
          } //for j
    } //for i

    if (found < 2) {
```

```
        KdPrint(("PPCIDMA.SYS: Failed to find frame buffer
address or interrupt\n"));
        status = STATUS_UNSUCCESSFUL;
        goto error;
    }

    /*
     * Enable I/O Space and Bus Master control bits
     */
    pciData->Command = 5;
    HalSetBusDataByOffset(PCIConfiguration, busId, slotId,
                          &pciData->Command,
                          offsetof(PCI_COMMON_CONFIG, Command),
                          sizeof(pciData->Command));



    //ResetBoard(skelDev);

    /*
     * 1. Initialize the device mutex
     * 2. Initialize the device spin lock to protect the DPC
routine
     *    for callers to SynchronizeDPC.
     * 3. Initialize the DPC data and register with Io system
     * 4. Connect the interrupt
     */
    ExInitializeFastMutex(&skelDev->IrpMutex);
    KeInitializeSpinLock(&skelDev->DeviceSpinLock);
    KeInitializeTimer(&skelDev->DeviceCheckTimer);
    KeInitializeTimer(&skelDev->StartIoTimer);
    skelDev->TimerStarted = FALSE;
    KeInitializeDpc(&skelDev->TimerDpc, TimeoutDPC, devObj);
    skelDev->DpcRequested = FALSE;
    IoInitializeDpcRequest(devObj, TransferDPC);

    status = IoConnectInterrupt(&skelDev->KIntrObj,
ServiceInterrupt,
                                devObj, NULL,
                                skelDev->KIntrVector,
                                skelDev->KIrql, skelDev->KIrql,
                                LevelSensitive,
                                TRUE, /* ShareVector */
                                skelDev->KIntrAffinity, FALSE);

    if (!NT_SUCCESS(status)) {
        KdPrint(("PPCIDMA.SYS: Unable to connect interrupt\n"));
        status = STATUS_UNSUCCESSFUL;
        goto error;
    }

    if (0) {
  error:
        IoDeleteDevice(devObj);
    } else {
        ExFreePool(resourceList);
    }
    return status;
}
```

```
/*
 *--------------------------------------------------------------
 *
 * ProbePCI
 *
 *      Attempt to find all Skeleton adapters in PCI address
space
 *
 * Return Value:
 *      STATUS_SUCCESSFUL if everything went OK,
STATUS_UNSUCCESSFUL
 *      if not.
 *
 *--------------------------------------------------------------
 */
static NTSTATUS
ProbePCI(IN PDRIVER_OBJECT drvObj, IN PUNICODE_STRING regPath)
{
    PCI_SLOT_NUMBER     slotNumber;
    PPCI_COMMON_CONFIG  pciData;
    UCHAR               buf[PCI_COMMON_HDR_LENGTH];
    ULONG               i, f, j, bus;
    BOOLEAN             flag;
    UCHAR               vendorString[5] = {0};
    UCHAR               deviceString[5] = {0};
    NTSTATUS           status;
    ULONG              total = 0;

    pciData = (PPCI_COMMON_CONFIG) buf;
    slotNumber.u.bits.Reserved = 0;

    flag = TRUE;
    for (bus = 0; flag; bus++) {

        for (i = 0; i < PCI_MAX_DEVICES  &&  flag; i++) {
            slotNumber.u.bits.DeviceNumber = i;

            for (f = 0; f < PCI_MAX_FUNCTION; f++) {
                slotNumber.u.bits.FunctionNumber = f;

                j = HalGetBusData(PCIConfiguration, bus,
slotNumber.u.AsULONG,
                    pciData, PCI_COMMON_HDR_LENGTH);

                if (j == 0) {
                    /* out of buses */
                    flag = FALSE;
                    break;
                }

                if (pciData->VendorID == PCI_INVALID_VENDORID) {
                    /* skip to next slot */
                    break;
                }


                KdPrint(("PciData: ------------------------\n"
                        "  Bus: %d\n"
                        "  Device: %d\n"
                        "  Function: %d\n"
                        "  Vendor Id: %x\n"
                        "  Device Id: %x\n"
                        "  Command: %x\n"
                        "  Status: %x\n"
                        "  Rev Id: %x\n"
                        "  Pro`gIf: %x\n"
```

```
"  SubClass: %x\n"
"  BaseClass: %x\n"
"  CacheLine: %x\n"
"  Latency: %x\n"
"  Header Type: %x\n"
"  BIST: %x\n"
"  Base Reg[0]: %x\n"
"  Base Reg[1]: %x\n"
"  Base Reg[2]: %x\n"
"  Base Reg[3]: %x\n"
"  Base Reg[4]: %x\n"
"  Base Reg[5]: %x\n"
"  Rom Base: %x\n"
"  Interrupt Line: %x\n"
"  Interrupt Pin: %x\n"
"  Min Grant: %x\n"
"  Max Latency: %x\n",
bus,
i,
f,
pciData->VendorID,
pciData->DeviceID,
pciData->Command,
pciData->Status,
pciData->RevisionID,
pciData->ProgIf,
pciData->SubClass,
pciData->BaseClass,
pciData->CacheLineSize,
pciData->LatencyTimer,
pciData->HeaderType,
pciData->BIST,
pciData->u.type0.BaseAddresses[0],
                        pciData->u.type0.BaseAddresses[1],
                        pciData->u.type0.BaseAddresses[2],
                        pciData->u.type0.BaseAddresses[3],
                        pciData->u.type0.BaseAddresses[4],
                        pciData->u.type0.BaseAddresses[5],
                        pciData->u.type0.ROMBaseAddress,
                        pciData->u.type0.InterruptLine,
                        pciData->u.type0.MinimumGrant,
                        pciData->u.type0.MaximumLatency));


                /*
                 * If we find the Skeleton id, create a device
                 */
                if (pciData->VendorID == PCIDMA_VENDORID &&
                                    pciData->DeviceID ==
PCIDMA_DEVICEID)
                {
                    status = CreateDevice(drvObj, regPath, bus,
                                          slotNumber.u.AsULONG,
pciData);
                    if (NT_SUCCESS(status)) {
                        total++;
                    }
                 }
             }
         }
    }


    if (total > 0) {
        return STATUS_SUCCESS;
    } else {
```

```
        return STATUS_NO_SUCH_DEVICE;
    }
}



/*
 *----------------------------------------------------------------
 * ServiceInterrupt --
 *
 *      Service an interrupt from the Skeleton board
 *
 * Results:
 *      TRUE if the interrupt was handled, FALSE otherwise.
 *----------------------------------------------------------------
 */
static BOOLEAN
ServiceInterrupt(IN PKINTERRUPT Interrupt, IN PVOID
ServiceContext)
{

        //The following lines will not compile, since there
really is no
        //interrupt service routine in this version of the device
driver.
#if 0
    PDEVICE_OBJECT devObj = (PDEVICE_OBJECT) ServiceContext;
    PSKELETON_DEVICE skelDev;

    skelDev = devObj->DeviceExtension;

        KdPrint(("PPCIDMA.SYS : UhOh : ServiceInterrupt
called.\n"));

    /*
     * XXX: Check if this interrupt was really intended for your
board.
     * If not, return FALSE;
     */
    if (skelDev->RequestDpc) {
        if (!skelDev->DpcRequested) {
            skelDev->DpcRequested = TRUE;
            IoRequestDpc(devObj, NULL, devObj);
        } else {
            KdPrint(("PPCIDMA.SYS: dpc overrun\n"));
        }
    }

    /*
     * Change this to TRUE when this routine does something
     */
    return TRUE;
#endif
        return FALSE;
}



/*********************************************************
* PPciDmaIoctlreadPort
*   Handle read port IOCTLs
*********************************************************/
NTSTATUS
PPciDmaIoctlReadPort(
    IN PSKELETON_DEVICE pLDI,
    IN PIRP pIrp,
    IN PIO_STACK_LOCATION IrpStack,
```

```
        IN ULONG IoctlCode
        )
/*++

Routine Description:
    This routine processes the IOCTLs which read from the ports.

Arguments:

    pLDI        - our local device data
    pIrp        - IO request packet
    IrpStack    - The current stack location
    IoctlCode   - The ioctl code from the IRP

Return Value:
    STATUS_SUCCESS           -- OK

    STATUS_INVALID_PARAMETER -- The buffer sent to the driver
                                was too small to contain the
                                port, or the buffer which
                                would be sent back to the driver
                                was not a multiple of the data
size.

    STATUS_ACCESS_VIOLATION  -- An illegal port number was given.

--*/

{
                                // NOTE:  Use METHOD_BUFFERED
ioctls.
    PULONG pIOBuffer;           // Pointer to transfer buffer
                                //      (treated as an array of
longs).
    ULONG InBufferSize;         // Amount of data avail. from
caller.
    ULONG OutBufferSize;        // Max data that caller can
accept.
    ULONG nPort;                // Port number to read
    ULONG DataBufferSize;

    // Size of buffer containing data from application
    InBufferSize  = IrpStack-
>Parameters.DeviceIoControl.InputBufferLength;
       //KdPrint(("PPCIDMA.SYS : ReadPort : InBufferSize =
%x\n", InBufferSize));

    // Size of buffer for data to be sent to application
    OutBufferSize = IrpStack-
>Parameters.DeviceIoControl.OutputBufferLength;
       //KdPrint(("PPCIDMA.SYS : ReadPort : OutBufferSize =
%x\n", OutBufferSize));

    // NT copies inbuf here before entry and copies this to
outbuf after
    // return, for METHOD_BUFFERED IOCTL's.
    pIOBuffer     = (PULONG)pIrp->AssociatedIrp.SystemBuffer;

    // Check to ensure input buffer is big enough to hold a port
number and
    // the output buffer is at least as big as the port data
width.
    //
    switch (IoctlCode)
    {
```

```
    //default:                    // There isn't really any
default but
       /* FALL THRU */         // this will quiet the compiler.
    case IOCTL_PPCIDMA_READ_PORT_UCHAR:
        DataBufferSize = sizeof(UCHAR);
        break;
    case IOCTL_PPCIDMA_READ_PORT_USHORT:
        DataBufferSize = sizeof(USHORT);
        break;
    case IOCTL_PPCIDMA_READ_PORT_ULONG:
        DataBufferSize = sizeof(ULONG);
        break;
       default:
              KdPrint(("PPCIDMA.SYS : ReadPort : CAUTION -
default entered on switch (IoctlCode).\n"));
    }

    if ( InBufferSize != sizeof(ULONG) || OutBufferSize <
DataBufferSize )
    {
        return STATUS_INVALID_PARAMETER;
    }

    // Buffers are big enough.

    nPort = *pIOBuffer;             // Get the I/O port number
from the buffer.
/*
    if (nPort >= pLDI->PortCount ||
        (nPort + DataBufferSize) > pLDI->PortCount ||
        (((ULONG)pLDI->FrameBase + nPort) & (DataBufferSize - 1))
!= 0)
    {
        return STATUS_ACCESS_VIOLATION;   // It was not legal.
              KdPrint(("PPCIDMA.SYS : ReadPort : Access
violation.\n"));
    }
*/


    if (pLDI->FrameMemType == 1)
    {
        // Address is in I/O space
        //KdPrint(("PPCIDMA.SYS : ReadPort : Address is in IO
space.\n"));
        switch (IoctlCode)
        {
        case IOCTL_PPCIDMA_READ_PORT_UCHAR:
            *(PUCHAR)pIOBuffer = READ_PORT_UCHAR(
                            (PUCHAR)((ULONG)pLDI->FrameBase +
nPort) );
                        //KdPrint(("PPCIDMA.SYS : ReadPort : Read
port %x.\n", *((ULONG)pLDI->FrameBase + nPort)));
                        //KdPrint(("PPCIDMA.SYS : ReadPort : Value
returned = %x\n", *(PUCHAR)pIOBuffer));
            break;
        case IOCTL_PPCIDMA_READ_PORT_USHORT:
            *(PUSHORT)pIOBuffer = READ_PORT_USHORT(
                            (PUSHORT)((ULONG)pLDI->FrameBase +
nPort) );
            break;
        case IOCTL_PPCIDMA_READ_PORT_ULONG:
            *(PULONG)pIOBuffer = READ_PORT_ULONG(
                            (PULONG)((ULONG)pLDI->FrameBase +
nPort) );
            break;
```

```
        }
    } else {
        // Address is in Memory space

               //KdPrint(("PPCIDMA.SYS : ReadPort : Address is in
memory space.\n"));
        switch (IoctlCode)
        {
        case IOCTL_PPCIDMA_READ_PORT_UCHAR:
            *(PUCHAR)pIOBuffer = READ_REGISTER_UCHAR(
                            (PUCHAR)((ULONG)pLDI->FrameBase +
nPort) );
            break;
        case IOCTL_PPCIDMA_READ_PORT_USHORT:
            *(PUSHORT)pIOBuffer = READ_REGISTER_USHORT(
                            (PUSHORT)((ULONG)pLDI->FrameBase +
nPort) );
            break;
        case IOCTL_PPCIDMA_READ_PORT_ULONG:
            *(PULONG)pIOBuffer = READ_REGISTER_ULONG(
                            (PULONG)((ULONG)pLDI->FrameBase +
nPort) );
            break;
        }
    }

    // Indicate # of bytes read
    //
    //KdPrint(("PPCIDMA.SYS : ReadPort : DataBufferSize = %x\n",
DataBufferSize));
    pIrp->IoStatus.Information = DataBufferSize;

    return STATUS_SUCCESS;
}



/*********************************************************
* PPciDmaIoctlWritePort
*  Handle write port IOCTLs
*********************************************************/
NTSTATUS
PPciDmaIoctlWritePort(
    IN PSKELETON_DEVICE pLDI,
    IN PIRP pIrp,
    IN PIO_STACK_LOCATION IrpStack,
    IN ULONG IoctlCode
    )

/*++

Routine Description:
    This routine processes the IOCTLs which write to the ports.

Arguments:

    pLDI        - our local device data
    pIrp        - IO request packet
    IrpStack    - The current stack location
    IoctlCode   - The ioctl code from the IRP

Return Value:
    STATUS_SUCCESS           -- OK

    STATUS_INVALID_PARAMETER -- The buffer sent to the driver
                                was too small to contain the
                                port, or the buffer which
```

```
                                would be sent back to the driver
                                was not a multiple of the data
size.

    STATUS_ACCESS_VIOLATION  -- An illegal port number was given.

--*/

{
                                // NOTE:  Use METHOD_BUFFERED
ioctls.
    PULONG pIOBuffer;           // Pointer to transfer buffer
                                //      (treated as array of
longs).
    ULONG InBufferSize ;        // Amount of data avail. from
caller.
    ULONG OutBufferSize ;       // Max data that caller can
accept.
    ULONG nPort;                // Port number to read or write.
    ULONG DataBufferSize;

    // Size of buffer containing data from application
    InBufferSize  = IrpStack-
>Parameters.DeviceIoControl.InputBufferLength;

    // Size of buffer for data to be sent to application
    OutBufferSize = IrpStack-
>Parameters.DeviceIoControl.OutputBufferLength;

    // NT copies inbuf here before entry and copies this to
outbuf after return,
    // for METHOD_BUFFERED IOCTL's.
    pIOBuffer     = (PULONG) pIrp->AssociatedIrp.SystemBuffer;

    // We don't return any data on a write port.
    pIrp->IoStatus.Information = 0;

    // Check to ensure input buffer is big enough to hold a port
number as well
    // as the data to write.
    //
    // The relative port # is a ULONG, and the data is the type
appropriate to
    // the IOCTL.
    //

    switch (IoctlCode)
    {
    default:                    // There isn't really any default
but
        /* FALL THRU */         // this will quiet the compiler.
    case IOCTL_PPCIDMA_WRITE_PORT_UCHAR:
        DataBufferSize = sizeof(UCHAR);
        break;
    case IOCTL_PPCIDMA_WRITE_PORT_USHORT:
        DataBufferSize = sizeof(USHORT);
        break;
    case IOCTL_PPCIDMA_WRITE_PORT_ULONG:
        DataBufferSize = sizeof(ULONG);
        break;
    }

    if ( InBufferSize < (sizeof(ULONG) + DataBufferSize) )
    {
        return STATUS_INVALID_PARAMETER;
    }
```

```
    nPort = *pIOBuffer++;

    if (nPort >= pLDI->PortCount ||
        (nPort + DataBufferSize) > pLDI->PortCount ||
        (((ULONG)pLDI->FrameBase + nPort) & (DataBufferSize - 1))
!= 0)
    {
        return STATUS_ACCESS_VIOLATION;   // Illegal port number
    }

    if (pLDI->FrameMemType == 1)
    {
        // Address is in I/O space

        switch (IoctlCode)
        {
        case IOCTL_PPCIDMA_WRITE_PORT_UCHAR:
            WRITE_PORT_UCHAR(
                (PUCHAR)((ULONG)pLDI->FrameBase + nPort),
                *(PUCHAR)pIOBuffer );
            break;
        case IOCTL_PPCIDMA_WRITE_PORT_USHORT:
            WRITE_PORT_USHORT(
                (PUSHORT)((ULONG)pLDI->FrameBase + nPort),
                *(PUSHORT)pIOBuffer );
            break;
        case IOCTL_PPCIDMA_WRITE_PORT_ULONG:
            WRITE_PORT_ULONG(
                (PULONG)((ULONG)pLDI->FrameBase + nPort),
                *(PULONG)pIOBuffer );
            break;
        }
    } else {
        // Address is in Memory space

        switch (IoctlCode)
        {
        case IOCTL_PPCIDMA_WRITE_PORT_UCHAR:
            WRITE_REGISTER_UCHAR(
                    (PUCHAR)((ULONG)pLDI->FrameBase + nPort),
                    *(PUCHAR)pIOBuffer );
            break;
        case IOCTL_PPCIDMA_WRITE_PORT_USHORT:
            WRITE_REGISTER_USHORT(
                    (PUSHORT)((ULONG)pLDI->FrameBase + nPort),
                    *(PUSHORT)pIOBuffer );
            break;
        case IOCTL_PPCIDMA_WRITE_PORT_ULONG:
            WRITE_REGISTER_ULONG(
                    (PULONG)((ULONG)pLDI->FrameBase + nPort),
                    *(PULONG)pIOBuffer );
            break;
        }
    }

    return STATUS_SUCCESS;
}
```

```
/*************************************************************
* PPciDmaMapBuffer
*   Map the DMA buffer into user memory space
*************************************************************/
NTSTATUS
PPciDmaMapBuffer(
    IN PSKELETON_DEVICE pLDI,
       IN PIRP pIrp,
       IN PIO_STACK_LOCATION IrpStack
       )
/*++

Routine Description:

    Given a physical address, maps this address into a user mode
process's
    address space

Arguments:

       Ummm, I'll take care of this later.

Return Value:

    STATUS_SUCCESS if sucessful, otherwise
    STATUS_UNSUCCESSFUL,
    STATUS_INSUFFICIENT_RESOURCES,
    (other STATUS_* as returned by kernel APIs)

--*/
{

    INTERFACE_TYPE           interfaceType;
    ULONG                    busNumber;
    ULONG                    length;
    UNICODE_STRING           physicalMemoryUnicodeString;
    OBJECT_ATTRIBUTES  objectAttributes;
    HANDLE                   physicalMemoryHandle  =
NULL;
    PVOID                    PhysicalMemorySection =
NULL;
    ULONG                    inIoSpace, inIoSpace2;
    NTSTATUS                 ntStatus;
    PHYSICAL_ADDRESS   physicalAddressBase;
    PHYSICAL_ADDRESS   physicalAddressEnd;
       PHYSICAL_ADDRESS      xlatedAddressBase;
       PHYSICAL_ADDRESS      xlatedAddressEnd;
    PHYSICAL_ADDRESS   viewBase;
    PHYSICAL_ADDRESS   mappedLength;
    BOOLEAN                  translateBaseAddress;
    BOOLEAN                  translateEndAddress;
    PVOID                    virtualAddress;
       ULONG                 OutputBufferLength;
       PULONG                pIOBuffer;
       PSKELETON_DEVICE      skelDev;


       OutputBufferLength = IrpStack-
>Parameters.DeviceIoControl.OutputBufferLength;
       pIOBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;

    if ( OutputBufferLength < sizeof (PVOID) )
    {
       KdPrint(("MAPMEM.SYS: Insufficient input or output
buffer\n"));
```

```
       ntStatus = STATUS_INSUFFICIENT_RESOURCES;

       goto done;
    }

       //skelDev                                    = pLDI-
>DeviceExtension;
    interfaceType                   = PCIBus;
    busNumber                       = pLDI->BusId;
    physicalAddressBase             = pLDI->LogicalAddress;
    //inIoSpace = inIoSpace2 = ppmi->AddressSpace;
       inIoSpace = inIoSpace2 = 0;          //Always map memory,
not IO
    length                                  = pLDI-
>BufferLength;

       KdPrint(("PPCIDMA.SYS : MapBuffer : PhysicalAddressBase =
%x.\n", physicalAddressBase));
       KdPrint(("PPCIDMA.SYS : MapBuffer : length = %x.\n",
length));

    //
    // Get a pointer to physical memory...
    //
    // - Create the name
    // - Initialize the data to find the object
    // - Open a handle to the oject and check the status
    // - Get a pointer to the object
    // - Free the handle
    //

    RtlInitUnicodeString (&physicalMemoryUnicodeString,
                          L"\\Device\\PhysicalMemory");

    InitializeObjectAttributes (&objectAttributes,
                                &physicalMemoryUnicodeString,
                                OBJ_CASE_INSENSITIVE,
                                (HANDLE) NULL,
                                (PSECURITY_DESCRIPTOR) NULL);

    ntStatus = ZwOpenSection (&physicalMemoryHandle,
                              SECTION_ALL_ACCESS,
                              &objectAttributes);

    if (!NT_SUCCESS(ntStatus))
    {
        KdPrint(("PPCIDMA.SYS: ZwOpenSection failed\n"));

        goto done;
    }

    ntStatus = ObReferenceObjectByHandle (physicalMemoryHandle,
                                          SECTION_ALL_ACCESS,
                                          (POBJECT_TYPE) NULL,
                                          KernelMode,
                                          &PhysicalMemorySection,

(POBJECT_HANDLE_INFORMATION) NULL);

    if (!NT_SUCCESS(ntStatus))
    {
        KdPrint(("PPCIDMA.SYS: ObReferenceObjectByHandle
failed\n"));

        goto close_handle;
    }
```

```
    //
    // Initialize the physical addresses that will be translated
    //

    physicalAddressEnd = RtlLargeIntegerAdd (physicalAddressBase,
RtlConvertUlongToLargeInteger(
                                            length));

    //
    // Translate the physical addresses.
    //

        KdPrint(("PPCIDMA.SYS : Attempting to translate
with\nphysicalAddress=%x\nphysicaladdressend=%x.\n",
                              physicalAddressBase,
physicalAddressEnd));

    translateBaseAddress =
        HalTranslateBusAddress (interfaceType,
                                busNumber,
                                physicalAddressBase,
                                &inIoSpace,
                                &xlatedAddressBase);

    translateEndAddress =
        HalTranslateBusAddress (interfaceType,
                                busNumber,
                                physicalAddressEnd,
                                &inIoSpace2,
                                &xlatedAddressEnd);

    if ( !(translateBaseAddress && translateEndAddress) )
    {
        KdPrint(("PPCIDMA.SYS: HalTranslatephysicalAddress
failed\n"));

        ntStatus = STATUS_UNSUCCESSFUL;

        goto close_handle;
    }

    //
    // Calculate the length of the memory to be mapped
    //

    mappedLength = RtlLargeIntegerSubtract (xlatedAddressEnd,
                                            xlatedAddressBase);

        KdPrint(("PPCIDMA.SYS : MapBuffer : mappedLength =
%x.\n", mappedLength));


    //
    // If the mappedlength is zero, somthing very weird happened
in the HAL
    // since the Length was checked against zero.
    //

    if (mappedLength.LowPart == 0)
    {
        KdPrint(("PPCIDMA.SYS: mappedLength.LowPart == 0\n"));

        ntStatus = STATUS_UNSUCCESSFUL;
```

```
        goto close_handle;
    }

    length = mappedLength.LowPart;



    //
    // If the address is in io space, just return the address,
otherwise
    // go through the mapping mechanism
    //
/*
    if (inIoSpace)
    {
        *((PVOID *) IoBuffer) = (PVOID)
physicalAddressBase.LowPart;
    }

    else
    {*/
        //
        // initialize view base that will receive the physical
mapped
        // address after the MapViewOfSection call.
        //

        viewBase = xlatedAddressBase;



        //
        // Let ZwMapViewOfSection pick an address
        //

        virtualAddress = NULL;



        //
        // Map the section
        //

        ntStatus = ZwMapViewOfSection (physicalMemoryHandle,
                                       (HANDLE) -1,
                                       &virtualAddress,
                                       0L,
                                       length,
                                       &viewBase,
                                       &length,
                                       ViewShare,
                                       0,
                                       PAGE_READWRITE |
PAGE_NOCACHE);

        if (!NT_SUCCESS(ntStatus))
        {
            KdPrint(("PPCIDMA.SYS: ZwMapViewOfSection
failed\n"));

            goto close_handle;
        }

        //
        // Mapping the section above rounded the physical address
down to the
```

```
        // nearest 64 K boundary. Now return a virtual address
that sits where
        // we wnat by adding in the offset from the beginning of
the section.
        //


        (ULONG) virtualAddress +=
(ULONG)xlatedAddressBase.LowPart -
                                 (ULONG)viewBase.LowPart;

               KdPrint(("PPCIDMA.SYS : virtualAddress is %x.\n",
virtualAddress));

        //*((PVOID *) IoBuffer) = virtualAddress;
               *((PVOID *)pIOBuffer) = virtualAddress;
               pIrp->IoStatus.Information = sizeof(ULONG);
       //Size of output buffer

    //}

    ntStatus = STATUS_SUCCESS;



close_handle:

    ZwClose (physicalMemoryHandle);

done:

    return ntStatus;
}
```

```
/*******************************************************
* ReturnMemoryInfo
*       Returns to the caller the physical address and size
* of the DMA buffer allocated.
* The value returned is a 32-bit value, the lowest 32-bits
* of the physical address.
*/
NTSTATUS PPciDmaReturnMemoryInfo(
        IN PSKELETON_DEVICE pLDI,
        IN PIRP pIrp,
        IN PIO_STACK_LOCATION IrpStack
        )
{
        ULONG   OutputBufferLength;
        PULONG  pIOBuffer;
        PHYSICAL_ADDRESS PhysicalAddress;

        OutputBufferLength = IrpStack-
>Parameters.DeviceIoControl.OutputBufferLength;
        pIOBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;

        pIrp->IoStatus.Information = sizeof(ULONG);  //The size
of the output buffer

    if ( OutputBufferLength < sizeof (ULONG) )
        //This line too implies 32-bit address space
    {
       KdPrint(("PPCIDMA.SYS: Insufficient input or output
buffer\n"));
       return(STATUS_INSUFFICIENT_RESOURCES);
    }
```

```
        PhysicalAddress = pLDI->LogicalAddress;             //64-
bit value

        *(PULONG)pIOBuffer = PhysicalAddress.LowPart;
        //If we ever exceed 32-bits of addressable space, then
I'll uncomment the line below
        //*(PULONG)(pIOBuffer + sizeof(PhysicalAddress.LowPart))
= PhysicalAddress.HighPart;

        return(STATUS_SUCCESS);
}



/*
 * Overrides for Emacs to get consistency.
 * Emacs will notice this stuff at the end of the file and
automatically
 * adjust the settings for this buffer only.  This must remain at
the end
 * of the file.
 * ---------------------------------------------------------------
-------------
 * Local variables:
 * tab-width: 8
 * c-brace-imaginary-offset: 0
 * c-brace-offset: -4
 * c-argdecl-indent: 4
 * c-label-offset: -2
 * c-continued-statement-offset: 4
 * c-continued-brace-offset: 0
 * c-indent-level: 4
 * End:
 */
```

PCIDMA_DEV.H

```
/*
 * This is the structure for Skeleton device info
 */
typedef struct {
    PVOID FrameBase;   // Frame buffer address in system memory
                       //This is the base IO address for the
                       //PPCIDMA driver.
    ULONG FrameMemType; // Address space: 0x0 = mem, 0x1 = I/O
    ULONG PortCount;    //Length occupied in IO space

    PKINTERRUPT KIntrObj;     //Interrrupt object from
                              //IoConnectInterrupt
    ULONG KIntrVector;        // Mapped system interrupt vector
    KIRQL KIrql;
    KAFFINITY KIntrAffinity;  //The processor set this interrupt
                              //affects
    FAST_MUTEX IrpMutex;      // Ensure 1 dispatch entry at a
                              //time
    KSPIN_LOCK DeviceSpinLock;

    ULONG BusId;
    ULONG SlotId;
    PHYSICAL_ADDRESS MemStart;       //Physical address of the
                                     //DMA buffer
    ULONG MemLength;                 //Length of the DMA buffer
    ULONG IntrLevel;
    ULONG IntrVector;
    ULONG IntrAffinity;
```

```
/* StartIo subroutine fields.  Used by QueryVideo() */
KDPC StartIoDpc;      //Used by the query video routine
KTIMER StartIoTimer;  //Timer used in StartIo subroutine
ULONG StartIoState;   //Current state of StartIo subroutine
PVOID StartIoBuffer;  //Current buffer we are using

/* Some DMA related fields */
PADAPTER_OBJECT AdaptorObj;   //Pointer to DMA adaptor object
ULONG DmaMapRegisters;        //Number of DMA map registers
PVOID MapRegisterBase;        //DMA map register base

//DMA buffer related fields
PVOID VirtualAddress;     //Virtual address of MDL for DMA
                          //transfer, return from
                          //HalAllocateCommonBuffer
ULONG BufferLength;       //Length of the DMA buffer
PHYSICAL_ADDRESS LogicalAddress;  //Logical (sort of
                                  //physical) address the
                                  //device will
                                  //use to transfer data
PMDL Pmdl;           //Pointer to MDL from IoAllocateMdl


KTIMER DeviceCheckTimer;  //Timer to check that DMA hasn't
                          //failed
BOOLEAN TimerStarted;     // Has the timer been started
KDPC TimerDpc;            // The DPC to run on timer trigger
BOOLEAN TransferDone;     // When the timeout occurs, find
                          //out if the tranfer was already
                          //done
PVOID IrpSystemBuffer;    // System buffer for the current
                          // IRP
ULONG IrpBufLen;   //Buffer length for the current IRP */
    NTSTATUS IrpStatus;        //The status of the transfer
    ULONG IrpBytesTransferred; //Number of bytes that were
                               //transfered

    CCHAR OperationType;       //current command (ie IRP_MJ_READ)
    ULONG TotalTransferLength;  //length of current transfer */
    BOOLEAN RequestDpc;       //Set this if the routine wishes to
    BOOLEAN DpcRequested;
    ULONG RestoreChannel;     //Channel was in use and needs
                              //restoring

    ULONG syncParam1;         //Parameters for the synch routine
    ULONG syncParam2;
    ULONG syncParam3;
    union {                   //Return value if needed
        ULONG ULong;
        int Int;
        long Long;
    } syncResult;

} SKELETON_DEVICE, *PSKELETON_DEVICE;

#define offsetof(type, field) ((ULONG) ((char *) &((type *) 0)-
>field))

#ifdef LITTLE_ENDIAN
#  define LITTLE_ENDIAN_16(x) (x)
#  define LITTLE_ENDIAN_32(x) (x)

static __inline unsigned short BIG_ENDIAN_16(unsigned short x)
{
    return (((x & 0x00FF) << 8) + ((x & 0xFF00) >> 8));
}
```

```
static __inline unsigned int BIG_ENDIAN_32(unsigned int x)
{
    return (((x & 0x000000FF) << 24) + ((x & 0x0000FF00) << 8) +
           ((x & 0x00FF0000) >> 8) + ((x & 0xFF000000) >> 24));
}
#else
#  define BIG_ENDIAN_16(x) (x)
#  define BIG_ENDIAN_32(x) (x)
static __inline unsigned short LITTLE_ENDIAN_16(unsigned short n)
{
   return (((n & 0x00FF) << 8) + ((n & 0xFF00) >> 8));
}

static __inline unsigned int LITTLE_ENDIAN_32(unsigned int n)
{
    return (((n & 0x000000FF) << 24) + ((n & 0x0000FF00) << 8) +
           ((n & 0x00FF0000) >> 8) + ((n & 0xFF000000) >> 24));
}
#endif
```

PPCIDMA_IOCTL.H

```
/*
 *------------------------------------------------------------------
 * Module Name:
 *      pcidma_ioctl.h
 *
 * Abstract:
 *      Include file for user and kernel space.
 *
 * Environment:
 *      Kernel and user modes
 *
 * Revision History:
 *
 *------------------------------------------------------------------
 */

//This is the maximum buffer length required for DMA transfers.
//Since this is the amount of memory that will actually be
//allocated and since it does need to be contiguous, please make
//sure that you do not allocate more memory than you need.
//Version 1.0 of the driver requires that this option be changed
//at compile time.
#define PPCIDMA_MAX_DMA_BUFFER_LENGTH 4000 * 1024


//The vendor ID and device ID of the PCI device
#define PCIDMA_VENDORID 0x010e8
#define PCIDMA_DEVICEID 0x04750

/*
 * Define the various device type values.  Note that values used
 * by Microsoft Corporation are in the range 0-32767, and 32768-
 * 65535 are reserved for use by customers.
 */
#define FILE_DEVICE_SKELETON  0x0000CBFC

/*
 * Macro definition for defining IOCTL and FSCTL function control
 * codes.  Note that function codes 0-2047 are reserved for
 * Microsoft Corporation, and 2048-4095 are reserved for
 * customers.
 */
#define SKELETON_IOCTL_BASE  0x800
```

```
/*
 * Define the PciDma IOCTLs.  There are two forms for these
 * defines: the NT ioctl name and the Skeleton ioctl name.
 */

#define IOCTL_SKELETON_NNN(offset, method, access) \
    (ULONG) CTL_CODE(FILE_DEVICE_SKELETON, SKELETON_IOCTL_BASE +
(offset), \
                     (method), (access))

#define IOCTL_PPCIDMA_MAP_USER_PHYSICAL_MEMORY \
    IOCTL_SKELETON_NNN(0, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_PPCIDMA_UNMAP_USER_PHYSICAL_MEMORY \
    IOCTL_SKELETON_NNN(1, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_PPCIDMA_READ_PORT_UCHAR \
        IOCTL_SKELETON_NNN(2, METHOD_BUFFERED, FILE_READ_ACCESS)

#define IOCTL_PPCIDMA_READ_PORT_USHORT \
        IOCTL_SKELETON_NNN(3, METHOD_BUFFERED, FILE_READ_ACCESS)

#define IOCTL_PPCIDMA_READ_PORT_ULONG \
        IOCTL_SKELETON_NNN(4, METHOD_BUFFERED, FILE_READ_ACCESS)

#define IOCTL_PPCIDMA_WRITE_PORT_UCHAR \
        IOCTL_SKELETON_NNN(5, METHOD_BUFFERED, FILE_WRITE_ACCESS)

#define IOCTL_PPCIDMA_WRITE_PORT_USHORT \
        IOCTL_SKELETON_NNN(6, METHOD_BUFFERED, FILE_WRITE_ACCESS)

#define IOCTL_PPCIDMA_WRITE_PORT_ULONG \
        IOCTL_SKELETON_NNN(7, METHOD_BUFFERED, FILE_WRITE_ACCESS)

#define IOCTL_PPCIDMA_RETURN_MEMORY_INFORMATION \
        IOCTL_SKELETON_NNN(8, METHOD_BUFFERED, FILE_ANY_ACCESS)

typedef struct  _PPCIDMA_WRITE_INPUT {
    ULONG   PortNumber;     // Port # to write to
    union   {               // Data to be output to port
        ULONG   LongData;
        USHORT  ShortData;
        UCHAR   CharData;
    };
}   PPCIDMA_WRITE_INPUT;
```

DPIB.C

```
// Generic Port I/O driver for NT  VERSION 1.0
// For the Eisa

#include "dpib.h"
#include "stdlib.h"

NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath
    )

/*++

Routine Description:
    This routine is the entry point for the driver.  It is
    responsible for setting the dispatch entry points in the
    driver object and creating the device object.  Any resources
```

```
    such as ports, interrupts and DMA channels used must be
    reported.  A symbolic link must be created between the device
    name and an entry in \DosDevices in order to allow Win32
    applications to open the device.

Arguments:

    DriverObject - Pointer to driver object created by the
system.

Return Value:

    STATUS_SUCCESS if the driver initialized correctly, otherwise
    an error indicating the reason for failure.

--*/

{
    ULONG PortBase;       // Port location, in NT's address form.
    ULONG PortCount;      // Count of contiguous I/O ports
    PHYSICAL_ADDRESS PortAddress;

    PLOCAL_DEVICE_INFO pLocalInfo;  //Device extension: local
                                    //information for each
device.
    NTSTATUS Status;
    PDEVICE_OBJECT DeviceObject;

    CM_RESOURCE_LIST ResourceList;  //Resource usage list to
                                    //report to system
    BOOLEAN ResourceConflict;       //This is set true if our
                                    //I/O ports conflict with
                                    //another driver

    // Try to retrieve base I/O port and range from the
Parameters
    // key of our entry in the Registry.
    // If there isn't anything specified then use the values
    // compiled into this driver.
    {
        static  WCHAR             SubKeyString[] =
L"\\Parameters";
        UNICODE_STRING            paramPath;
        RTL_QUERY_REGISTRY_TABLE  paramTable[3];
        ULONG                     DefaultBase = BASE_PORT;
        ULONG                     DefaultCount = NUMBER_PORTS;

        //
        // Since the registry path parameter is a "counted"
        // UNICODE string, it might not be zero terminated.  For
        // a very short time allocate memory to hold the registry
        // path as well as the Parameters key name zero
terminated
        // so that we can use it to delve into the registry.
        //

        paramPath.MaximumLength = RegistryPath->Length +
                                      sizeof(SubKeyString);
        paramPath.Buffer = ExAllocatePool(PagedPool,
                                  paramPath.MaximumLength);

        if (paramPath.Buffer != NULL)
        {
            RtlMoveMemory(
                paramPath.Buffer, RegistryPath->Buffer,
                                  RegistryPath->Length);
```

```
    RtlMoveMemory(
        &paramPath.Buffer[RegistryPath->Length / 2],
        SubKeyString,
        sizeof(SubKeyString));

    paramPath.Length = paramPath.MaximumLength - 2;

    RtlZeroMemory(&paramTable[0], sizeof(paramTable));

    paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
    paramTable[0].Name = L"IoPortAddress";
    paramTable[0].EntryContext = &PortBase;
    paramTable[0].DefaultType = REG_DWORD;
    paramTable[0].DefaultData = &DefaultBase;
    paramTable[0].DefaultLength = sizeof(ULONG);

    paramTable[1].Flags = RTL_QUERY_REGISTRY_DIRECT;
    paramTable[1].Name = L"IoPortCount";
    paramTable[1].EntryContext = &PortCount;
    paramTable[1].DefaultType = REG_DWORD;
    paramTable[1].DefaultData = &DefaultCount;
    paramTable[1].DefaultLength = sizeof(ULONG);

    if (!NT_SUCCESS(RtlQueryRegistryValues(
        RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
        paramPath.Buffer, &paramTable[0], NULL, NULL)))
    {
        PortBase = DefaultBase;
        PortCount = DefaultCount;
    }
    ExFreePool(paramPath.Buffer);
}
```

```
    }

    PortAddress.LowPart  = PortBase;
    PortAddress.HighPart = 0;

    // Register resource usage (ports)
    //
    // This ensures that there isn't a conflict between this
    // driver and a previously loaded one or a future loaded one.

    RtlZeroMemory((PVOID)&ResourceList, sizeof(ResourceList));

    ResourceList.Count = 1;
    ResourceList.List[0].InterfaceType = Isa;
    // ResourceList.List[0].Busnumber = 0;             Already 0
    ResourceList.List[0].PartialResourceList.Count = 1;

    ResourceList.List[0].PartialResourceList. \
        PartialDescriptors[0].Type = CmResourceTypePort;

    ResourceList.List[0].PartialResourceList. \
        PartialDescriptors[0].ShareDisposition =
                        CmResourceShareDriverExclusive;

    ResourceList.List[0].PartialResourceList. \
        PartialDescriptors[0].Flags =
                        CM_RESOURCE_PORT_IO;
    ResourceList.List[0].PartialResourceList. \
        PartialDescriptors[0].u.Port.Start =
                        PortAddress;

    ResourceList.List[0].PartialResourceList. \
        PartialDescriptors[0].u.Port.Length =
```

```
                    PortCount;

// Report our resource usage and detect conflicts
Status = IoReportResourceUsage(
                NULL,
                DriverObject,
                &ResourceList,
                sizeof(ResourceList),
                NULL,
                NULL,
                0,
                FALSE,
                &ResourceConflict);

if (ResourceConflict)
    Status = STATUS_DEVICE_CONFIGURATION_ERROR;

if (!NT_SUCCESS(Status))
{
    KdPrint( ("Resource reporting problem %8X", Status) );

    return Status;
}

// Initialize the driver object dispatch table.
// NT sends requests to these routines.

DriverObject->MajorFunction[IRP_MJ_CREATE]          =
                                                DpibDispatch;
DriverObject->MajorFunction[IRP_MJ_CLOSE]           =
                                                DpibDispatch;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]  =
                                                DpibDispatch;
DriverObject->DriverUnload                          =
                                                DpibUnload;

// Create our device.
Status = DpibCreateDevice(
                DPIB_DEVICE_NAME,
                DPIB_TYPE,
                DriverObject,
                &DeviceObject
                );

if ( NT_SUCCESS(Status) )
{
    PHYSICAL_ADDRESS MappedAddress;
    ULONG MemType;

    // Convert the IO port address into a form NT likes.
    MemType = 1;                    // located in IO space
    HalTranslateBusAddress( Isa,
                            0,
                            PortAddress,
                            &MemType,
                            &MappedAddress );


    // Initialize the local driver info for each device
    // object.
    pLocalInfo = (PLOCAL_DEVICE_INFO)
                    DeviceObject->DeviceExtension;

    if (MemType == 0)
    {
        // Port is accessed through memory space - so get
```

```
        // a virtual address

        pLocalInfo->PortWasMapped = TRUE;

        // BUGBUG
        // MmMapIoSpace can fail if we run out of PTEs,
        // we should be checking the return value here

        pLocalInfo->PortBase = MmMapIoSpace(MappedAddress,
                                        PortCount, FALSE);
    }
    else
    {
        pLocalInfo->PortWasMapped = FALSE;
        pLocalInfo->PortBase = (PVOID)MappedAddress.LowPart;
    }

    pLocalInfo->DeviceObject    = DeviceObject;
    pLocalInfo->DeviceType      = DPIB_TYPE;
    pLocalInfo->PortCount       = PortCount;
    pLocalInfo->PortMemoryType  = MemType;
}
else
{
    //
    // Error creating device - release resources
    //

    RtlZeroMemory((PVOID)&ResourceList,
                        sizeof(ResourceList));

    // Unreport our resource usage
    Status = IoReportResourceUsage(
                        NULL,
                        DriverObject,
                        &ResourceList,
                        sizeof(ResourceList),
                        NULL,
                        NULL,
                        0,
                        FALSE,
                        &ResourceConflict);
    }

    return Status;
}

NTSTATUS
DpibCreateDevice(
    IN   PWSTR              PrototypeName,
    IN   DEVICE_TYPE        DeviceType,
    IN   PDRIVER_OBJECT     DriverObject,
    OUT  PDEVICE_OBJECT     *ppDevObj
    )

/*++

Routine Description:
    This routine creates the device object and the symbolic link
    in \DosDevices.

    Ideally a name derived from a "Prototype", with a number
    appended at the end should be used.  For simplicity, just use
    the fixed name defined in the include file.  This means that
    only one device can be created.
```

```
    A symbolic link must be created between the device name and
    an entry in \DosDevices in order to allow Win32 applications
    to open the device.

Arguments:

    PrototypeName - Name base, # WOULD be appended to this.

    DeviceType - Type of device to create

    DriverObject - Pointer to driver object created by the
system.

    ppDevObj - Pointer to place to store pointer to created
               device object

Return Value:

    STATUS_SUCCESS if the device and link are created correctly,
    otherwise an error indicating the reason for failure.

--*/


{
    NTSTATUS Status;                // Status of utility calls
    UNICODE_STRING NtDeviceName;
    UNICODE_STRING Win32DeviceName;


    // Get UNICODE name for device.

    RtlInitUnicodeString(&NtDeviceName, PrototypeName);

    Status = IoCreateDevice(        // Create it.
                DriverObject,
                sizeof(LOCAL_DEVICE_INFO),
                &NtDeviceName,
                DeviceType,
                0,
                FALSE,                      // Not Exclusive
                ppDevObj
                );

    if (!NT_SUCCESS(Status))
        return Status;             // Give up if create failed.

    // Clear local device info memory
    RtlZeroMemory((*ppDevObj)->DeviceExtension,
                          sizeof(LOCAL_DEVICE_INFO));

    //
    // Set up the rest of the device info
    //  These are used for IRP_MJ_READ and IRP_MJ_WRITE which
    //  we don't use
    //
    //  (*ppDevObj)->Flags |= DO_BUFFERED_IO;
    //  (*ppDevObj)->AlignmentRequirement = FILE_BYTE_ALIGNMENT;
    //

    RtlInitUnicodeString(&Win32DeviceName, DOS_DEVICE_NAME);

    Status = IoCreateSymbolicLink( &Win32DeviceName,
                                   &NtDeviceName );

    if (!NT_SUCCESS(Status))    // If we we couldn't create the
```

```
                                // link then
    {                           //  abort installation.
        IoDeleteDevice(*ppDevObj);
    }

    return Status;
}

NTSTATUS
DpibDispatch(
    IN    PDEVICE_OBJECT pDO,
    IN    PIRP pIrp
    )

/*++

Routine Description:
    This routine is the dispatch handler for the driver.  It is
    responsible for processing the IRPs.

Arguments:

    pDO - Pointer to device object.

    pIrp - Pointer to the current IRP.

Return Value:

    STATUS_SUCCESS if the IRP was processed successfully,
    otherwise an error indicating the reason for failure.

--*/
{
    PLOCAL_DEVICE_INFO pLDI;
    PIO_STACK_LOCATION pIrpStack;
    NTSTATUS Status;

    //  Initialize the irp info field.
    //      This is used to return the number of bytes
transfered.

    pIrp->IoStatus.Information = 0;
    //Get local info struct
    pLDI = (PLOCAL_DEVICE_INFO)pDO->DeviceExtension;

    pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

    //  Set default return status
    Status = STATUS_NOT_IMPLEMENTED;

    // Dispatch based on major fcn code.

    switch (pIrpStack->MajorFunction)
    {
        case IRP_MJ_CREATE:
        case IRP_MJ_CLOSE:
            // We don't need any special processing on
            // open/close so we'll
            // just return success.
            Status = STATUS_SUCCESS;
            break;

        case IRP_MJ_DEVICE_CONTROL:
            //  Dispatch on IOCTL
```

```
            switch (pIrpStack-
>Parameters.DeviceIoControl.IoControlCode)
            {
            case IOCTL_DPIB_READ_PORT_UCHAR:
            case IOCTL_DPIB_READ_PORT_USHORT:
            case IOCTL_DPIB_READ_PORT_ULONG:
                Status = DpibIoctlReadPort(
                            pLDI,
                            pIrp,
                            pIrpStack,
                            pIrpStack-
>Parameters.DeviceIoControl.IoControlCode
                            );
                break;

            case IOCTL_DPIB_WRITE_PORT_UCHAR:
            case IOCTL_DPIB_WRITE_PORT_USHORT:
            case IOCTL_DPIB_WRITE_PORT_ULONG:
                Status = DpibIoctlWritePort(
                            pLDI,
                            pIrp,
                            pIrpStack,
                            pIrpStack-
>Parameters.DeviceIoControl.IoControlCode
                            );
                break;
            }
            break;
    }

    // We're done with I/O request.  Record the status of the
    // I/O action.
    pIrp->IoStatus.Status = Status;

    // Don't boost priority when returning since this took
    // little time.
    IoCompleteRequest(pIrp, IO_NO_INCREMENT );

    return Status;
}

NTSTATUS
DpibIoctlReadPort(
    IN PLOCAL_DEVICE_INFO pLDI,
    IN PIRP pIrp,
    IN PIO_STACK_LOCATION IrpStack,
    IN ULONG IoctlCode  )


/*++

Routine Description:
    This routine processes the IOCTLs which read from the ports.

Arguments:

    pLDI        - our local device data
    pIrp        - IO request packet
    IrpStack    - The current stack location
    IoctlCode   - The ioctl code from the IRP

Return Value:
    STATUS_SUCCESS           -- OK

    STATUS_INVALID_PARAMETER -- The buffer sent to the driver
                                was too small to contain the
```

```
                                port, or the buffer which
                                would be sent back to the driver
                                was not a multiple of the data
                                size.

    STATUS_ACCESS_VIOLATION  -- An illegal port number was given.

--*/

{
                            // NOTE:  Use METHOD_BUFFERED ioctls.
    PULONG pIOBuffer;       // Pointer to transfer buffer
                            // (treated as an array of longs).
    ULONG InBufferSize;     // Amount of data avail. from caller.
    ULONG OutBufferSize;    // Max data that caller can accept.
    ULONG nPort;            // Port number to read
    ULONG DataBufferSize;

    // Size of buffer containing data from application
    InBufferSize  = IrpStack-
>Parameters.DeviceIoControl.InputBufferLength;

    // Size of buffer for data to be sent to application
    OutBufferSize = IrpStack-
>Parameters.DeviceIoControl.OutputBufferLength;

    // NT copies inbuf here before entry and copies this to
    // outbuf after return, for METHOD_BUFFERED IOCTL's.
    pIOBuffer     = (PULONG)pIrp->AssociatedIrp.SystemBuffer;

    // Check to ensure input buffer is big enough to hold a port
    // number and the output buffer is at least as big as the
    // port data width.
    //
    switch (IoctlCode)
    {
    default:              // There isn't really any default but
       /* FALL THRU */         // this will quiet the compiler.
    case IOCTL_DPIB_READ_PORT_UCHAR:
        DataBufferSize = sizeof(UCHAR);
        break;
    case IOCTL_DPIB_READ_PORT_USHORT:
        DataBufferSize = sizeof(USHORT);
        break;
    case IOCTL_DPIB_READ_PORT_ULONG:
        DataBufferSize = sizeof(ULONG);
        break;
    }

    if ( InBufferSize != sizeof(ULONG) || OutBufferSize <
DataBufferSize )
    {
        return STATUS_INVALID_PARAMETER;
    }

    // Buffers are big enough.

    nPort = *pIOBuffer;     // Get the I/O port number from
                            // the buffer.

    if (nPort >= pLDI->PortCount ||
        (nPort + DataBufferSize) > pLDI->PortCount ||
        (((ULONG)pLDI->PortBase + nPort) & (DataBufferSize - 1))
!= 0)
    {
        return STATUS_ACCESS_VIOLATION;   // It was not legal.
```

```
    }

    if (pLDI->PortMemoryType == 1)
    {
        // Address is in I/O space

        switch (IoctlCode)
        {
        case IOCTL_DPIB_READ_PORT_UCHAR:
            *(PUCHAR)pIOBuffer = READ_PORT_UCHAR(
                            (PUCHAR)((ULONG)pLDI->PortBase +
nPort) );
            break;
        case IOCTL_DPIB_READ_PORT_USHORT:
            *(PUSHORT)pIOBuffer = READ_PORT_USHORT(
                            (PUSHORT)((ULONG)pLDI->PortBase +
nPort) );
            break;
        case IOCTL_DPIB_READ_PORT_ULONG:
            *(PULONG)pIOBuffer = READ_PORT_ULONG(
                            (PULONG)((ULONG)pLDI->PortBase +
nPort) );
            break;
        }
    } else {
        // Address is in Memory space

        switch (IoctlCode)
        {
        case IOCTL_DPIB_READ_PORT_UCHAR:
            *(PUCHAR)pIOBuffer = READ_REGISTER_UCHAR(
                            (PUCHAR)((ULONG)pLDI->PortBase +
nPort) );
            break;
        case IOCTL_DPIB_READ_PORT_USHORT:
            *(PUSHORT)pIOBuffer = READ_REGISTER_USHORT(
                            (PUSHORT)((ULONG)pLDI->PortBase +
nPort) );
            break;
        case IOCTL_DPIB_READ_PORT_ULONG:
            *(PULONG)pIOBuffer = READ_REGISTER_ULONG(
                            (PULONG)((ULONG)pLDI->PortBase +
nPort) );
            break;
        }
    }

    // Indicate # of bytes read
    //

    pIrp->IoStatus.Information = DataBufferSize;

    return STATUS_SUCCESS;
}


NTSTATUS
DpibIoctlWritePort(
    IN PLOCAL_DEVICE_INFO pLDI,
    IN PIRP pIrp,
    IN PIO_STACK_LOCATION IrpStack,
    IN ULONG IoctlCode
    )

/*++
```

```
Routine Description:
    This routine processes the IOCTLs which write to the ports.

Arguments:

    pLDI        - our local device data
    pIrp        - IO request packet
    IrpStack    - The current stack location
    IoctlCode   - The ioctl code from the IRP

Return Value:
    STATUS_SUCCESS           -- OK

    STATUS_INVALID_PARAMETER -- The buffer sent to the driver
                                was too small to contain the
                                port, or the buffer which
                                would be sent back to the driver
                                was not a multiple of the data
                                size.

    STATUS_ACCESS_VIOLATION  -- An illegal port number was given.

--*/

{
                            // NOTE:  Use METHOD_BUFFERED ioctls.
    PULONG pIOBuffer;       // Pointer to transfer buffer
                            //      (treated as array of longs).
    ULONG InBufferSize ;    // Amount of data avail. from caller.
    ULONG OutBufferSize ;   // Max data that caller can accept.
    ULONG nPort;            // Port number to read or write.
    ULONG DataBufferSize;

    // Size of buffer containing data from application
    InBufferSize  = IrpStack-
>Parameters.DeviceIoControl.InputBufferLength;

    // Size of buffer for data to be sent to application
    OutBufferSize = IrpStack-
>Parameters.DeviceIoControl.OutputBufferLength;

    // NT copies inbuf here before entry and copies this to
    // outbuf after return, for METHOD_BUFFERED IOCTL's.
    pIOBuffer     = (PULONG) pIrp->AssociatedIrp.SystemBuffer;

    // We don't return any data on a write port.
    pIrp->IoStatus.Information = 0;

    // Check to ensure input buffer is big enough to hold a port
    // number as well as the data to write.
    //
    // The relative port # is a ULONG, and the data is the type
    // appropriate to the IOCTL.
    //

    switch (IoctlCode)
    {
    default:              // There isn't really any default but
       /* FALL THRU */         // this will quiet the compiler.
    case IOCTL_DPIB_WRITE_PORT_UCHAR:
        DataBufferSize = sizeof(UCHAR);
        break;
    case IOCTL_DPIB_WRITE_PORT_USHORT:
        DataBufferSize = sizeof(USHORT);
        break;
    case IOCTL_DPIB_WRITE_PORT_ULONG:
```

```
        DataBufferSize = sizeof(ULONG);
        break;
    }

    if ( InBufferSize < (sizeof(ULONG) + DataBufferSize) )
    {
        return STATUS_INVALID_PARAMETER;
    }

    nPort = *pIOBuffer++;

    if (nPort >= pLDI->PortCount ||
        (nPort + DataBufferSize) > pLDI->PortCount ||
        (((ULONG)pLDI->PortBase + nPort) & (DataBufferSize - 1))
!= 0)
    {
        return STATUS_ACCESS_VIOLATION;   // Illegal port number
    }

    if (pLDI->PortMemoryType == 1)
    {
        // Address is in I/O space

        switch (IoctlCode)
        {
        case IOCTL_DPIB_WRITE_PORT_UCHAR:
            WRITE_PORT_UCHAR(
                (PUCHAR)((ULONG)pLDI->PortBase + nPort),
                *(PUCHAR)pIOBuffer );
            break;
        case IOCTL_DPIB_WRITE_PORT_USHORT:
            WRITE_PORT_USHORT(
                (PUSHORT)((ULONG)pLDI->PortBase + nPort),
                *(PUSHORT)pIOBuffer );
            break;
        case IOCTL_DPIB_WRITE_PORT_ULONG:
            WRITE_PORT_ULONG(
                (PULONG)((ULONG)pLDI->PortBase + nPort),
                *(PULONG)pIOBuffer );
            break;
        }
    } else {
        // Address is in Memory space

        switch (IoctlCode)
        {
        case IOCTL_DPIB_WRITE_PORT_UCHAR:
            WRITE_REGISTER_UCHAR(
                    (PUCHAR)((ULONG)pLDI->PortBase + nPort),
                    *(PUCHAR)pIOBuffer );
            break;
        case IOCTL_DPIB_WRITE_PORT_USHORT:
            WRITE_REGISTER_USHORT(
                    (PUSHORT)((ULONG)pLDI->PortBase + nPort),
                    *(PUSHORT)pIOBuffer );
            break;
        case IOCTL_DPIB_WRITE_PORT_ULONG:
            WRITE_REGISTER_ULONG(
                    (PULONG)((ULONG)pLDI->PortBase + nPort),
                    *(PULONG)pIOBuffer );
            break;
        }
    }

    return STATUS_SUCCESS;
}
```

```
VOID
DpibUnload(
    PDRIVER_OBJECT DriverObject
    )

/*++

Routine Description:
    This routine prepares our driver to be unloaded.  It is
    responsible for freeing all resources allocated by
    DriverEntry as well as any allocated while the driver was
    running.  The symbolic link must be deleted as well.

Arguments:

    DriverObject - Pointer to driver object created by the
system.

Return Value:

    None

--*/

{
    PLOCAL_DEVICE_INFO pLDI;
    CM_RESOURCE_LIST NullResourceList;
    BOOLEAN ResourceConflict;
    UNICODE_STRING Win32DeviceName;

    // Find our global data

    pLDI = (PLOCAL_DEVICE_INFO)DriverObject->DeviceObject-
>DeviceExtension;

    // Unmap the ports

    if (pLDI->PortWasMapped)
    {
        MmUnmapIoSpace(pLDI->PortBase, pLDI->PortCount);
    }

    // Report we're not using any hardware.  If we don't do this
    // then we'll conflict with ourselves (!) on the next load

    RtlZeroMemory((PVOID)&NullResourceList,
sizeof(NullResourceList));

    IoReportResourceUsage(
              NULL,
              DriverObject,
              &NullResourceList,
              sizeof(ULONG),
              NULL,
              NULL,
              0,
              FALSE,
              &ResourceConflict );

    // Assume all handles are closed down.
    // Delete the things we allocated - devices, symbolic links

    RtlInitUnicodeString(&Win32DeviceName, DOS_DEVICE_NAME);

    IoDeleteSymbolicLink(&Win32DeviceName);
```

```
    IoDeleteDevice(pLDI->DeviceObject);
}


//morrph.h
//Adapted from genport.h from the SDK
#include <ntddk.h>
#include <string.h>
#include <devioctl.h>
#include "dpib_ioctl.h"        // Get IOCTL interface definitions

/* Default base port, and # of ports */
#define BASE_PORT       0x304
#define NUMBER_PORTS        3
```

DPIB.H

```
// NT device name
#define DPIB_DEVICE_NAME L"\\Device\\Dpib0"

// File system device name.   When you execute a CreateFile call
// to open the device, use "\\.\DpibDev", or, given C's
// conversion of \\ to \, use
// "\\\\.\\DpibDev"

#define DOS_DEVICE_NAME L"\\DosDevices\\DpibDev"


// driver local data structure specific to each device object
typedef struct _LOCAL_DEVICE_INFO {
    ULONG               DeviceType;     // Our private Device
Type
    BOOLEAN             PortWasMapped;  // If TRUE, we have to
                                        // unmap on unload
    PVOID               PortBase;       // base port address
    ULONG               PortCount;      // Count of I/O addresses
                                        // used
    ULONG               PortMemoryType; // HalTranslateBusAddress
                                        // MemoryType
    PDEVICE_OBJECT      DeviceObject;   // The Gpd device object.
} LOCAL_DEVICE_INFO, *PLOCAL_DEVICE_INFO;

/******************** function prototypes ********************/
//

NTSTATUS    DriverEntry(        IN  PDRIVER_OBJECT DriverObject,
                                IN  PUNICODE_STRING RegistryPath);

NTSTATUS    DpibCreateDevice(   IN  PWSTR szPrototypeName,
                                IN  DEVICE_TYPE DeviceType,
                                IN  PDRIVER_OBJECT DriverObject,
                                OUT PDEVICE_OBJECT *ppDevObj   );

NTSTATUS    DpibDispatch(       IN  PDEVICE_OBJECT pDO,
                                IN  PIRP pIrp                  );

NTSTATUS    DpibIoctlReadPort(  IN  PLOCAL_DEVICE_INFO pLDI,
                                IN  PIRP pIrp,
                                IN  PIO_STACK_LOCATION IrpStack,
                                IN  ULONG IoctlCode            );

NTSTATUS    DpibIoctlWritePort( IN  PLOCAL_DEVICE_INFO pLDI,
                                IN  PIRP pIrp,
                                IN  PIO_STACK_LOCATION IrpStack,
                                IN  ULONG IoctlCode            );
```

```
VOID        DpibUnload(         IN  PDRIVER_OBJECT DriverObject);
```

DPIB_IOCTL.H

```
// gpioctl.h    Include file for Generic Port I/O Example Driver
//
// Define the IOCTL codes we will use.  The IOCTL code contains
// a command identifier, plus other information about the device,
// the type of access with which the file must have been opened,
// and the type of buffering.
//
// Adapted from Microsoft's DDK by Panos Arvanitis, 9/13/96

// Device type           -- in the "User Defined" range."
#define DPIB_TYPE 43425

// The IOCTL function codes from 0x800 to 0xFFF are for customer
// use.

#define IOCTL_DPIB_READ_PORT_UCHAR \
  CTL_CODE( DPIB_TYPE, 0xB00, METHOD_BUFFERED, FILE_READ_ACCESS)

#define IOCTL_DPIB_READ_PORT_USHORT \
  CTL_CODE( DPIB_TYPE, 0xB01, METHOD_BUFFERED, FILE_READ_ACCESS)

#define IOCTL_DPIB_READ_PORT_ULONG \
  CTL_CODE( DPIB_TYPE, 0xB02, METHOD_BUFFERED, FILE_READ_ACCESS)

#define IOCTL_DPIB_WRITE_PORT_UCHAR \
  CTL_CODE(DPIB_TYPE,  0xB10, METHOD_BUFFERED, FILE_WRITE_ACCESS)

#define IOCTL_DPIB_WRITE_PORT_USHORT \
  CTL_CODE(DPIB_TYPE,  0xB11, METHOD_BUFFERED, FILE_WRITE_ACCESS)

#define IOCTL_DPIB_WRITE_PORT_ULONG \
  CTL_CODE(DPIB_TYPE,  0xB12, METHOD_BUFFERED, FILE_WRITE_ACCESS)

typedef struct  _DPIB_WRITE_INPUT {
    ULONG   PortNumber;     // Port # to write to
    union   {               // Data to be output to port
        ULONG   LongData;
        USHORT  ShortData;
        UCHAR   CharData;
    };
}   DPIB_WRITE_INPUT;
```

Appendix D. Software Libraries Source Code

Appendix D includes the source code for the *hardware.h* and *sensor.hpp* libraries. These header files contain functions to access hardware device drivers and to control the prototype system components.

HARDWARE.H

```
/*********************************************************
* Module        : hardware.h
* Purpose       : A library of functions used with the Windows NT
*                 drivers developed in the SDA Lab.  Functions to
*                 open drivers, read and write ports and special
*                 functions for the PCI board are provided.
* Author        : Panos Arvanitis
* Date          : July 1996
* Version       : 1.0
* Revisions     :
*********************************************************/

//System includes
#include <windows.h>
#include <winioctl.h>
#include <stddef.h>

//Driver IOCTLs for each device driver
#include "\users\panos\progs\devdrv\morrph\morrph_ioctl.h"
#include "\users\panos\progs\devdrv\dpib\dpib_ioctl.h"
#include "\users\panos\progs\devdrv\pcidma2\ppcidma_ioctl.h"

//Function return codes
#define STATUS_SUCCESS 0
#define STATUS_FAILURE 255

//Path to driver (used to obtain handle)
#define STRING_MORRPHPATH     "\\\\.\\MorrphDev"
#define STRING_PCIDMAPATH     "\\\\.\\PPciDma0"
#define STRING_DIFFPAIRPATH     "\\\\.\\DpibDev"

//.POD filenames used to program boards
#define STRING_MORRPH_FILENAME        "MORRPH.POD"
#define STRING_PCIDMA_FILENAME        "PCIDMA.POD"
#define STRING_DIFFPAIR_FILENAME "DPIB.POD"

//Relative port addresses for Will's ISA interface
//Used for MORRPH and DPIB
#define ADDRESS_PORT    0x00
#define DATA_PORT       0x01
#define PROGRAM_PORT    0x02


/*************************************************************/
/*  AMCC Operation Register Offsets
*/
/*************************************************************/
#define  AMCC_OP_REG_OMB1       0x000
#define  AMCC_OP_REG_OMB2       0X004
#define  AMCC_OP_REG_OMB3       0X008
#define  AMCC_OP_REG_OMB4       0X00C
#define  AMCC_OP_REG_IMB1       0X010
#define  AMCC_OP_REG_IMB2       0X014
#define  AMCC_OP_REG_IMB3       0X018
#define  AMCC_OP_REG_IMB4       0X01C
#define  AMCC_OP_REG_FIFO       0X020
#define  AMCC_OP_REG_MWAR       0X024
#define  AMCC_OP_REG_MWTC       0X028
#define  AMCC_OP_REG_MRAR       0X02C
#define  AMCC_OP_REG_MRTC       0X030
#define  AMCC_OP_REG_MBEF       0X034
#define  AMCC_OP_REG_INTCSR     0X038
#define  AMCC_OP_REG_MCSR       0X03C
```

```
#define  AMCC_OP_REG_MCSR_NVDATA (AMCC_OP_REG_MCSR + 2) /* Data
in byte 2 */
#define  AMCC_OP_REG_MCSR_NVCMD  (AMCC_OP_REG_MCSR + 3) /*
Command in byte 3 */

//Structure used with WritePort functions to hold port and data
values
typedef struct _DRIVER_WRITE_INPUT {
        ULONG PortNumber;
        union {
                ULONG LongData;
                USHORT ShortData;
                UCHAR CharData;
        };
} DRIVER_WRITE_INPUT;



/*********************************************************
* OpenDriverHandle
*       Open a handle to a device driver.
*
* Arguments
*       hndFile         - handle to device driver object
*       DriverPath      - registry path to the device driver
*
* Return
*       STATUS_SUCCESS - Handle opened successfuly
*       STATUS_FAILURE - Error in opening handle
*********************************************************/
int OpenDriverHandle(HANDLE *hndFile, char *DriverPath)
{
        //Open the device driver
        *hndFile = CreateFile(DriverPath, GENERIC_WRITE |
GENERIC_READ, 0,
                NULL, OPEN_EXISTING, 0, NULL);

        //Driver not opened, return error
        if (*hndFile == INVALID_HANDLE_VALUE)
                return(STATUS_FAILURE);

        //Driver opened OK
        return(STATUS_SUCCESS);
}



/*********************************************************
* WritePort
*       Write a value to a port.  The handle to the device
* driver must be already open.
*
* Arguments
*       hndFile         - Handle to device driver
*       Port            - Relative port address to write to
*       Value           - Value to write
*       IoctlCode       - IoctlCode for operation
* Return
*       Standard Error Code
*********************************************************/
int WritePort(HANDLE hndFile, int Port, int Value, LONG
IoctlCode)
{
        DRIVER_WRITE_INPUT InputBuffer;             //buffer
passed to driver
        ULONG DataLength;
```

```
	BOOL IoctlResult;
	ULONG ReturnedLength;

	//Place port address and value in structure
	InputBuffer.PortNumber = (ULONG)Port;
	InputBuffer.CharData = (UCHAR)Value;

	//Determine size of data
	DataLength = offsetof(DRIVER_WRITE_INPUT, CharData) +

	sizeof(InputBuffer.CharData);

	//Send write request to device driver
	IoctlResult = DeviceIoControl(hndFile, IoctlCode,
&InputBuffer, DataLength,
                                  NULL, 0, &ReturnedLength,
NULL);

	if (IoctlResult)
		return(STATUS_SUCCESS);
	else
		return(STATUS_FAILURE);
}



/*******************************************************
* WritePortDouble
*	Write a value to a port.  The handle to the device
* driver must be already open.
*
* Arguments
*	 hndFile	- Handle to device driver
*	 Port		- Relative port address to write to
*	 Value		- Value to write
*	 IoctlCode	- IoctlCode for operation
* Return
*	 Standard Error Code
*******************************************************/
int WritePortDouble(HANDLE hndFile, int Port, ULONG Value, LONG
IoctlCode)
{
	DRIVER_WRITE_INPUT InputBuffer;
	ULONG DataLength;
	ULONG ReturnedLength;
	BOOL IoctlResult;

	//Place port address and value in structure
	InputBuffer.PortNumber = (ULONG)Port;
	InputBuffer.LongData = Value;

	//Determine data size
	DataLength = offsetof(DRIVER_WRITE_INPUT, LongData) +

	sizeof(InputBuffer.LongData);

	//Send write request to device driver
	IoctlResult = DeviceIoControl(hndFile, IoctlCode,
&InputBuffer,
                                      DataLength, NULL, 0,
&ReturnedLength, NULL);

	if (IoctlResult)
		return(STATUS_SUCCESS);
	else
		return(STATUS_FAILURE);
```

```
}



/*********************************************************
* ReadPort
*       Read a value from a port.  The handle to the device
* driver must be open.
*
* Arguments
*       hndFile    - handle to the device driver
*       Port       - port to read from
*       Value      - value returned from port
*       IoctlCode - IOCTL for port byte read
* Return
*       Standard Error Code
*********************************************************/
int ReadPort(HANDLE hndFile, int Port, UCHAR *Value, LONG
IoctlCode)
{
       BOOL IoctlResult;
       union {
              ULONG   LongData;
              USHORT  ShortData;
              UCHAR   CharData;
       } DataBuffer;
       ULONG DataLength;
       DWORD ReturnedLength;

       //Datermine data size
       DataLength = sizeof(DataBuffer.CharData);

       //Send read request to device driver
       IoctlResult = DeviceIoControl(hndFile, IoctlCode, &Port,
sizeof(Port),
                                          &DataBuffer,
DataLength, &ReturnedLength, NULL);

       if (IoctlResult) {
              //Place returned value in output if read was
successful
              *Value = DataBuffer.CharData;
              return(STATUS_SUCCESS);
       }
       else
              return(STATUS_FAILURE);
}



/*********************************************************
* ReadPortDouble
*       Read a double byte from a port.  The handle to the device
* driver must be already open.
*
* Arguments
*       hndFile         - Handle to device driver
*       Port            - Relative port address to write to
*       Value           - Value returned
*       IoctlCode       - IoctlCode for operation
*********************************************************/
int ReadPortDouble(HANDLE hndFile, int Port, ULONG *Value, LONG
IoctlCode)
{
       BOOL IoctlResult;
       union {
```

```
                ULONG   LongData;
                USHORT  ShortData;
                UCHAR   CharData;
        } DataBuffer;
        ULONG DataLength;
        DWORD ReturnedLength;

        //Determine data size
        DataLength = sizeof(DataBuffer.LongData);

        //Send rear request to device driver
        IoctlResult = DeviceIoControl(hndFile, (DWORD) IoctlCode,
&Port,
                                                sizeof(Port),
&DataBuffer, DataLength,
                                                &ReturnedLength,
NULL);

        if (IoctlResult) {
                //Place returned data in output, if read was
successful
                *Value = DataBuffer.LongData;
                return(STATUS_SUCCESS);
        }
        else
                return(STATUS_FAILURE);
}



/*******************************************************
* MapPciDmaBuffer
*       Map the PCI DMA buffer to the calling process address
* space.
*
* Arguments
*       hndFile         - Handle to device driver
*       VirtualAddress  - mapped DMA buffer address (in user
space)
*******************************************************/
int MapPciDmaBuffer(HANDLE hndFile, ULONG *VirtualAddress)
{
        BOOL IoctlResult;
        ULONG DataLength;
        ULONG DataBuffer;
        DWORD ReturnedLength;

        //Determine size of data
        DataLength = sizeof(DataBuffer);

        //Send map request to PCI device driver
        IoctlResult = DeviceIoControl(hndFile, (DWORD)
                IOCTL_PPCIDMA_MAP_USER_PHYSICAL_MEMORY,
                NULL, 0, &DataBuffer, DataLength, &ReturnedLength,
NULL);

        if (IoctlResult) {
                //Place virtual address in the output
                *VirtualAddress = DataBuffer;
                return(STATUS_SUCCESS);
        }
        else
                return(STATUS_FAILURE);
}
```

```
/*********************************************************
* UnMapPciDmaBuffer
*       Unmap the DMA buffer from the calling process address
* space.
*
* Arguments
*       hndFile         - Handle to device driver
*       VirtualAddress  - The mapped DMA address
*********************************************************/
int UnMapPciDmaBuffer(HANDLE hndFile, ULONG VirtualAddress)
{
        BOOL IoctlResult;
        ULONG InputBuffer;
        ULONG DataLength;
        ULONG ReturnedLength;

        //Place the virtual address in the structure
        InputBuffer = VirtualAddress;
        //Determine length of data
        DataLength = sizeof(VirtualAddress);

        //Send unmap request to PCI device driver
        IoctlResult = DeviceIoControl(hndFile,
                        (DWORD)
IOCTL_PPCIDMA_UNMAP_USER_PHYSICAL_MEMORY,
                        &InputBuffer, DataLength, NULL, 0,
&ReturnedLength, NULL);

        if (IoctlResult)
                return(STATUS_SUCCESS);
        else
                return(STATUS_FAILURE);
}



/*********************************************************
* GetPciDmaAddress
*   Get the physical address of the PCIDMA board.  Used to
* program PCIDMA address registers.
*
* Arguments
*       hndFile         - Handle to device driver
*       PhysicalAddress - Returned physical address
*********************************************************/
int GetPciDmaAddress(HANDLE hndFile, ULONG *PhysicalAddress)
{
        BOOL IoctlResult;
        ULONG DataLength;
        ULONG DataBuffer;
        DWORD ReturnedLength;

        //Determine size of buffer
        DataLength = sizeof(DataBuffer);     //size of input
buffer

#ifdef DEBUG
        printf("GetPciDmaAddress sending IOCTL %x.\n", (DWORD)
IOCTL_PPCIDMA_RETURN_MEMORY_INFORMATION);
#endif

        //Send request to PCIDMA device driver
        IoctlResult = DeviceIoControl(hndFile, (DWORD)
                IOCTL_PPCIDMA_RETURN_MEMORY_INFORMATION,
```

```
                NULL, 0, &DataBuffer, DataLength, &ReturnedLength,
NULL);

        if (IoctlResult) {
                //Return the physical address, if the request was
successful
                *PhysicalAddress = DataBuffer;
                return(STATUS_SUCCESS);
        }
        else
                return(STATUS_FAILURE);
}
```

SENSOR.HPP

```
/*********************************************************
* Module        : sensor.hpp
* Purpose       : A library of functions used in the FAA
*                 software for the AS&E System.  Function to
*                 initiliaze and access the hardware and sensors
*                 are provided.
* Author        : Panos Arvanitis
* Date          : January 1997
* Version       : 1.0
* Revisions     :
*********************************************************/

#include <conio.h>

//Constants used for DPIB ports
#define DPIB_MOTOR_PORT         0x06    //Conveyor motor port
#define DPIB_SENSOR_PORT        0x07    //Luggage sensor port
#define DPIB_MOTOR_FORWARD      0x01    //Conveyor move forward
command
#define DPIB_MOTOR_REVERSE      0x02    //Conveyor move reverse
command
#define DPIB_MOTOR_STOP         0x00    //Conveyor stop command
#define DPIB_SENSOR_FRONT       0x01    //Front sensor broken bit
#define DPIB_SENSOR_REAR        0x02    //Rear sensor broken bin

//Constants used to access the X-ray controller
#define XRAY_CONTROLLER_PORT    "COM2"  //Serial port for Ball
Controller
#define XRAY_CONTROLLER_BAUD    9600    //Baud rate
#define XRAY_CONTROLLER_BITS    8       //Data bits
//For these two constants, check the GetCommState help page, DCB
structure
#define XRAY_CONTROLLER_STOP    0       //1 Stop bit
#define XRAY_CONTROLLER_PARITY  0       //No Parity

//Constants used to access the filter motor controller
#define MOTOR_CONTROLLER_PORT   "COM1"  //Serial port for motor
controller
#define MOTOR_CONTROLLER_BAUD   1200    //Baud rate
#define MOTOR_CONTROLLER_BITS   8       //Data bits
//For these two constants, check the GetCommState help page, DCB
structure
#define MOTOR_CONTROLLER_STOP   0       //1 Stop Bit
#define MOTOR_CONTROLLER_PARITY 0       //No Parity
```

```
//
// Function     : WaitSeconds
// Purpose      : Wait the specified number of seconds.
// Arguments    :
//                SecWait = number of seconds to wait
// Return       : None
void WaitSeconds(int SecWait)
{
  //Store start time and current time
  DWORD BeginTime, CurrTime;

  BeginTime = GetTickCount();   //Current system tick count

  //Wait until the given number of clock ticks has occured
  //This is not the best way to time delay, but it works
  //and can be set-up easily, unlike a Windows timer.
  do {
    CurrTime = GetTickCount();
  } while ( CurrTime-BeginTime < SecWait * 1000);

}



//
// Function     : WaitTSeconds
// Purpose      : Wait the specified number of tenths of a
second.
// Arguments    :
//                TSecWait = number of tenths of second to wait
// Return       : None
void WaitTSeconds(int TSecWait)
{
  //Start time and current time
  DWORD BeginTime, CurrTime;

  //Get the current tick count
  BeginTime = GetTickCount();

  //Wait until the specified number of clock ticks has
  //occured
  do {
    CurrTime = GetTickCount();
  } while ( CurrTime-BeginTime < TSecWait * 100);
}



//
// Function     : MoveBeltForward
// Purpose      : Move the conveyor belt in the forward
direction.
// Arguments    :
//                hndFile = handle to the DPIB device driver
// Return       : None
void MoveBeltForward(HANDLE hndFile)
{
  int status;

  //Write the address to the motor controller port on the DPIB
  status = WritePort(hndFile, ADDRESS_PORT, DPIB_MOTOR_PORT,
                     IOCTL_DPIB_WRITE_PORT_UCHAR);

  //Set the move forward bit
  if (status == STATUS_SUCCESS)
    status = WritePort(hndFile, DATA_PORT, DPIB_MOTOR_FORWARD,
```

```
                              IOCTL_DPIB_WRITE_PORT_UCHAR);
  if (status != STATUS_SUCCESS)
    ReportFailWrite(hndFile);
}



//
// Function     : MoveBeltReverse
// Purpose      : Move the conveyor belt in the reverse
direction.
// Arguments    :
//                hndFile = handle to the DPIB device driver
// Return       : None
void MoveBeltReverse(HANDLE hndFile)
{
  int status;

  //Write the address of the motor controller port
  status = WritePort(hndFile, ADDRESS_PORT, DPIB_MOTOR_PORT,
                     IOCTL_DPIB_WRITE_PORT_UCHAR);

  //Set the move reverse bit
  if (status == STATUS_SUCCESS)
    status = WritePort(hndFile, DATA_PORT, DPIB_MOTOR_REVERSE,
                     IOCTL_DPIB_WRITE_PORT_UCHAR);

  if (status != STATUS_SUCCESS)
    ReportFailWrite(hndFile);
}
```

```
//
// Function     : StopBelt
// Purpose      : Stop the conveyor belt.
// Arguments    :
//                hndFile = handle to the DPIB device driver
// Return       : None
void StopBelt(HANDLE hndFile)
{
  int status;

  //Write the address of the motor controller port
  status = WritePort(hndFile, ADDRESS_PORT, DPIB_MOTOR_PORT,
                      IOCTL_DPIB_WRITE_PORT_UCHAR);

  //Clear all bits
  if (status == STATUS_SUCCESS)
       status = WritePort(hndFile, DATA_PORT, DPIB_MOTOR_STOP,
                               IOCTL_DPIB_WRITE_PORT_UCHAR);

  if (status != STATUS_SUCCESS)
       ReportFailWrite(hndFile);
}



//
// Function     : BreakFrontSensor
// Purpose      : Wait until the front luggage sensor is
interrupted.
// Arguments    :
//                hndFile = handle to the DPIB device driver
// Return       : None
void BreakFrontSensor(HANDLE hndFile)
```

```
{
  UCHAR sensorval;     //Sensor status
  int status;

  //Write the address of the sensor port on the DPIB
  status = WritePort(hndFile, ADDRESS_PORT, DPIB_SENSOR_PORT,
                     IOCTL_DPIB_WRITE_PORT_UCHAR);

  if (status != STATUS_SUCCESS)
       ReportFailWrite(hndFile);

  do {
    //Read sensor port
    //and wait until the bit is cleared
    status = ReadPort(hndFile, DATA_PORT, &sensorval,
                      IOCTL_DPIB_READ_PORT_UCHAR);
    if (status != STATUS_SUCCESS)
         ReportFailRead(hndFile);

  } while (sensorval & DPIB_SENSOR_FRONT);
}



//
// Function     : BreakRearSensor
// Purpose      : Wait until the rear luggage sensor is
interrupted.
// Arguments    :
//                hndFile = handle to the DPIB device driver
// Return       : None
void BreakRearSensor(HANDLE hndFile)
{
  UCHAR sensorval;
  int status;

  status = WritePort(hndFile, ADDRESS_PORT, DPIB_SENSOR_PORT,
                      IOCTL_DPIB_WRITE_PORT_UCHAR);
  if (status != STATUS_SUCCESS)
       ReportFailWrite(hndFile);

  do {  //Break rear beam
    status = ReadPort(hndFile, DATA_PORT, &sensorval,
                       IOCTL_DPIB_READ_PORT_UCHAR);
    if (status != STATUS_SUCCESS)
         ReportFailRead(hndFile);
  } while (sensorval & DPIB_SENSOR_REAR);
}



//
// Function     : UnBreakFrontSensor
// Purpose      : Wait until the front luggage sensor is cleared.
// Arguments    :
//                hndFile = handle to the DPIB device driver
// Return       : None
void UnBreakFrontSensor(HANDLE hndFile)
{
  UCHAR sensorval;
  int status;

  status = WritePort(hndFile, ADDRESS_PORT, DPIB_SENSOR_PORT,
                      IOCTL_DPIB_WRITE_PORT_UCHAR);

  if (status != STATUS_SUCCESS)
```

```
      ReportFailWrite;

  do {  //"un"break front beam
    status = ReadPort(hndFile, DATA_PORT, &sensorval,
                      IOCTL_DPIB_READ_PORT_UCHAR);
    if (status != STATUS_SUCCESS)
         ReportFailRead;
  } while (!(sensorval & DPIB_SENSOR_FRONT));
}



//
// Function     : UnBreakRearSensor
// Purpose      : Wait until the rear luggage sensor is cleared.
// Arguments    :
//                hndFile = handle to DPIB device driver
// Return       : None
void UnBreakRearSensor(HANDLE hndFile)
{
  UCHAR sensorval;
  int status;


  status = WritePort(hndFile, ADDRESS_PORT, DPIB_SENSOR_PORT,
                     IOCTL_DPIB_WRITE_PORT_UCHAR);
  if (status != STATUS_SUCCESS)
       ReportFailWrite;

  do {  //"un"break rear beam
    status = ReadPort(hndFile, DATA_PORT, &sensorval,
                      IOCTL_DPIB_READ_PORT_UCHAR);
    if (status != STATUS_SUCCESS)
         ReportFailRead;
  } while (!(sensorval & DPIB_SENSOR_REAR));
}



//
// Function     : SetKV75
// Purpose      : Set the X-ray voltage to 75KV.
// Arguments    :
//                hCom = handle to the serial port
// Return       : None
void SetKV75(HANDLE hCom)
{
  //This routine sends a command to the serial port controller
  //Although the baud rate is defined in the controller manual,
  //the controller is too slow to respond to consecutive
characters
  //sent at the defined baud rate.  Therefore a time delay is
  //inserted between each character to ensure correct receipt by
  //the x-ray controller.


  //Serial port buffer, contains character to be sent out
  char buffer;
  unsigned long BytesOut = 5;

  buffer = '!';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = 'V';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
```

```
  WaitTSeconds(5);

  buffer = '0';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '7';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '5';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = 0x0D;;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

}


//
// Function     : SetKV150
// Purpose      : Set the X-ray voltage to 150KV.
// Arguments    :
//                hCom = handle to the serial port
// Return       : None
void SetKV150(HANDLE hCom)
{
  //This routine sends a command to the serial port controller
  //Although the baud rate is defined in the controller manual,
  //the controller is too slow to respond to consecutive
characters
  //sent at the defined baud rate.  Therefore a time delay is
  //inserted between each character to ensure correct receipt by
  //the x-ray controller.


  char buffer;
  unsigned long BytesOut = 5;

  buffer = '!';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = 'V';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '1';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '5';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '0';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = 0x0D;;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

}
```

```
//
// Function     : SetmA300
// Purpose      : Set the X-ray current to 300mA.
// Arguments    :
//                hCom = handle to the serial port
// Return       : None
void SetmA300(HANDLE hCom)
{
  //This routine sends a command to the serial port controller
  //Although the baud rate is defined in the controller manual,
  //the controller is too slow to respond to consecutive
characters
  //sent at the defined baud rate.  Therefore a time delay is
  //inserted between each character to ensure correct receipt by
  //the x-ray controller.


  char buffer;
  unsigned long BytesOut = 5;

  buffer = '!';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = 'I';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '0';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '3';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = '0';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  buffer = 0x0D;;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

}



//
// Function     : TurnXrayON
// Purpose      : Turn the X-ray source on.
// Arguments    :
//                hCom = handle to the serial port
// Return       : None
void TurnXrayON(HANDLE hCom)
{
  //This routine sends a command to the serial port controller
  //Although the baud rate is defined in the controller manual,
  //the controller is too slow to respond to consecutive
characters
  //sent at the defined baud rate.  Therefore a time delay is
  //inserted between each character to ensure correct receipt by
  //the x-ray controller.
```

```
  char buffer;
  unsigned long BytesOut = 5;

  buffer = '!';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(8);

  buffer = 'X';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(8);

  buffer = 0x0D;;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(8);

}



//
// Function     : TurnXrayOFF
// Purpose      : Turn the X-ray source off.
// Arguments    :
//                hCom = handle to the serial port
// Return       : None
void TurnXrayOFF(HANDLE hCom)
{
  //This routine sends a command to the serial port controller
  //Although the baud rate is defined in the controller manual,
  //the controller is too slow to respond to consecutive
characters
  //sent at the defined baud rate.  Therefore a time delay is
  //inserted between each character to ensure correct receipt by
  //the x-ray controller.


  char buffer;
  unsigned long BytesOut = 5;

  buffer = '!';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(8);

  buffer = 'O';
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(8);

  buffer = 0x0D;;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(8);

}



//
// Function     : LowerFilter
// Purpose      : Lower the copper filter.
// Arguments    :
//                hCom = handle to the serial port
// Return       : None
void LowerFilter(HANDLE hCom)
{
  //A time delay is inserted between each character to
  //ensure proper receipt by the motor controller
```

```
  char buffer[10];
  unsigned long BytesOut = 5;

  strcpy(buffer, "0");
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
}



//
// Function     : RaiseFilter
// Purpose      : Raise the copper filter.
// Arguments    :
//                hCom = handle to the serial port.
// Return       : None
void RaiseFilter(HANDLE hCom)
{
  //A time delay is inserted between each character to
  //ensure proper receipt by the motor controller

  char buffer[10];
  unsigned long BytesOut = 5;

  strcpy(buffer, "75");
  WriteFile(hCom, &buffer, 3, &BytesOut, NULL);
  WaitTSeconds(5);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
}
```

```
//This function is maintained for compatibility with older
//console mode applications and should no longer be used.
int GetCorVal()
{
  int RegVal;

  cout << "Please enter new value: ";
  cin >> RegVal;
  cout << endl;

  return(RegVal);
}



//
// Function     : WriteRegister
// Purpose      : Write a value to a DPIB, not an ISA, port
// Arguments    :
//                RegNum  = port address
//                RegVal  = value to write
//                hndFile = handle to device driver
// Return       : None
void WriteRegister(UCHAR RegNum, int RegVal, HANDLE hndFile)
{
  int status;

  status = WritePort(hndFile, ADDRESS_PORT, RegNum,
                     IOCTL_DPIB_WRITE_PORT_UCHAR);
```

```
  if (status != STATUS_SUCCESS)
       ReportFailWrite;
  status = WritePort(hndFile, DATA_PORT, RegVal,
                       IOCTL_DPIB_WRITE_PORT_UCHAR);
  if (status != STATUS_SUCCESS)
         ReportFailRead;

}



//This function is maintained for compatibility with older
//console mode applications and should no longer be used.
BOOLEAN ShowMenu(int *LCorVal1, int *LCorVal2, int *LCorVal3,
                int *HCorVal1, int *HCorVal2, int *HCorVal3)
{
  char c;
  BOOLEAN ValidChoice;

  do {
    clrscr();
    ValidChoice = TRUE;
    cout << "          C. Continue Collection" << endl << endl;
    cout << "          1. Modify Low KV Offset Value 1 (Currently
" <<
        *LCorVal1 << ")" << endl;
    cout << "          2. Modify Low KV Offset Value 2 (Currently
" <<
        *LCorVal2 << ")" << endl;
    cout << "          3. Modify Low KV Offset Value 3 (Currently
" <<
      *LCorVal3 << ")" << endl << endl;
    cout << "          4. Modify Hi KV Offset Value 1 (Currently
" <<
      *HCorVal1 << ")" << endl;
    cout << "          5. Modify Hi KV Offset Value 2 (Currently
" <<
      *HCorVal2 << ")" << endl;
    cout << "          6. Modify Hi KV Offset Value 3 (Currently
" <<
      *HCorVal3 << ")" << endl << endl;

    cout << "          Q. Quit Program" << endl << endl;
    cout << "Please make a selection: " << endl;

    c = getch();
    switch (c) {
      case 'C', 'c'     :  return(TRUE);
      case '1'                  :  *LCorVal1 = GetCorVal();
                                        ValidChoice = FALSE;
                                        break;
      case '2'                  :  *LCorVal2 = GetCorVal();
                                        ValidChoice = FALSE;
                                        break;
      case '3'                  :  *LCorVal3 = GetCorVal();
                                        ValidChoice = FALSE;
                                        break;
      case '4'                  :       *HCorVal1 = GetCorVal();
                                        ValidChoice = FALSE;
                                        break;
      case '5'                  :       *HCorVal2 = GetCorVal();
                                        ValidChoice = FALSE;
                        break;
      case '6'                  :       *HCorVal3 = GetCorVal();
                                        ValidChoice = FALSE;
```

```
                    break;
      case 'Q', 'q'     :  return(FALSE);
      default                   :  ValidChoice = FALSE;
    }
  } while (!ValidChoice);

}



//
// Function    : SetUpCorrVal
// Purpose     : Program the data compensation values into the
DPIB
// Arguments   :
//               CVal1   = correction value for channel 1
//               CVal2   = correction value for channel 2
//               CVal3   = correction value for channel 3
//               hndFile = handle to DPIB
//
void SetUpCorrVal(int CVal1, int CVal2, int CVal3, HANDLE
hndFile)
{
  WriteRegister(1, CVal1, hndFile);
  WriteRegister(2, CVal2, hndFile);
  WriteRegister(3, CVal3, hndFile);
}



//
// Function     : ProgMotorController
// Purpose      : Program the filter motor controller.
// Arguments    :
//                hCom = handle to the serial port
// Return       : TRUE  = motor controller programmed
successfully
//                FALSE = failed to program motor controller
BOOLEAN ProgMotorController(HANDLE hCom)
{
  //A time delay is inserted between each character to
  //ensure proper receipt by the motor controller

  char buffer[25];
  ULONG BytesOut;

  strcpy(buffer, "&");
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);

  strcpy(buffer, "E");
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);

  strcpy(buffer, "90 V1=800");
  WriteFile(hCom, &buffer, 9, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  strcpy(buffer, "95 R1=3");
  WriteFile(hCom, &buffer, 7, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
```

```
  WaitTSeconds(5);

  strcpy(buffer, "100 C1=0.023");
  WriteFile(hCom, &buffer, 12, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  strcpy(buffer, "110 INPUT A1");
  WriteFile(hCom, &buffer, 12, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  strcpy(buffer, "120 @:GOTO 110");
  WriteFile(hCom, &buffer, 14, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);

  strcpy(buffer, "RUN90");
  WriteFile(hCom, &buffer, 5, &BytesOut, NULL);
  buffer[0] = 0x0D;
  WriteFile(hCom, &buffer, 1, &BytesOut, NULL);
  WaitTSeconds(5);
}



//
// Function     : SetUpXrayController
// Purpose      : Configure the X-ray controller serial port.
// Arguments    :
//                hCom = returns handle to the serial port
// Return       : TRUE  = serial port configured properly.
//              : FALSE = failed to configure port
BOOLEAN SetUpXrayController(HANDLE *hCom)
{
  DCB dcb1;

  //Open a handle to the COM port
  *hCom = CreateFile(XRAY_CONTROLLER_PORT, GENERIC_READ |
GENERIC_WRITE,
0, NULL, OPEN_EXISTING, 0, NULL);

  if (*hCom == INVALID_HANDLE_VALUE)
    return FALSE;

  //Get the DCB structure
  if (!GetCommState(*hCom, &dcb1))
    return FALSE;

  //Enter configuration values in the DCB structure
  dcb1.BaudRate = XRAY_CONTROLLER_BAUD;
  dcb1.ByteSize = XRAY_CONTROLLER_BITS;
  dcb1.Parity   = XRAY_CONTROLLER_PARITY;
  dcb1.StopBits = XRAY_CONTROLLER_STOP;

  //Configure the serial port
  if (!SetCommState(*hCom, &dcb1))
    return FALSE;

  return TRUE;
}
```

```
//
// Function     : SetUpDPIB
// Purpose      : Set-up the initial port values for the DPIB.
// Arguments    :
//                hndDpib = returns a handle to the DPIB
// Return       : TRUE  = DPIB ports initialized successfully
//                FALSE = Failed to initialize DPIB ports
BOOLEAN SetUpDPIB(HANDLE *hndDpib)
{
  int status;

  status = OpenDriverHandle(hndDpib, STRING_DIFFPAIRPATH);
  if (status == STATUS_SUCCESS)
  {
    //Setup the timing values for the DPIB
    /*WriteRegister(6, 0x070, hndDpib);
    WriteRegister(7, 0x06C, hndDpib);
    WriteRegister(8, 0x070, hndDpib);
    WriteRegister(9, 0x050, hndDpib);
    WriteRegister(10, 0x052, hndDpib);
    WriteRegister(11, 0x05E, hndDpib);
    WriteRegister(12, 0x05F, hndDpib);
    WriteRegister(13, 0x059, hndDpib);
    WriteRegister(14, 0x070, hndDpib);*/
    WriteRegister(8, 0x01, hndDpib);
    WriteRegister(9, 0x0C2, hndDpib);
    WriteRegister(10, 0x00, hndDpib);
    WriteRegister(11, 0x0E1, hndDpib);
    WriteRegister(12, 0x00, hndDpib);
    WriteRegister(13, 0x0E1, hndDpib);

    return TRUE;
  }
  else
    return FALSE;
}



//
// Function     : SetUpMotorController
// Purpose      : Configure the motor controller serial port.
// Arguments    :
//                hCom = returns a handle to the serial port
// Return       : TRUE  = Serial port configured properly
//                FALSE = failed to configure the serial port
BOOLEAN SetUpMotorController(HANDLE *hCom)
{
  DCB dcb1;

  //Open a handle to the serial port
  *hCom = CreateFile(MOTOR_CONTROLLER_PORT, GENERIC_READ |
GENERIC_WRITE,
0, NULL, OPEN_EXISTING, 0, NULL);

  if (hCom == INVALID_HANDLE_VALUE)
    return FALSE;

  //Get the DCB structure
  if (!GetCommState(*hCom, &dcb1))
    return FALSE;

  //Load the DCB structure with the serial port operating
parameters
  dcb1.BaudRate = MOTOR_CONTROLLER_BAUD;
```

```
  dcb1.ByteSize = MOTOR_CONTROLLER_BITS;
  dcb1.Parity   = MOTOR_CONTROLLER_PARITY;
  dcb1.StopBits = MOTOR_CONTROLLER_STOP;

  //Configure the serial port
  if (!SetCommState(*hCom, &dcb1))
    return FALSE;

  return TRUE;
}



//This function is used to report error conditions with a console
//application.  It is maintained for backwards compatibility only
//and should no longer be used.
void ReportFailWrite(HANDLE hndFile)
{
  cout << endl << "ERROR: Failed to write to DPIB driver." <<
endl;
  cout << "Program will now terminate." << endl;
  CloseHandle(hndFile);
  exit(255);
}



//This function is used to report error conditions with a console
//application.  It is maintained for backwards compatibility only
//and should no longer be used.
void ReportFailRead(HANDLE hndFile)
{
  cout << endl << "ERROR: Failed to read from DPIB driver." <<
endl;
  cout << "Program will now terminate." << endl;
  CloseHandle(hndFile);
  exit(255);
}
```

Appendix E. Utilities and GUI Source Code

Appendix E includes the source code for the following utilities: *progall, colpul-silent* and *edisp*. Also included is the source code for *Galaxie*, the graphical user interface. *Galaxie* is a Borland C++ 5.0 project and contains resource files which are not shown in this appendix.

PROGALL.CPP

```
/*********************************************************
* Program : progall.c
*
* Version : See progall.hpp
*
* Author  : Panos Arvanitis
*
* Date    : August 10, 1996
*
* Purpose : Program all the boards for the Stuart flooring
*           project.  This includes MORRPH, PCIDMA, DPIB.
*
* Options :
*       -s              : Silent mode (no output)
*       -h              : Display help
*       -m <filename>   : Alternate filename for MORRPH
*                         (default is MORRPH.POD)
*       -p <filename>   : Alternate filename for PCIDMA
*                         (default is PCIDMA.POD)
*       -d <filename>   : Alternate filename for DPIB
*                         (default is DPIB.POD)
* Return  :
*       0 : Success
*/
/************************************
****
**** TODO:
****           Fix Program_PciDma (and maybe others)
****           so that if anything fails, you still close
****           the handle to the device.
****
*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

//My libraries
#include "\users\panos\lib\hardware.h"  //Access to ports through
                                        // device drivers
#include "progall.hpp"                      //Program specific
header


int Program_PciDma();
int Program_DiffPair();
int ProgCol(HANDLE hndFile, FILE *podfile, int column,
              LONG IoctlCode);

int main(int argc, char *argv[])
{
        int status;            //error code to return

#ifdef DEBUG
        GetLocalTime(&TimeStart);
#endif

        printf("ProgAll V%s\n", PROGRAM_VERSION);

#ifdef DEBUG
        printf("\n------- DEBUG VERSION - DO NOT RELEASE -------
\n\n");
#endif
/*char
```

```
	int numoptions;		//number of command line
options

	if (argc != 0) //have some command line options
		for(numoptions=1; numoptions = argc; numoptions++)
			switch
*/
	//Program_Morrph();
	printf("Begin programming PCIDMA board.\n");
	status = Program_PciDma();
	if (status != STATUS_SUCCESS) {
		printf("ERROR : Failed to program PCIDMA.\n");
		return(status);
	}
	printf("PCIDMA programmed succesfully.\n\n");

	printf("Begin programming DPIB board.\n");
	status = Program_DiffPair();
	if (status != STATUS_SUCCESS) {
		printf("ERROR : Unable to program PDPIB.\n");
		return(status);
	}
	printf("DPIB programmed succesfully.\n");

#ifdef DEBUG
	GetLocalTime(&TimeEnd);

	printf("Program started at  %d:%d.%d.\n",
TimeStart.wMinute,
		TimeStart.wSecond, TimeStart.wMilliseconds);

	printf("Program started at  %d:%d.%d.\n",
TimeEnd.wMinute,
		TimeEnd.wSecond, TimeEnd.wMilliseconds);

#endif

	return(status);
}


/*********************************************************
* Program_PciDma
*	Program the PciDma board
*/
int Program_PciDma()
{
  int status;
  char a[3], b[5], byteval;
  ULONG DValue = 0;
  HANDLE hndFile;
  FILE *podfile;

  status = OpenDriverHandle(&hndFile, STRING_PCIDMAPATH);
  if (status != STATUS_SUCCESS) {
	printf("ERROR : Unable to get handle to PCIDMA
driver.\n");
	 return(STATUS_HANDLE_OPEN_FAIL);
  }


  //PCI Initiated, FIFO Bus Mastering Setup
  //PRE-RELEASE : Do I need to set this up?????
  status = WritePortDouble(hndFile, AMCC_OP_REG_INTCSR,
		0x000000000, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
  if (status != STATUS_SUCCESS) {
```

```
        printf("ERROR : WritePort failed to PCI DMA board.\n");
        return(status);
  }

  status = WritePortDouble(hndFile, AMCC_OP_REG_MCSR,
                0x00E007400, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
  if (status != STATUS_SUCCESS) {
        printf("ERROR : WritePort failed to PCI DMA board.\n");
        return(status);
  }

  //Clear FIFO
  status = WritePortDouble(hndFile, AMCC_OP_REG_MCSR,
                0x00E007400, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
  if (status != STATUS_SUCCESS) {
        printf("ERROR : WritePort failed to PCI DMA board.\n");
        return(status);
  }

  podfile = fopen(STRING_PCIDMA_FILENAME, "r");
  if (podfile == NULL) {
        printf("ERROR : Unable to open %s.\n",
STRING_PCIDMA_FILENAME);
        return(STATUS_FAILURE);
  }

  while (fgets(a, 3, podfile) != NULL) {
        strcpy(b, "0x");
        strncat(b, a, 3);
        byteval = strtol(b, NULL, 0);

        WritePort(hndFile, AMCC_OP_REG_FIFO, byteval,
                IOCTL_PPCIDMA_WRITE_PORT_UCHAR);

        //Wait for empty PCI to ADD-ON FIFO
        do {

                status = ReadPortDouble(hndFile,
AMCC_OP_REG_MCSR, &DValue,

       IOCTL_PPCIDMA_READ_PORT_ULONG);
                if (status != STATUS_SUCCESS) {
                        printf("ERROR : Read from AMCC FIFO
register failed.\n");
                        return(status);
                }
                byteval = DValue & 0x004;
        } while (byteval == 0);

  }    //while

  fclose(podfile);

  status = WritePort(hndFile, AMCC_OP_REG_FIFO, 0x0FF,
                        IOCTL_PPCIDMA_WRITE_PORT_UCHAR);
  if (status != STATUS_SUCCESS) {
        printf("ERROR : Write to AMCC OP REG failed.\n");
        return(status);
  }

  status = WritePort(hndFile, AMCC_OP_REG_FIFO, 0x0FF,
                        IOCTL_PPCIDMA_WRITE_PORT_UCHAR);
  if (status != STATUS_SUCCESS) {
        printf("ERROR : Write to AMCC OP REG failed.\n");
        return(status);
  }
```

```
  if (!CloseHandle(hndFile)) {
          printf("ERROR : Failed to close device handle for
PCIDMA.\n");
          return(STATUS_FAILURE);
  }

  return(STATUS_SUCCESS);

}       //Program_PciDma




/*******************************************************
* Program_DiffPair
*
*       Program the differential pair board.  Currently only
* programs the first DMA board.
*
*/
int Program_DiffPair()
{
  int status;                 //error code to return
  HANDLE hndFile;             //handle to driver
  FILE *podfile;

  status = OpenDriverHandle(&hndFile, STRING_DIFFPAIRPATH);

  if (status != STATUS_SUCCESS) {
          printf("ERROR : Unable to get handle to DPIB
driver.\n");
          return(STATUS_FAILURE);
  }

  podfile  = fopen(STRING_DIFFPAIR_FILENAME, "r");
  if (podfile == NULL) {
          printf("ERROR: Unable to open %s.\n",
                          STRING_DIFFPAIR_FILENAME);
          return(STATUS_FAILURE);
  }

  //column = 0, 1, or 2
  status = ProgCol(hndFile, podfile, 0,
                          IOCTL_DPIB_WRITE_PORT_UCHAR);

  if (!CloseHandle(hndFile)) {
          printf("ERROR : Failed to close device handle for
DPIB.\n");
          return(STATUS_FAILURE);
  }

  fclose(podfile);

  return(status);
}




/*******************************************************
* ProgCol
*       Program a column
*
```

```
* Arguments
*       hndFile         - handle to driver "file"
*       podfile         - pointer to .POD file
*   column              - column to program (must be 0, 1, or 2)
*       IoctlCode       - code to write to device driver
*/
int ProgCol(HANDLE hndFile, FILE *podfile, int column,
                                        LONG IoctlCode)
{
  char a[3], b[5], byteval, i;       //Don't try to use column
                                //instead of i
  int status;


#ifdef DEBUG
  BOOL DispVals = TRUE;
#endif

  switch (column) {
  case 0       :      i = 1;
                      break;
  case 1       :      i = 2;
                      break;
  case 2       :      i = 4;
                      break;
  } //switch


  status = WritePort(hndFile, ADDRESS_PORT, i,
                              IoctlCode);
  if (status != STATUS_SUCCESS) {
         printf("ERROR : WritePort to address with column
failed.\n");
         return(status);
  }

  Sleep(10);

  status = WritePort(hndFile, PROGRAM_PORT, i,
                              IoctlCode);
  if (status != STATUS_SUCCESS) {
         printf("ERROR : WritePort to program with column
failed.\n");
         return(status);
  }

  Sleep(10);

  //Set PROGRAM pin low
  status = WritePort(hndFile, ADDRESS_PORT, column * 32,
                              IoctlCode);
  if (status != STATUS_SUCCESS) {
         printf("ERROR : WritePort with re-program failed.\n");
         return(status);
  }

  Sleep(10);

  while (fgets(a, 3, podfile) != NULL) {     //loop to program
Xilinx
         strcpy(b, "0x");     //to convert from hex
         strncat(b, a, 3);    //append hex value read in
         byteval = strtol(b, NULL, 0);
         status = WritePort(hndFile, DATA_PORT, byteval,
IoctlCode);
```

```
        if (status != STATUS_SUCCESS) {
                printf("ERROR : WritePort failed.\n");
                return(status);
        }

  } //while

#ifdef DEBUG
  printf("Last byte written %x.\n", byteval);
#endif

  //Write an extra byte to the Xilinx
  status = WritePort(hndFile, DATA_PORT, byteval, IoctlCode);
  if (status != STATUS_SUCCESS) {
        printf("ERROR: WritePort with extra byte failed.\n");
        return(status);
  }

  return(STATUS_SUCCESS);
}
```

PROGALL.HPP

```
//Program version
#define PROGRAM_VERSION "1.00a"

//Defines for returned error codes.
#define STATUS_IOCTLFAILED      10
#define STATUS_FILE_OPEN_FAIL   20
#define STATUS_HANDLE_OPEN_FAIL 30
```

COLPUL-SILENT.C

```
/*********************************************************
* ColPul
*       A port of the PciDma collection utility for Windows NT
*
* Things to do:
*       - Update the configuration file to include information
for all
*         six channels.  Then compute memory size for each
channel.
*/

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
//#include <conio.h> Maybe don't need?

#include "\users\panos\progs\lib\hardware.h"

//#define DEBUG
#define CONFIGURATION_FILENAME       "PCIDMA.CFG"
#define HIGH_BYTE(ax)   (ax >> 8)
#define LOW_BYTE(ax)    (ax & 0x0FF)

int Read_Configuration(int *numframes, int *width1, int *width2,
int *width3, int *startpix, int *channel_enable);
void save_image(unsigned char *image, int numchan, int width, int
numframes, FILE *out1, FILE *sfile);
int read_timer(int subtimer, int maintimer, HANDLE hndFile);
void program_timer(int numframes, int width, int startpix, int
timer, HANDLE hndFile);
```

```
void main ()
{
       unsigned long dma_addr, sec_addr, tri_addr, me_addr,
bob_addr, JIM_addr;
    DWORD write_data[29];
       int i, width, startpix, read_numframes[2][4];
       int width1, width2, width3;
   int numframes;
   int memsize1, memsize2, memsize3;
       BYTE j;
       DWORD box;
   int channel_enable;

       unsigned char *dmaptr, *secptr, *triptr, *meptr, *bobptr,
*jimptr;
       FILE *outfile1, *outfile2, *outfile3;
       FILE *outfile4, *outfile5, *outfile6, *outfile7,
*outfile8, *outfile9;

       int status;
       HANDLE hndFile;
       ULONG PhysicalAddress;
       ULONG VirtualAddress;

       //Fix
       width = width3;

       outfile5 = fopen("sixchan.dat","wb");

       outfile6 = fopen("four.img","wb");
       outfile7 = fopen("five.img","wb");
       outfile8 = fopen("six.img","wb");
       outfile9 = fopen("strong.dat","wb");

       outfile1 = fopen("one.img","wb");
       outfile2 = fopen("two.img","wb");
       outfile3 = fopen("three.img","wb");
       outfile4 = fopen("string.dat","wb");

       status = Read_Configuration(&numframes, &width1, &width2,
&width3, &startpix, &channel_enable);
       if (status != STATUS_SUCCESS) {
              printf("ERROR : Failed  to read configuration
file.\n");
              return(STATUS_FAILURE);
       }

       width = width3;


#ifdef DEBUG
       printf("----------- DEBUG VERSION - DO NOT RELEASE ------
----\n");
       printf("Collection configuration (PCIDMA.CFG):\n");
       printf("Numframes = %d\nWidth1 = %d\nWidth2 = %d\nWidth3
= %d\n", numframes, width1, width2, width3);
       printf("Startpix = %d\nChannel_enable = %d\n", startpix,
channel_enable);
       printf("End of read configuration data.\n");
#endif


       status = OpenDriverHandle(&hndFile, STRING_PCIDMAPATH);
       if (status != STATUS_SUCCESS) {
```

```
                printf("ERROR : Failed to open driver handle.\n");
                return(status);
        }

        status = GetPciDmaAddress(hndFile, &PhysicalAddress);
        if (status != STATUS_SUCCESS) {
                printf("ERROR : Could not get physical
address.\n");
                goto finish2;
        }

        //printf("Buffer located at physical address %x.\n",
PhysicalAddress);

        status = MapPciDmaBuffer(hndFile, &VirtualAddress);
        if (status != STATUS_SUCCESS) {
                printf("ERROR : Mapping failed.\n");
                goto finish1;
        }

        //printf("Buffer mapped at virtual address %x.\n",
VirtualAddress);

//fix
        memsize1 = (numframes * width1) + 50000; /* allocate
memory */
   memsize2 = (numframes * width2) + 50000;
   memsize3 = (numframes * width3) + 50000;

        dma_addr = PhysicalAddress;
        sec_addr = dma_addr + memsize1;
        tri_addr = sec_addr + memsize2;
        me_addr  = tri_addr + memsize3;
        bob_addr = me_addr  + memsize3;
        JIM_addr = bob_addr + memsize3;

        dmaptr = (PUCHAR) VirtualAddress;
        secptr = dmaptr + memsize1;
        triptr = secptr + memsize2;
        meptr  = triptr + memsize3;
        bobptr = meptr  + memsize3;
        jimptr = bobptr + memsize3;

/*
        memset(dmaptr, 0x0A, memsize1);
        memset(secptr, 0x0B, memsize1);
        memset(triptr, 0x0C, memsize1);
        memset(meptr, 0x0D, memsize1);
        memset(bobptr, 0x0E, memsize1);
        memset(jimptr, 0x0F, memsize1);
*/

        /*PCI Initiated, FIFO Bus Mastering Setup */
        WritePortDouble(hndFile, AMCC_OP_REG_INTCSR, 0x000000000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        /* Have to write from "on-board" to set board interrupts
*/
        WritePortDouble(hndFile, AMCC_OP_REG_MCSR, 0x00F000700,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 0x00e007400
        WritePortDouble(hndFile, AMCC_OP_REG_MWAR, 0x03377AAEE,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        WritePortDouble(hndFile, AMCC_OP_REG_MRAR, 0x000000000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        WritePortDouble(hndFile, AMCC_OP_REG_MWTC, 0x0000000FF,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
```

```
       WritePortDouble(hndFile, AMCC_OP_REG_MRTC, 0x000000000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

       WritePortDouble(hndFile, AMCC_OP_REG_MCSR, 0x00E000700,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 0x00e007400

/*************************************************************
********/
/*  SEND COMMAND
*/
/*  0x080000000 resets the flip-flops in the Xilinx
*/
/*************************************************************
********/
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x080000000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
       //printf("Reset Flip-flops!\n");

       /* Data for MAIL_ID command */
       write_data[0]=0x01F000000;    // Call for ID in Mailbox
(proof that Xilinx programmed correctly)
       write_data[1]=0x01E000000;    // Setup AMCC DMA transfer
- 3 steps
       write_data[6]=0x01D000000;    // Load address for DMA
into AMCC
       write_data[2]=((dma_addr&0x0FFFF)|0x000000000);  // low
low byte for Xilinx
       write_data[3]=(((dma_addr>>16)&0x0FFFF)|0x001000000); //
low high byte for Xilinx
       write_data[4]=((sec_addr&0x0FFFF)|0x002000000); // high
low byte for Xilinx
       write_data[5]=(((sec_addr>>16)&0x0FFFF)|0x003000000); //
high high byte for Xilinx
    write_data[7]=0x01C000003;    // Enable Transfer and Collect
       write_data[8]=0x01C000000;    // Disable Transfer and
Collect
       write_data[14]=0x019000000;   // Write to Dual Port Ram -
Buffer 0 (TEST - 0x019110000)
       write_data[15]=0x019100000;   // Clear write (for DP Ram)
- Buffer 0
       write_data[16]=0x019330000;   // Write to Dual Port Ram -
Buffer 1
       write_data[17]=0x019320000;   // Clear write (for DP Ram)
- Buffer 1
       write_data[9]=((tri_addr&0x0FFFF)|0x004000000); // high
low byte for Xilinx
       write_data[10]=(((tri_addr>>16)&0x0FFFF)|0x005000000); //
high high byte for Xilinx
       write_data[18]=0x019340000;
       write_data[11]=((me_addr&0x0FFFF)|0x006000000); // high
low byte for Xilinx
       write_data[12]=(((me_addr>>16)&0x0FFFF)|0x007000000); //
high high byte for Xilinx
       write_data[19]=((bob_addr&0x0FFFF)|0x008000000); // high
low byte for Xilinx
       write_data[20]=(((bob_addr>>16)&0x0FFFF)|0x009000000); //
high high byte for Xilinx
       write_data[21]=((JIM_addr&0x0FFFF)|0x00a000000); // high
low byte for Xilinx
       write_data[22]=(((JIM_addr>>16)&0x0FFFF)|0x00b000000); //
high high byte for Xilinx
       write_data[23]=0x019160000;
       write_data[24]=0x019180000;
       write_data[25]=0x0191A0000;
    write_data[26]=0x01C000004; // Gate for the timers - disable
with [8]
```

```
    write_data[27]=0x018000000; // Clock for the timers

               /* Attempt to retrieve Internal Board ID */
                //printf("               data_written =
%lx\n",write_data[0]);
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[0],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

               /* Check to see if BOARD is responding */
       box = 0x0000;
       i=0;

       do {
               ReadPortDouble(hndFile, AMCC_OP_REG_MBEF, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
               box = box & 0x0000F0000;
               //if (box != 0) printf("  Written to
successfully!\n");
               i++;
       } while (box!=0x0000F0000 && i<800);

       /*printf(" LOCATION = %lx       READ VALUE = %lx, i=%d
\n", ioaddr + 0x034, box, i);
    getchar();*/

       if (box == 0)
               printf("\n\nMailbox Not Written
To!!!!!!!!!!!!!!!!!!!!\n\n\n");

       /* Is the data correct? */
       ReadPortDouble(hndFile, AMCC_OP_REG_IMB1, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);

       if((box&0x0FFFF0000) != 0x0A1E90000)
               printf("Incorrectly READ VALUE = %lx \n", box);
       //else
               //printf("        READ VALUE = %lx \n", box);

       Sleep(1);
       /* Start AMCC Setup Routine on board */
       //printf("    Setup data_written = %lx\n",write_data[1]);
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[1],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
       Sleep(1);

    /* Load BUFFER 0 low low DMA address in Xilinx */
       //printf("               data_written =
%lx\n",write_data[2]);
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[2],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
       Sleep(1);
    /* Load BUFFER 0 low high DMA address in Xilinx */
       //printf("               data_written =
%lx\n",write_data[3]);
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[3],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

       /* Load BUFFER 1 low low DMA address in Xilinx */
       //printf("               data_written =
%lx\n",write_data[4]);
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[4],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
       Sleep(1);
       /* Load BUFFER 1 low high DMA address in Xilinx */
```

```
        //printf("                data_written =
%lx\n",write_data[5]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[5],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        /* Load BUFFER 2 low low DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[9]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[9],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        Sleep(1);
        /* Load BUFFER 2 low high DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[10]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[10], IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        /* Load BUFFER 3 low low DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[11]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[11], IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        Sleep(1);
        /* Load BUFFER 3 low high DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[12]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[12], IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        /* Load BUFFER 4 low low DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[19]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[19], IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        Sleep(1);
        /* Load BUFFER 4 low high DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[20]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[20], IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        /* Load BUFFER 5 low low DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[21]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[21], IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        Sleep(1);
        /* Load BUFFER 5 low high DMA address in Xilinx */
        //printf("                data_written =
%lx\n",write_data[22]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[22], IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        Sleep(1);

        //printf("        Switch address to buffer 0 -  %lx\n",
write_data[17]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[17], IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        //getchar();

        /* Load the First address into the AMCC */
        //printf("Load the first address = %lx\n",write_data[6]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[6],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
```

```
        Sleep(1);

        /* Read the write address for the DMA transfer */

        ReadPortDouble(hndFile, AMCC_OP_REG_MWAR, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
        //printf("AMCC WRITE ADDRESS BUFFER 1!!! (%lx) \n", box);
        if (box!=sec_addr){
                printf(" WRONG WRITE ADDRESS!!! (%lx) \n",
sec_addr);
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[8], IOCTL_PPCIDMA_WRITE_PORT_ULONG); //stop transfer
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[14], IOCTL_PPCIDMA_WRITE_PORT_ULONG); // Switch back
to buffer 0 if not already
               // exit(0);
        }



        /* Program Timer 0 for pulnix camera */
    program_timer(numframes+1, numframes+1, numframes+1, 0,
hndFile);
    program_timer(numframes+1, numframes+1, numframes+1, 1,
hndFile);

    /* Clock and Gate the new count values into the timers */
    //printf("Enable Gate\n");
    WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[26],
IOCTL_PPCIDMA_WRITE_PORT_ULONG); //gate=1

    //printf("Enable CLOCK for Timers\n");
    WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[27],
IOCTL_PPCIDMA_WRITE_PORT_ULONG); //clock pulsed

    //printf("Disable Gate\n");
    WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[8],
IOCTL_PPCIDMA_WRITE_PORT_ULONG); //gate=0

        /* Read back the numframes programmed */
    for(i=0; i<2; i++){
               for(j=1; j<4; j++){
                       read_numframes[i][j] = read_timer(j, i,
hndFile);  // '3' is the third subtimer( of timer 0) which is
numframes
            //printf("read numframes = %d [i,j](%d,%d)\n",
read_numframes, i, j);
        }
    }

    //printf("        Which channels should be
enabled?(7=3enabled, 63=6enabled)\n");
    //scanf("%d", &channel_enable);
    //printf("        Enable Channels -  %lx\n",
(0x01B000000|channel_enable));
    WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
(0x01B000000|channel_enable), IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        //printf("        Switch out of test mode -  %lx\n",
write_data[14]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[14], IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        //getchar();

        /* Enable Transfer and Collect enables */
        //printf("        Enable Transfer -  %lx\n",
write_data[7]);
```

```
       WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[7],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

       /* Check for end of transfer via polling */
       box = 0x0000;
       i=0;

       do {
              ReadPortDouble(hndFile, AMCC_OP_REG_MBEF, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
              box = box & 0x0000F0000;
              i++;
    } while (box!=0x0000F0000 && i<300000);

    /* Disable collection and transfer */
    WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[8],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
/*
    if(box!=0){
              printf(" LOCATION = %lx       READ VALUE = %lx, i
= %d \n", 0x034, box, i);
        printf(" Interrupt occurred!!\n");
    }

    if(box==0){
        printf("Interrupt didn't occur!!!  Press a key to
continue.\n");
        getchar();
        printf(" LOCATION = %lx       READ VALUE = %lx, i = %d
\n", 0x034, box, i);
    }
*/
```

```
#ifdef DEBUG
        printf("Reading Timer values");
#endif
    /* Read back the numframes programmed */
    read_numframes[0][3] = numframes - read_timer(3, 0, hndFile);
// '3' is the third subtimer( of timer 0) which is numframes
        if(read_numframes[0][3]<1){ read_numframes[0][3]=10;}
    //printf("Numframes (Timer0, subtimer2)= %d\n",
read_numframes);
    //getchar();
    read_numframes[0][2] = numframes - read_timer(2, 0, hndFile);
// '3' is the third subtimer( of timer 0) which is numframes
        if(read_numframes[0][2]<1){ read_numframes[0][2]=10;}
    //printf("Numframes (Timer0, subtimer1) = %d\n",
read_numframes);
    //getchar();
    read_numframes[0][1] = numframes - read_timer(1, 0, hndFile);
// '3' is the third subtimer( of timer 0) which is numframes
        if(read_numframes[0][1]<1){ read_numframes[0][1]=10;}
    //printf("Numframes (Timer0, subtimer0) = %d\n",
read_numframes);
    //getchar();

        read_numframes[1][3] = numframes - read_timer(3, 1,
hndFile);  // '3' is the third subtimer( of timer 0) which is
numframes
        if(read_numframes[1][3]<1){ read_numframes[1][3]=10;}
    //printf("Numframes (Timer1, subtimer2)= %d\n",
read_numframes);
    //getchar();
    read_numframes[1][2] = numframes - read_timer(2, 1, hndFile);
// '3' is the third subtimer( of timer 0) which is numframes
```

```
        if(read_numframes[1][2]<1){ read_numframes[1][2]=10;}
    //printf("Numframes (Timer1, subtimer1)= %d\n",
read_numframes);
    //getchar();
    read_numframes[1][1] = numframes - read_timer(1, 1, hndFile);
// '3' is the third subtimer( of timer 0) which is numframes
        if(read_numframes[1][1]<1){ read_numframes[1][1]=10;}
    //printf("Numframes (Timer1, subtimer0)= %d\n",
read_numframes);
    //getchar();



        save_image(dmaptr, 1, width1 , read_numframes[0][1],
outfile1, outfile4);
        save_image(secptr, 1, width2 , read_numframes[0][2],
outfile2, outfile5);
        save_image(triptr, 1, width3 , read_numframes[0][3],
outfile3, outfile4);
        save_image(meptr, 1, width1 , read_numframes[1][1],
outfile6, outfile4);
        save_image(bobptr, 1, width1 , read_numframes[1][2],
outfile7, outfile9);
        save_image(jimptr, 1, width1 , read_numframes[1][3],
outfile8, outfile4);




        //printf("ID data_written = %lx\n",write_data[0]);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[0],
IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        Sleep(1);

        ReadPortDouble(hndFile, AMCC_OP_REG_MBEF, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
        //printf("        MAILBOX = %lx \n", box);
        ReadPortDouble(hndFile, 0x010, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
        //printf("IMB1    READ VALUE = %lx \n", box);

        /*Reset data count value and reset to buffer 0 */
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
write_data[15], IOCTL_PPCIDMA_READ_PORT_ULONG); //switch to
buffer 0
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, write_data[8],
IOCTL_PPCIDMA_READ_PORT_ULONG); //stop transfer
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x080000000,
IOCTL_PPCIDMA_READ_PORT_ULONG);
        //printf("Reset Flip-flops - 0x080000000\n\n");

//      free(dmaptr);
//      free(secptr);
//      free(triptr);
//      free(meptr);
//      free(bobptr);
//      free(jimptr);
//      fclose(in_addr);
        fclose(outfile1);
        fclose(outfile2);
        fclose(outfile3);
        fclose(outfile4);
        fclose(outfile5);
        fclose(outfile6);
        fclose(outfile7);
        fclose(outfile8);
```

```
        fclose(outfile9);


        //printf(" END!!! \n");

finish1:
#ifdef DEBUG
        printf("Attempting to unmap, at address %x\n",
VirtualAddress);
#endif

        status = UnMapPciDmaBuffer(hndFile, VirtualAddress);
        if (status != STATUS_SUCCESS)
                printf("ERROR : Failed to unmap memory.\n");

finish2:
        if (!CloseHandle(hndFile))
                printf("ERROR : Failed to close handle.\n");

        return(status);

}


/*********************************************************
* Saveimage
*       This routine will write the output image to a disk file
in the ELAS format
*
*/
void save_image(unsigned char *image, int numchan, int width, int
numframes, FILE *out1, FILE *sfile)
{
        unsigned long int indx;
        long int x[7];
        unsigned char bytev, b[900];
        unsigned int row, col;


        //printf("\n  Saving image to disk ...");

        x[0] = 0;
        x[1] = 0;
        x[2] = 1;
        x[3] = numframes;
        x[4] = 1;
        x[5] = width;
        x[6] = numchan;
        memcpy(b,x,28);
/* Copy the header to file */
        for(indx=0; indx < width; indx++) fputc(b[indx], out1);


#ifdef DEBUG
        printf("Save the string data - %x\n", image);
#endif
         for(row=0; row<200; row++){
                col=0;
                do{
                          bytev = *(image + width*row + col);
                        fprintf(sfile, "%2x ", bytev);
                        col++;
                } while(col<width);
                fprintf(sfile, "__\n");
        }
```

```
#ifdef DEBUG
        printf("Next save the fourth file - %x\n", image);
#endif
        for(row=0; row<numframes; row++){
                for(col=0; col < width; col++){
                        fputc( *(image + width*row + col), out1);
                }
        }


        //printf(" Images are stored to disk!!!\n");

}


/*********************************************************
* ReadTimer
*
*/
int read_timer(int subtimer, int maintimer, HANDLE hndFile)
{
        int i, low, high, number;
        DWORD box, time, control;

        if (maintimer==0) {
                time = (subtimer<<19)|0x01A220000;
                control = 0x01A430000;
        }

        if (maintimer==1) {
                time = (subtimer<<19)|0x01A240000;
                control = 0x01A450000;
        }

        /* Latch the timer values to be read (Counter Latch
Command)*/
        if(subtimer==1) {
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
control|0x0E200, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
         //getchar();
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO, time,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
                Sleep(1);
         //printf("time = %lx\n", time);
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01F000000, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
         //getchar();

        /* Wait for BOARD to respond */
                box = 0x0000;
                i=0;
                do {
                        ReadPortDouble(hndFile, AMCC_OP_REG_MBEF,
&box, IOCTL_PPCIDMA_READ_PORT_ULONG);
                        box = box & 0x0000F0000;
                        i++;
                } while (box!=0x0000F0000 && i<4800);

                if(box!=0x000F0000) printf("MAILBOX FLAG NOT
SET!\n");

                Sleep(1);

        /* Is the data correct? */
                ReadPortDouble(hndFile, 0x010, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
```

```
        //printf("        READ VALUE = %lx \n", box);
                low = box&0x0FF;
        //printf("low = %x \n", low);
        //        getchar();
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
control, IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 32 Timer 0 - counter
0
        //        printf("Latched data for first timer =
0x01A430000\n");
        }  //if subtimer==1


        if (subtimer==2) {
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
control|0x0E400, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        //getchar();
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO, time,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
                Sleep(1);
        //printf("time = %lx\n", time);
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01F000000, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        //getchar();

        /* Wait for BOARD to respond */
                box = 0x0000;
                i=0;
                do {
                        ReadPortDouble(hndFile, AMCC_OP_REG_MBEF,
&box, IOCTL_PPCIDMA_READ_PORT_ULONG);
                        box = box & 0x0000F0000;
                        i++;
                } while (box!=0x0000F0000 && i<4800);
                if(box!=0x000F0000) printf("MAILBOX FLAG NOT
SET!\n");

                Sleep(1);
                /* Is the data correct? */
                ReadPortDouble(hndFile, 0x010, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
        //printf("        READ VALUE = %lx \n", box);
                low = box&0x0FF;
        //printf("low = %x \n", low);
        //        getchar();
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
control|0x04000, IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 32 Timer 0 -
counter 0
        //        printf("Latched data for second timer = %x\n",
control|0x04000);
        } //if subtimer==2


        if (subtimer==3) {
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
control|0x0E800, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        //getchar();
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO, time,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
                Sleep(1);
        //printf("time = %lx\n", time);
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01F000000, IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        //getchar();

                /* Wait for BOARD to respond */
```

```
                box = 0x0000;
                i=0;
                do {
                        ReadPortDouble(hndFile, AMCC_OP_REG_MBEF,
&box, IOCTL_PPCIDMA_READ_PORT_ULONG);
                        box = box & 0x0000F0000;
                        i++;
                } while (box!=0x0000F0000 && i<4800);

                if(box!=0x000F0000) printf("MAILBOX FLAG NOT
SET!\n");

                Sleep(1);

                /* Is the data correct? */
                ReadPortDouble(hndFile, 0x010, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
        //printf("        READ VALUE = %lx \n", box);
                low = box&0x0FF;
        //printf("low = %x \n", low);
        //        getchar();
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
control|0x08000, IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 32 Timer 0 -
counter 0
        //        printf("Latched data for third timer = %x\n",
control|0x08000);
        } //if subtimer ==3


    //getchar();
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, time,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
        Sleep(1);
    //printf("time = %lx\n", time);
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01F000000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
    //getchar();

        /* Wait for BOARD to respond */
        box = 0x0000;
        i=0;
        do {
                ReadPortDouble(hndFile, AMCC_OP_REG_MBEF, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
                box = box & 0x0000F0000;
                i++;
        } while (box!=0x0000F0000 && i<4800);

        if(box!=0x000F0000) printf("MAILBOX FLAG NOT SET!\n");

        Sleep(1);
        /* Is the data correct? */
        ReadPortDouble(hndFile, 0x010, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
    //printf("        READ VALUE = %lx \n", box);
        low = box&0x0FF;
    //printf("low = %x \n", low);

        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, time,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
    //printf("                data_written = %lx\n",time);
        Sleep(1);

        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01F000000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
    //getchar();
```

```
        /* Wait for BOARD to respond */
        box = 0x0000;
        i=0;
        do {
                ReadPortDouble(hndFile, AMCC_OP_REG_MBEF, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
                box = box & 0x0000F0000;
               i++;
        } while (box!=0x0000F0000 && i<4800);

        if(box!=0x000F0000) printf("MAILBOX FLAG NOT SET!\n");

        /* Is the data correct? */
        ReadPortDouble(hndFile, 0x010, &box,
IOCTL_PPCIDMA_READ_PORT_ULONG);
    //printf("        READ VALUE = %lx \n", box);
        high = box&0x0FF;
    //printf(" high = %x \n", high);

        number = (high<<8)|low;
    //printf(" number = %d \n", number);
        return(number);
}


/*********************************************************
* ProgramTimer
*
*/
void program_timer(int numframes, int width, int startpix, int
timer, HANDLE hndFile)
{
int offset;

        if(timer==0) {
                /* First program the Control Words */
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A433000, IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 32 Timer 0 -
counter 0
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01A437000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 72 Timer 0 - counter 1
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01A43B000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // B0 Timer 0 - counter 2
                /* Next Program the count values */
                offset = (numframes&0x0FF);
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A4A0000+offset, IOCTL_PPCIDMA_WRITE_PORT_ULONG); // offset
(40 + startpix)
        offset = ((numframes>>8)&0x0FF);
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A4A0000+offset, IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        width = width;
                WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A520000+(width&0x0FF), IOCTL_PPCIDMA_WRITE_PORT_ULONG); //
width of window
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A520000+((width>>8)&0x0FF), IOCTL_PPCIDMA_WRITE_PORT_ULONG);

        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A5A0000+(startpix&0x0FF), IOCTL_PPCIDMA_WRITE_PORT_ULONG);
//number of frames
        WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A5A0000+((startpix>>8)&0x0FF),
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
```

```
	}

	if(timer==1) {
		/* First program the Control Words */
	 WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01A453000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 30 Timer 0 - counter 0
	 WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01A457000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // 70 Timer 0 - counter 1
	 WritePortDouble(hndFile, AMCC_OP_REG_FIFO, 0x01A45B000,
IOCTL_PPCIDMA_WRITE_PORT_ULONG); // B0 Timer 0 - counter 2

		 /* Next Program the count values */
		offset = (numframes&0x0FF);
		WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A4C0000+offset, IOCTL_PPCIDMA_WRITE_PORT_ULONG); // offset
(40 + startpix)
	 offset = ((numframes>>8)&0x0FF);
		WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A4C0000+offset, IOCTL_PPCIDMA_WRITE_PORT_ULONG);

	 width = width;
		WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A540000+(width&0x0FF), IOCTL_PPCIDMA_WRITE_PORT_ULONG);
	 WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A540000+((width>>8)&0x0FF), IOCTL_PPCIDMA_WRITE_PORT_ULONG);

	 WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A5C0000+(startpix&0x0FF), IOCTL_PPCIDMA_WRITE_PORT_ULONG);
//number of frames
	 WritePortDouble(hndFile, AMCC_OP_REG_FIFO,
0x01A5C0000+((startpix>>8)&0x0FF),
IOCTL_PPCIDMA_WRITE_PORT_ULONG);
	}
}



int Read_Configuration(int *numframes, int *width1, int *width2,
int *width3, int *startpix, int *channel_enable)
{
	FILE *cfgfile;
	int status;

	cfgfile = fopen(CONFIGURATION_FILENAME, "r");
	if(cfgfile == NULL) {
		printf("ERROR : Unable to open %s.\n",
CONFIGURATION_FILENAME);
		return(STATUS_FAILURE);
	}

	status = fscanf(cfgfile, "NUMFRAMES = %d\n", numframes);
	if ((status == EOF) || (status == 0))
		goto error;

	status = fscanf(cfgfile, "WIDTH1 = %d\n", width1);
	if ((status == EOF) || (status == 0))
		goto error;

	status = fscanf(cfgfile, "WIDTH2 = %d\n", width2);
	if ((status == EOF) || (status == 0))
		goto error;

	status = fscanf(cfgfile, "WIDTH3 = %d\n", width3);
	if ((status == EOF) || (status == 0))
```

```
                goto error;


        status = fscanf(cfgfile, "STARTPIX = %d\n", startpix);
        if ((status == EOF) || (status == 0))
                goto error;

#ifdef DEBUG
        printf("Read STARTPIX = %d\n", *startpix);
#endif

        status = fscanf(cfgfile, "CHANNELS = %d\n",
channel_enable);
        if ((status == EOF) || (status == 0))
                goto error;

#ifdef DEBUG
        printf("Read CHANNELS = %d\n", *channel_enable);
#endif

        fclose(cfgfile);

        return(STATUS_SUCCESS);

error:
        printf("ERROR : Invalid configuration file.\n");
        fclose(cfgfile);
        return(STATUS_FAILURE);
}
```

EDISP.CPP

```
//
// EDISP
//   A program to display ELAS images under Windows.
//
// Written by : Panos Arvanitis
// Date       :
// Revised    : 9/14/1997
// Version    : 2.0
//
//
// DO NOT ERASE THE FOLLOWING LINES
// No, you don't know what they say, but I do.  So, leave them
// there.
//
// Ìðïñåß íá ñùôÞóåé êáíåßò "ãéáôß ôï Ýãñáøåò óôá Åëëçíéêü áõôü
// ñå, áöïý êáíÝíáò äå ìðïñåß íá ôï äéáâÜóåé;".  ¸ëá üìùò ðïõ
// êÜðïéïò ôï äéáâÜæåé!!!  ÊÜèïìáé åäþ êé áêïýù ÍôáëÜñá, êáé
// åßðá íá ðù: "×áßñå ÅëëÜá, ðáôñßäá"

#include <windows.h>
#include <string.h>
#include <mem.h>
#include <math.h>


int numframes, width, numchan;
int BoxWidth, BoxHeight;    //Dimensions of display box
int HScrollPos, VScrollPos; //Position of X and Y scroll bar
HBITMAP hBitMap;
LPVOID lpvBits;
HINSTANCE hCurrInst;    //handle to current application instance
```

```
HWND hDialog = NULL;    //handle to dialog window

#include "edisp.rh"
#include "edisp.hpp"
#include "RGBVal_Dialog.h"


void DrawImage(HWND hWnd)
{
  HDC hdc, hMemDC;
  PAINTSTRUCT ps;

  hdc = BeginPaint(hWnd, &ps);
  hMemDC = CreateCompatibleDC(hdc);
  SelectObject(hMemDC, hBitMap);

  //BitBlt(hdc, 0, 0, width, numframes, hMemDC, 0, 0, SRCCOPY);
  /*SetStretchBltMode(hdc, HALFTONE);
  StretchBlt(hdc, 0, 0, BoxWidth, numframes, hMemDC, 0, 0,
               width, numframes, SRCCOPY);*/
  BitBlt(hdc, 0, 0, width, numframes, hMemDC, 0, VScrollPos-1,
SRCCOPY);


  DeleteDC(hMemDC);
  EndPaint(hWnd, &ps);
  return;
}


void InitBitMap(HWND hWnd, HANDLE hndFile, char *ElasName)
{
  HDC hdc;
  LPBITMAPINFOHEADER lpbih;
  DWORD BytesRead;
  HANDLE TempFile;
  int count1, count2, reslt, curraddr;
  unsigned char *ElasPtr; //pointer to convert color ELAS files
  unsigned char *ElasDat; //ELAS file
  unsigned char *TempDat; //RAW (.TMP) file
  char TempName[258];     //Temporary filename

  //append .TMP to input filename
  strcpy(TempName, ElasName);
  strcat(TempName, ".raw");

  //allocate memory for DIB structure
  lpbih = (LPBITMAPINFOHEADER)GlobalLock(GlobalAlloc(GHND,
1000));

  //set DIB structure fields for DIB
  lpbih->biSize           = sizeof(BITMAPINFOHEADER);
  lpbih->biWidth          = width;      //bitmap width
  lpbih->biHeight         = -numframes; //top-down bitmap
  lpbih->biPlanes         = 1;          //one bit plane
  lpbih->biBitCount       = 24;         //24-bit bitmap
  lpbih->biCompression    = BI_RGB;     //RGB mode (no palette)
  lpbih->biSizeImage      = 0;
  lpbih->biXPelsPerMeter  = 0;
  lpbih->biYPelsPerMeter  = 0;
  lpbih->biClrUsed        = 0;
  lpbih->biClrImportant   = 0;

  //create device context for main window
  hdc = GetDC(hWnd);
```

```
  //create bitmap for DC
  hBitMap = CreateDIBSection(hdc, (LPBITMAPINFO) lpbih,
                             DIB_RGB_COLORS, &lpvBits, NULL, 0L);

  ReleaseDC(hWnd, hdc);

  //move file pointer to beginning of ELAS file
  SetFilePointer(hndFile, 0, NULL, FILE_BEGIN);

  //Create a temporary file
  TempFile = CreateFile(TempName, GENERIC_WRITE, 0,
                                          NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_TEMPORARY, NULL);

  //Allocate memory for ELAS file including header
  ElasDat = new unsigned char[(numframes+1) * width * numchan];

  //Allocate memory for temporary DIB
  TempDat = new unsigned char[numframes*width*3];

  //read the ELAS file and header in ElasDat buffer
  ReadFile(hndFile, ElasDat, (numframes+1)* width * numchan,
&BytesRead, NULL);

  //Initialize for first loop iteration
  count2 = 0;               //pixel counter
  reslt = fmod(width, 4);   //bytes for padding

  //go down bitmap, line by line
  for(count1=0; count1<numframes; count1++)
  {
    //address to beginning of current line in TMP file
    curraddr = count1 * width;

   //go across bitmap, pixel by pixel in selected line
   for(count2=0; count2<width; count2++)
   {
     if (numchan == 1)   //monochrome ELAS image
     {
       //place same value for RGB to get grayscale
       *(TempDat+curraddr)   = *(ElasDat + width + count2
                                  + count1 * width);
       *(TempDat+curraddr+1) = *(TempDat + curraddr);
       *(TempDat+curraddr+2) = *(TempDat + curraddr);

       curraddr += 3;  //increment address counter
     } // if

     else  //color ELAS file

     {
       //reduce calculation by using this variable
       ElasPtr = ElasDat + count2 + (3 * count1 + 1) * width;

       //convert lines of RGB to pixels of BGR
       *(TempDat + curraddr)     = *(ElasPtr + 2 * width);
       *(TempDat + curraddr + 1) = *(ElasPtr + width);
       *(TempDat + curraddr + 2) = *(ElasPtr);

       curraddr += 3;  //increment address counter
     } // else

   } //for count2

  //write the current line of data to the temp file
  WriteFile(TempFile, TempDat+count1*width, 3*width, &BytesRead,
```

```
            NULL);

   switch (reslt)
   {
     case 0           : break;

     case 1           : WriteFile(TempFile,
TempDat+count1*width, 1,
                            &BytesRead, NULL);
                                  break;

     case 2           : WriteFile(TempFile,
TempDat+count1*width, 1,
                            &BytesRead, NULL);
               WriteFile(TempFile, TempDat+count1*width, 1,
                            &BytesRead, NULL);
               break;

     case 3           : WriteFile(TempFile,
TempDat+count1*width, 1,
                            &BytesRead, NULL);
                                  WriteFile(TempFile,
TempDat+count1*width, 1,
                            &BytesRead, NULL);
               WriteFile(TempFile, TempDat+count1*width, 1,
                            &BytesRead, NULL);
               break;

     default   :      break;
   }  //switch

  } //for count1
```

```
  //close the temp file
  CloseHandle(TempFile);

  //re-open temp (DIB) file for reading
  TempFile = CreateFile(TempName, GENERIC_READ, 0,
                                                          NULL,
OPEN_EXISTING,
                        FILE_ATTRIBUTE_TEMPORARY, NULL);
  //read the raw image data
  ReadFile(TempFile, lpvBits, 3*numframes*width, &BytesRead,
NULL);

  //close the temp file
  CloseHandle(TempFile);

  //free memory
  delete(ElasDat);
  delete(TempDat);

  //delete temp file
  DeleteFile(TempName);
}

// WindowResize
//    This function is executed when the window is resized.
//    Scroll bar information must be updated.
void WindowResize(HWND hWnd, int BoxWidth, int BoxHeight)
{
  SCROLLINFO si;

  si.cbSize = sizeof(si);
  si.fMask  = SIF_ALL;
  si.nMin   = 1;
```

```
  si.nMax   = numframes - BoxHeight - 1;
  si.nPage  = BoxHeight;
  si.nPos   = 1;
  SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
  si.fMask  = SIF_PAGE;
  SetScrollInfo(hWnd, SB_VERT, &si, TRUE);

  ShowScrollBar(hWnd, SB_VERT, TRUE);

  DrawImage(hWnd);

}


void VerticalScroll(HWND hWnd, int ScrollCode, int ScrollPos)
{
  int NewPos = 0;     //new position
  int DeltaPos = 0;   //change from previous pos
  SCROLLINFO si;      //structure to hold scroll bar info

  switch (ScrollCode)
  {
    case SB_PAGEUP:
      NewPos = VScrollPos - 50;
      break;
    case SB_PAGEDOWN:
      NewPos = VScrollPos + 50;
      break;
    case SB_LINEUP:
      NewPos = VScrollPos - 5;
      break;
    case SB_LINEDOWN:
      NewPos = VScrollPos + 5;
      break;
    case SB_THUMBPOSITION:
      NewPos = ScrollPos;
      break;
    default:
      NewPos = VScrollPos;
  }

  //Do bound checking
  NewPos = max(0, NewPos);    //0 is minimum value
  NewPos = min(numframes - BoxHeight + 1, NewPos);  //max number
of lines

  if (NewPos == VScrollPos)
    return;   //No scrolling

  //Update position
  DeltaPos = NewPos - VScrollPos;
  VScrollPos = NewPos;

  //Scroll the window
  ScrollWindowEx(hWnd, 0, -DeltaPos, (CONST RECT *) NULL,
                   (CONST RECT *) NULL, (HRGN) NULL,
                   (LPRECT) NULL, SW_INVALIDATE);
  //and update it (re-draw)
  UpdateWindow(hWnd);

  //Now tell Windows to re-position the scroll bar
  si.cbSize = sizeof(si);
  si.fMask  = SIF_POS;
  si.nPos   = VScrollPos;
  SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
}
```

```
void LeftButtonPushed(HWND hWnd, int xPos, int yPos)
{

  //Create a dialog box if it doesn't already exist
  if (hDialog == NULL)
    hDialog = CreateDialog(hCurrInst,
MAKEINTRESOURCE(IDD_RGBVAL),
                           hWnd, (DLGPROC) RGBValProc);
}


void CommandProc(HWND hWnd, int ControlID)
{
  if (ControlID == IDB_DISMISS)
  {
    DestroyWindow(hDialog);
    hDialog = NULL;
  }
}



//Read the ELAS header
//For PC, byte order is reversed, i.e. 5 is 5000, not 0005
//Check bytes 8-11 (should be 1) to decide whether to swap the
header
BOOL ReadHeader(HANDLE hndFile)
{
  DWORD BytesRead;
  char header[HEADERLENGTH];

  if (ReadFile(hndFile, &header, HEADERLENGTH, &BytesRead, NULL))
  {
    //Assign numframes
    memcpy(&numframes, &header[NUMFRAMESLOC], 4);
    //Assign width
    memcpy(&width, &header[WIDTHLOC], 4);
    //Assign numchan
    memcpy(&numchan, &header[NUMCHANLOC], 4);
    if ((numchan != 1) & (numchan != 3))
    {
      MessageBox(NULL, "Only grayscale images can be displayed",
                                        "Unsupported Feature",
MB_OK);
      return FALSE;
    }
  }

  return TRUE;
}



// WndProc
//    Handles all messages to the main window
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
  switch (uMsg)
  {
    case WM_PAINT:
      DrawImage(hWnd);
```

```
      break;
    case WM_DESTROY:
      PostQuitMessage(0);
      break;
    case WM_SIZE:
      BoxWidth  = LOWORD(lParam);
      BoxHeight = HIWORD(lParam);
      WindowResize(hWnd, BoxWidth, BoxHeight);
      break;
    case WM_VSCROLL:
      VerticalScroll(hWnd, (int) LOWORD(wParam),
                       (int) HIWORD(wParam));
      break;
    case WM_LBUTTONDOWN:
        LeftButtonPushed(hWnd, (int) LOWORD(lParam),

        (int) HIWORD(lParam));
      break;
    case WM_COMMAND:
      if ((int) HIWORD(wParam) == BN_CLICKED)
      {
        CommandProc(hWnd, (int) LOWORD(wParam));
        break;
      }
    default:
      return DefWindowProc(hWnd, uMsg, wParam, lParam);
  }

  return 0;
}
```

```
int WINAPI WinMain(HINSTANCE hCur, HINSTANCE hPrev, LPSTR
lpCmdLn, int CmdShow)
{
  MSG msg;
  HWND hWnd;          //handle to main window
  HWND hDesktop;      //handle to desktop
  HDC  hDesktopDC;    //Device Context to desktop
  HANDLE ImgFile;     //handle to ELAS file
  WNDCLASS wndClass;
  SCROLLINFO si;
  int WinWidth, WinHeight;  //dimensions of original window
  int ScreenRes;
  char ElasName[255];   //filename of ELAS file

  strcpy(ElasName, lpCmdLn);  //command line is filename

  //Open the input file and optimize for sequential scan
  ImgFile = CreateFile(ElasName, GENERIC_READ, FILE_SHARE_READ,
              NULL, OPEN_EXISTING,
              FILE_ATTRIBUTE_NORMAL | FILE_FLAG_SEQUENTIAL_SCAN,
              NULL);

  if (ImgFile == INVALID_HANDLE_VALUE)
    MessageBox(NULL, "Error opening input file", "Fatal Error",
              MB_OK);
  else
  {
    if (hPrev == NULL)
    {
      memset(&wndClass, 0, sizeof(wndClass));
      wndClass.style = CS_HREDRAW | CS_VREDRAW;
      wndClass.lpfnWndProc = WndProc;
```

```
      wndClass.hInstance = hCur;
      wndClass.hCursor = LoadCursor(NULL, IDC_NO);
      wndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
      wndClass.lpszClassName = "ELASDISP";
      if (!RegisterClass(&wndClass)) return FALSE;
    }

    //Read the header of the ELAS file and check
    //Assumes that the images are in PC format
    if (!ReadHeader(ImgFile))
      //Error in reading header
      MessageBox(NULL, "Format Error", "Incorrect file format",
MB_OK);

    else
    {
      hDesktop   = GetDesktopWindow();  //get handle to desktop
      hDesktopDC = GetDC(hDesktop);     //get DC to desktop

      //check if image is wider than the screen
      ScreenRes = GetDeviceCaps(hDesktopDC, HORZRES);
      if (width > ScreenRes - 8)
        BoxWidth = ScreenRes - 20;
      else
        BoxWidth = width;

      //check if image is longer than the screen
      ScreenRes = GetDeviceCaps(hDesktopDC, VERTRES);
      if (numframes > ScreenRes - 30)
        BoxHeight = ScreenRes - 100;
      else
        BoxHeight = numframes;

      HScrollPos = 1;
      VScrollPos = 1;

      //Determine the dimensions of the window.  We must allow
      //space for borders and title bars.
      WinWidth = BoxWidth + 2 * GetSystemMetrics(SM_CXBORDER) +
                 GetSystemMetrics(SM_CXVSCROLL) + 10;

      WinHeight = BoxHeight + 2 * GetSystemMetrics(SM_CYBORDER) +
                 //GetSystemMetrics(SM_CYHSCROLL) +
                 GetSystemMetrics(SM_CYSIZE) + 10;

      hWnd = CreateWindow("ELASDISP", lpCmdLn,
                          //WS_OVERLAPPEDWINDOW |
                          //WS_HSCROLL |
                          WS_OVERLAPPED | WS_CAPTION |
                          WS_SYSMENU | WS_MINIMIZEBOX |
                          WS_MAXIMIZEBOX |
                          WS_VSCROLL,
                          CW_USEDEFAULT, 0,
                          WinWidth, WinHeight,
                          NULL, NULL, hCur, NULL);

      hCurrInst = hCur;

      //ShowWindow(hWnd, CmdShow);
      ShowWindow(hWnd, SW_MINIMIZE);

      SetFilePointer(ImgFile, 0, NULL, FILE_BEGIN);
      InitBitMap(hWnd, ImgFile, &ElasName[0]);
      CloseHandle(ImgFile);

      UpdateWindow(hWnd);
```

```
      si.cbSize = sizeof(si);
      si.fMask  = SIF_ALL;
      si.nMin   = 1;
      si.nMax   = numframes - BoxHeight - 1;
      si.nPage  = BoxHeight;
      si.nPos   = 1;
      SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
      ShowWindow(hWnd, SW_RESTORE);

      si.fMask  = SIF_PAGE;
      SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
      ShowScrollBar(hWnd, SB_VERT, TRUE);

      while (GetMessage(&msg, NULL, 0, 0))
        DispatchMessage(&msg);
      return msg.wParam;
    }  //else
  }
}
```

EDISP.HPP

```
#define HEADERLENGTH   28
#define NUMFRAMESLOC   12
#define WIDTHLOC       20
#define NUMCHANLOC     24
```

GLXDLGCL.CPP

```
//----------------------------------------------------------------
//  Project Galaxie
//
//  Copyright © 1997. All Rights Reserved.
//
//  SUBSYSTEM:    Galaxie Application
//  FILE:         glxdlgcl.cpp
//  AUTHOR:
//
//  OVERVIEW
//  ~~~~~~~~
//  Source file for implementation of TGalaxieDlgClient
// (TDialog).
//
//----------------------------------------------------------------

#include <owl/pch.h>

#include "galaxapp.h"
#include "glxdlgcl.h"

#include <stdlib.h>
#include <process.h>

#include "\users\panos\progs\lib\hardware.h"
#include "\users\panos\progs\lib\sensor.hpp"
#include "globals.h"
#include "galaxie.hpp"

//
// Build a response table for all messages/commands handled by
the application.
```

```
//
DEFINE_RESPONSE_TABLE1(TGalaxieDlgClient, TDialog)
//{{TGalaxieDlgClientRSP_TBL_BEGIN}}
  EV_BN_CLICKED(IDC_RADIOBUTTONXMISSION, BNClickedXMission),
  EV_BN_CLICKED(IDC_BUTTONSCAN, BNClickedScan),
  EV_BN_CLICKED(IDC_RADIOBUTTONBSCATTER, BNClickedBScatter),
  EV_BN_CLICKED(IDC_RADIOBUTTONEHIGH, BNClickedEHigh),
  EV_BN_CLICKED(IDC_RADIOBUTTONELOW, BNClickedELow),
  EV_BN_CLICKED(IDC_RADIOBUTTONFSCATTER, BNClickedFScatter),
  EV_BN_CLICKED(IDC_RADIOBUTTONOVOFF, BNClickedOvOFF),
  EV_BN_CLICKED(IDC_RADIOBUTTONOVON, BNClickedOvON),
  EV_BN_CLICKED(IDC_BUTTONDISPLAY, BNClickedDisplay),
//{{TGalaxieDlgClientRSP_TBL_END}}
END_RESPONSE_TABLE;


//{{TGalaxieDlgClient Implementation}}


//----------------------------------------------------------
// TGalaxieDlgClient
// ~~~~~~~~~~
// Construction/Destruction handling.
//
TGalaxieDlgClient::TGalaxieDlgClient(TWindow* parent, TResId
resId, TModule* module)
:
  TDialog(parent, resId, module)
{
       int CorrVal1, CorrVal2, CorrVal3;

  //Add a box here that will let the user know the status of the
initilization
  if(!SetUpMotorController(&hMotor))
  {
    MessageBox("Failed to initialize the motor controller",
"Fatal Error",
                                                    MB_ICONHAND
| MB_OK);
    exit(EXIT_FAILURE);
  }

  ProgMotorController(hMotor);

  if (!SetUpXrayController(&hXray))
  {
    MessageBox("Failed to initialize the X-ray controller",
"Fatal Error",
                                                    MB_ICONHAND
| MB_OK);
    exit(EXIT_FAILURE);
  }

  if (!SetUpDPIB(&hDpib))
  {
    MessageBox("Failed to initialize DPIB", "Fatal Error",
                                                    MB_ICONHAND
| MB_OK);
    exit(EXIT_FAILURE);
  }

  StopBelt(hDpib);
  RaiseFilter(hMotor);
  SetKV75(hXray);
  SetmA300(hXray);
  TurnXrayON(hXray);
```

```
  CorrVal1 = CORRVAL1LOW;
  CorrVal2 = CORRVAL2LOW;
  CorrVal3 = CORRVAL3LOW;
  SetUpCorrVal(CorrVal1, CorrVal2, CorrVal3, hDpib);
}


TGalaxieDlgClient::~TGalaxieDlgClient()
{
  StopBelt(hDpib);
  LowerFilter(hMotor);
  TurnXrayOFF(hXray);

  CloseHandle(hMotor);
  CloseHandle(hXray);
  CloseHandle(hDpib);

  Destroy();

}

void TGalaxieDlgClient::BNClickedXMission()
{
  if (CurrAppStatus.ImgSource != Transmission)
  {
    CurrAppStatus.ImgSource = Transmission;
  }

}


void TGalaxieDlgClient::BNClickedScan()
{
  int CorrVal1, CorrVal2, CorrVal3;
  int childproc;

  //Start the conveyor belt
  SetDlgItemText(IDC_EDITSTATUS, "Starting conveyor belt...");

       //Turn the Xray ON, just to make sure...
  SetKV75(hXray);
  SetmA300(hXray);
  TurnXrayON(hXray);

  //Wait for luggage
  SetDlgItemText(IDC_EDITSTATUS, "Waiting for luggage...");
  MoveBeltForward(hDpib);
  BreakFrontSensor(hDpib);
  StopBelt(hDpib);

  //Prepare to scan
  SetDlgItemText(IDC_EDITSTATUS, "Luggage Detected! Preparing to
scan...");
/*  CorrVal1 = CORRVAL1LOW;
  CorrVal2 = CORRVAL2LOW;
  CorrVal3 = CORRVAL3LOW;
  SetUpCorrVal(CorrVal1, CorrVal2, CorrVal3, hDpib);
*/
       //*****************************************************
******************

//*************************************************************
***********
  // Use CreateProcess instead of spawn
```

```
//*************************************************************
//*************************************************************

  childproc = spawnlp(P_NOWAIT, "COLPUL-SILENT", "COLPUL-SILENT",
NULL);
  SetDlgItemText(IDC_EDITSTATUS, "Start scanning...");
  WaitSeconds(1);

  MoveBeltForward(hDpib);
  BreakRearSensor(hDpib);
  UnBreakRearSensor(hDpib);

  SetDlgItemText(IDC_EDITSTATUS, "Waiting for collection to
finish...");
  StopBelt(hDpib);
  cwait(NULL, childproc, WAIT_CHILD);

  //Copy the files to new names

  WaitSeconds(5);
  spawnlp(P_WAIT, "imgconv", "imgconv one.img", NULL);
  spawnlp(P_WAIT, "imgconv", "imgconv two.img", NULL);
  spawnlp(P_WAIT, "imgconv", "imgconv three.img", NULL);

  CopyFile(CHAN1COLLECTED, XMISSIONLOWCOLLECTED, FALSE);
  CopyFile(CHAN2COLLECTED, BSCATTERLOWCOLLECTED, FALSE);
  CopyFile(CHAN3COLLECTED, FSCATTERLOWCOLLECTED, FALSE);


  SetDlgItemText(IDC_EDITSTATUS, "Lowering filter and raising
voltage...");
  LowerFilter(hMotor);
  SetKV150(hXray);
  WaitSeconds(4);

  SetDlgItemText(IDC_EDITSTATUS, "Moving back luggage and
preparing to scan...");
/*  CorrVal1 = CORRVAL1HIGH;
  CorrVal2 = CORRVAL2HIGH;
  CorrVal3 = CORRVAL3HIGH;
  SetUpCorrVal(CorrVal1, CorrVal2, CorrVal3, hDpib);
*/

  MoveBeltReverse(hDpib);
  BreakRearSensor(hDpib);
  BreakFrontSensor(hDpib);
  UnBreakFrontSensor(hDpib);
  StopBelt(hDpib);

  SetDlgItemText(IDC_EDITSTATUS, "Start scanning...");
  childproc = spawnlp(P_NOWAIT, "COLPUL-SILENT", "COLPUL-SILENT",
NULL);
  WaitSeconds(1);

  MoveBeltForward(hDpib);
  BreakRearSensor(hDpib);
  UnBreakRearSensor(hDpib);

  SetDlgItemText(IDC_EDITSTATUS, "Raising filter and lowering
voltage...");
  SetKV75(hXray);
  RaiseFilter(hMotor);

  SetDlgItemText(IDC_EDITSTATUS, "Waiting for collection to
finish...");
```

```
  cwait(NULL, childproc, WAIT_CHILD);
  StopBelt(hDpib);
  TurnXrayOFF(hXray);

  WaitSeconds(5);
  spawnlp(P_WAIT, "imgconv", "imgconv one.img", NULL);
  spawnlp(P_WAIT, "imgconv", "imgconv two.img", NULL);
  spawnlp(P_WAIT, "imgconv", "imgconv three.img", NULL);

  CopyFile(CHAN1COLLECTED, XMISSIONHIGHCOLLECTED, FALSE);
  CopyFile(CHAN2COLLECTED, BSCATTERHIGHCOLLECTED, FALSE);
  CopyFile(CHAN3COLLECTED, FSCATTERHIGHCOLLECTED, FALSE);



  spawnlp(P_WAIT, "process1.bat", "process1.bat v150a.img
v75a.img v75c.img v75b.img",
                                                  NULL);

  spawnlp(P_NOWAIT, "FaaDisp", "FaaDisp cut_lt.img", NULL);

  spawnlp(P_WAIT, "process2.bat", "process2.bat", NULL);

  SetDlgItemText(IDC_EDITSTATUS, "Processing finished!");
}


void TGalaxieDlgClient::BNClickedBScatter()
{
  if (CurrAppStatus.ImgSource != Backscatter)
  {
    CurrAppStatus.ImgSource = Backscatter;
  }
}


void TGalaxieDlgClient::BNClickedEHigh()
{
  if (CurrAppStatus.ESource != High)
  {
    CurrAppStatus.ESource = High;
  }

}


void TGalaxieDlgClient::BNClickedELow()
{
  if (CurrAppStatus.ESource != Low)
  {
    CurrAppStatus.ESource = Low;
  }

}


void TGalaxieDlgClient::BNClickedFScatter()
{
  if (CurrAppStatus.ImgSource != ForwardScatter)
  {
    CurrAppStatus.ImgSource = ForwardScatter;
  }

}
```

```
void TGalaxieDlgClient::BNClickedOvOFF()
{
       if (CurrAppStatus.Over != Off)
  {
    CurrAppStatus.Over = Off;
  }

}


void TGalaxieDlgClient::BNClickedOvON()
{
       if (CurrAppStatus.Over != On)
  {
    CurrAppStatus.Over = On;
  }

}


void TGalaxieDlgClient::BNClickedDisplay()
{
  if (CurrAppStatus.Over == On)
  {
    DoOverlap();

    switch (CurrAppStatus.ImgSource)
    {
         case Transmission           :

                              if (CurrAppStatus.ESource == Low)
                              SetDlgItemText(IDC_EDITSTATUS, "Low
energy transmission with overlap");
                              else
                              SetDlgItemText(IDC_EDITSTATUS,
"High energy transmission with overlap");
                              break;
       case Backscatter              :

                              if (CurrAppStatus.ESource == Low)
                              SetDlgItemText(IDC_EDITSTATUS, "Low
energy backscatter with overlap");
                              else
                                     MessageBox("File not
available", "Unsupported Feature",

       MB_ICONINFORMATION | MB_OK);
                              break;
       case ForwardScatter    :

                              if (CurrAppStatus.ESource == Low)
                              SetDlgItemText(IDC_EDITSTATUS, "Low
energy forward scatter with overlap");
                              else
                                     MessageBox("File not
available", "Unsupported feature",

       MB_ICONINFORMATION | MB_OK);
                              break;
       default                                                :
       break;
       }

   spawnlp(P_NOWAIT, "FaaDisp", "FaaDisp overon.img", NULL);
  }
```

```
  else
  {
    switch (CurrAppStatus.ImgSource)
    {
          case Transmission           :

                                if (CurrAppStatus.ESource == Low)
                               {
                                 SetDlgItemText(IDC_EDITSTATUS, "Low
energy transmission, no overlap");

                                        spawnlp(P_NOWAIT,
"FaaDisp", "FaaDisp cut_lt.img", NULL);
                               }

                                 else
                               {
                                 SetDlgItemText(IDC_EDITSTATUS,
"High energy transmission, no overlap");
                                        spawnlp(P_NOWAIT,
"FaaDisp", "FaaDisp cut_ht.img", NULL);
                               }
                                 break;
        case Backscatter              :

                                 if (CurrAppStatus.ESource == Low)
                               {
                                 SetDlgItemText(IDC_EDITSTATUS, "Low
energy backscatter, no overlap");
                                        spawnlp(P_NOWAIT,
"FaaDisp", "FaaDisp cut_bs.img", NULL);
                               }
                                 else
                                        MessageBox("File not
available", "Unsupported Feature",

        MB_ICONINFORMATION | MB_OK);
                                  break;
        case ForwardScatter    :

                                  if (CurrAppStatus.ESource == Low)
                                {
                                  SetDlgItemText(IDC_EDITSTATUS, "Low
energy forward scatter, no overlap");
                                        spawnlp(P_NOWAIT,
"FaaDisp", "FaaDisp cut_fs.img", NULL);
                                }
                                  else
                                        MessageBox("File not
available", "Unsupported feature",

        MB_ICONINFORMATION | MB_OK);
                                  break;
        default                                          :
        break;
        }
  }
}
```

GLXDLGCL.H

```
//----------------------------------------------------------------
//  Project Galaxie
//
//  Copyright © 1997. All Rights Reserved.
//
```

```
//  SUBSYSTEM:    Galaxie Application
//  FILE:         glxdlgcl.h
//  AUTHOR:
//
//  OVERVIEW
//  ~~~~~~~~
//  Class definition for TGalaxieDlgClient (TDialog).
//
//----------------------------------------------------------------
-------------
#if !defined(glxdlgcl_h)              // Sentry, use file only if
it's not already included.
#define glxdlgcl_h

#include "galaxapp.rh"                  // Definition of all
resources.


#include <owl/commctrl.h>

//{{TDialog = TGalaxieDlgClient}}
class TGalaxieDlgClient : public TDialog {
  public:
    TGalaxieDlgClient(TWindow* parent, TResId resId = IDD_CLIENT,
TModule* module = 0);
    virtual ~TGalaxieDlgClient();



//{{TGalaxieDlgClientRSP_TBL_BEGIN}}
  protected:
    void BNClickedXMission();
    void BNClickedScan();
    void BNClickedBScatter();
    void BNClickedEHigh();
    void BNClickedELow();
    void BNClickedFScatter();
    void BNClickedOverON();
    void BNClickedOvOFF();
    void BNClickedOvON();
    void BNClickedDisplay();
//{{TGalaxieDlgClientRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(TGalaxieDlgClient);
};    //{{TGalaxieDlgClient}}


#endif  // glxdlgcl_h sentry.
```

GALAXAPP.CPP

```
//----------------------------------------------------------------
//  Project Galaxie
//
//  Copyright © 1997. All Rights Reserved.
//
//  SUBSYSTEM:    Galaxie Application
//  FILE:         galaxapp.cpp
//  AUTHOR:
//
//  OVERVIEW
//  ~~~~~~~~
//  Source file for implementation of TGalaxieApp (TApplication).
//
//----------------------------------------------------------------

#include <owl/pch.h>
```

```
#include <owl/statusba.h>
#include <stdio.h>

#include "galaxapp.h"
#include "glxdlgcl.h"                          // Definition of
client class.


//{{TGalaxieApp Implementation}}


//
// Build a response table for all messages/commands handled
// by the application.
//
DEFINE_RESPONSE_TABLE1(TGalaxieApp, TApplication)
//{{TGalaxieAppRSP_TBL_BEGIN}}
  EV_COMMAND(CM_HELPABOUT, CmHelpAbout),
//{{TGalaxieAppRSP_TBL_END}}
END_RESPONSE_TABLE;


//----------------------------------------------------------
// TGalaxieApp
//
TGalaxieApp::TGalaxieApp() : TApplication("Galaxie")
{

  // INSERT>> Your constructor code here.
}


TGalaxieApp::~TGalaxieApp()
{
  // INSERT>> Your destructor code here.
}


//----------------------------------------------------------
// TGalaxieApp
// ~~~~~
// Application intialization.
//
void TGalaxieApp::InitMainWindow()
{
  if (nCmdShow != SW_HIDE)
    nCmdShow = (nCmdShow != SW_SHOWMINNOACTIVE) ? SW_SHOWNORMAL :
nCmdShow;

  TSDIDecFrame* frame = new TSDIDecFrame(0, GetName(), 0, true);
  frame->SetFlag(wfShrinkToClient);

  // Override the default window style for the main window.
  //
      frame->Attr.Style |= WS_BORDER | WS_CAPTION |
WS_CLIPCHILDREN | WS_MAXIMIZEBOX | WS_MINIMIZEBOX | WS_SYSMENU |
WS_VISIBLE;
    frame->Attr.Style &= ~(WS_CHILD | WS_THICKFRAME);

  // Assign icons for this application.
  //
  frame->SetIcon(this, IDI_SDIAPPLICATION);
  frame->SetIconSm(this, IDI_SDIAPPLICATION);

  SetMainWindow(frame);
```

```
}



//{{TSDIDecFrame Implementation}}


TSDIDecFrame::TSDIDecFrame(TWindow* parent, const char far*
title, TWindow* clientWnd, bool trackMenuSelection, TModule*
module)
:
  TDecoratedFrame(parent, title, !clientWnd ? new
TGalaxieDlgClient(0) : clientWnd, trackMenuSelection, module)
{
  // INSERT>> Your constructor code here.

}


TSDIDecFrame::~TSDIDecFrame()
{
  // INSERT>> Your destructor code here.

}


void TSDIDecFrame::SetupWindow()
{
  TDecoratedFrame::SetupWindow();
  TRect  r;
  GetWindowRect(r);

  r.bottom += 30;
  SetWindowPos(0, r, SWP_NOZORDER | SWP_NOMOVE);

  // INSERT>> Your code here.

}


//------------------------------------------------------
// TGalaxieApp
// ~~~~~~~~~~~
// Menu Help About Galaxie command
//
void TGalaxieApp::CmHelpAbout()
{
}



int OwlMain(int , char* [])
{
  TGalaxieApp   app;
  return app.Run();
}
```

GALAXAPP.H

```
//------------------------------------------------------------
//  Project Galaxie
//
//  Copyright © 1997. All Rights Reserved.
//
//  SUBSYSTEM:    Galaxie Application
//  FILE:         galaxapp.h
```

```
//  AUTHOR:
//
//  OVERVIEW
//  ~~~~~~~~
//  Class definition for TGalaxieApp (TApplication).
//
//----------------------------------------------------------------------
#if !defined(galaxapp_h)              // Sentry, use file only if
it's not already included.
#define galaxapp_h

#include <owl/opensave.h>


#include "galaxapp.rh"            // Definition of all resources.


//
// FrameWindow must be derived to override Paint for Preview and
Print.
//
//{{TDecoratedFrame = TSDIDecFrame}}
class TSDIDecFrame : public TDecoratedFrame {
  public:
    TSDIDecFrame(TWindow* parent, const char far* title, TWindow*
clientWnd, bool trackMenuSelection = false, TModule* module = 0);
    ~TSDIDecFrame();

//{{TGalaxieAppVIRTUAL_BEGIN}}
  public:
    virtual void SetupWindow();
//{{TGalaxieAppVIRTUAL_END}}
};    //{{TSDIDecFrame}}
```

```
//{{TApplication = TGalaxieApp}}
class TGalaxieApp : public TApplication {
  private:


  public:
    TGalaxieApp();
    virtual ~TGalaxieApp();

//{{TGalaxieAppVIRTUAL_BEGIN}}
  public:
    virtual void InitMainWindow();
//{{TGalaxieAppVIRTUAL_END}}

//{{TGalaxieAppRSP_TBL_BEGIN}}
  protected:
    void CmHelpAbout();
//{{TGalaxieAppRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(TGalaxieApp);
};    //{{TGalaxieApp}}


#endif  // galaxapp_h sentry.
```

GALAXIE.HPP

```
BOOLEAN ReadHeader(HANDLE hndFile, int *numframes, int *width,
int *numchan)
{
  DWORD BytesRead;
  char header[HEADERLENGTH];
```

```
  if (ReadFile(hndFile, &header, HEADERLENGTH, &BytesRead, NULL))
  {
    memcpy(numframes, &header[NUMFRAMESLOC], 4);
    memcpy(width, &header[WIDTHLOC], 4);
    memcpy(numchan, &header[NUMCHANLOC], 4);
    if ((*numchan != 1) && (*numchan != 3))
    {
      MessageBox(NULL,"Only grayscale or color images can be
displayed",
                                              "Unsuported
Feature", MB_OK | MB_ICONSTOP);
      return FALSE;
    }
  }

  return TRUE;
}


BOOLEAN DoOverlap()
{
  HANDLE ImgCut, ImgExpl, ImgDet, ImgThick, ImgOver;
  int numframes, width, numchan, curraddr;
  int count1, count2;
  unsigned char *CutData, *ExplData, *DetData, *ThickData,
*CombData;
  char SourceImage[255];
  DWORD BytesRead;

  switch (CurrAppStatus.ImgSource)
  {
       case Transmission            :
                        if (CurrAppStatus.ESource == Low)

                                strcpy(SourceImage,
XMISSIONLOWCUT);

                        else
                                strcpy(SourceImage,
XMISSIONHIGHCUT);
                           break;
    case Backscatter          :

                        if (CurrAppStatus.ESource == Low)
                                strcpy(SourceImage,
BSCATTERLOWCUT);
                           else
                                strcpy(SourceImage,
BSCATTERHIGHCUT);
                           break;
    case ForwardScatter       :

                        if (CurrAppStatus.ESource == Low)
                                strcpy(SourceImage,
FSCATTERLOWCUT);
                           else
                                strcpy(SourceImage,
FSCATTERHIGHCUT);
                           break;
    default                                                 :
        break;
  }
```

```
  //Open the explosives bitmap ELAS file
  ImgExpl = CreateFile(EXPLOSIVEBITMAP, GENERIC_READ,
FILE_SHARE_READ, NULL,

                       OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_SEQUENTIAL_SCAN,
                         NULL);
  if (ImgExpl == INVALID_HANDLE_VALUE)
  {
    MessageBox(NULL,"Error opening explosive bitmap image", "File
I/O error",
                                                    MB_OK |
MB_ICONERROR);
    return FALSE;
  }

  //Open the detonator bimap ELAS file
  ImgDet = CreateFile(DETONATORBITMAP, GENERIC_READ,
FILE_SHARE_READ, NULL,

                       OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_SEQUENTIAL_SCAN,
                         NULL);
  if (ImgDet == INVALID_HANDLE_VALUE)
  {
    MessageBox(NULL,"Error opening detonator bitmap image", "File
I/O error",
                                                    MB_OK |
MB_ICONERROR);
    return FALSE;
  }
```

```
  //Open the thickness bitmap ELAS file
  ImgThick = CreateFile(THICKNESSBITMAP, GENERIC_READ,
FILE_SHARE_READ, NULL,

                       OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_SEQUENTIAL_SCAN,
                         NULL);
  if (ImgThick == INVALID_HANDLE_VALUE)
  {
    MessageBox(NULL,"Error opening thickness bitmap image", "File
I/O error",
                                                    MB_OK |
MB_ICONERROR);
    return FALSE;
  }
  else
  {
      //Open the cut (pre-processed) image
    ImgCut = CreateFile(SourceImage, GENERIC_READ,
FILE_SHARE_READ, NULL,

                       OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN,
NULL);
    if (ImgCut == INVALID_HANDLE_VALUE)
    {
      MessageBox(NULL,"Error opening cut image", "File I/O
error",
                                                    MB_OK |
MB_ICONERROR);
      return FALSE;
    }
```

```
    else
    {
        //This step assumes that all ELAS headers are correct.
      //If there is a mismatch in any one, the program will yield
incorrect
      //results, and may even crash.
      ReadHeader(ImgCut, &numframes, &width, &numchan);
      ReadHeader(ImgExpl, &numframes, &width, &numchan);
      ReadHeader(ImgDet, &numframes, &width, &numchan);
      ReadHeader(ImgThick, &numframes, &width, &numchan);

      //Dynamically allocate memory for each of the images,
      //currently the memory buffer also includes the ELAS
header.
      CutData   = new unsigned char[(numframes+1) * width];
      ExplData  = new unsigned char[(numframes+1) * width];
      DetData   = new unsigned char[(numframes+1) * width];
      ThickData = new unsigned char[(numframes+1) * width];
      CombData  = new unsigned char[(3*numframes+1) * width];

      //Re-initialize the file pointers, to include the ELAS
header
                      SetFilePointer(ImgExpl, 0, NULL,
FILE_BEGIN);
                      SetFilePointer(ImgDet, 0, NULL,
FILE_BEGIN);
                      SetFilePointer(ImgThick, 0, NULL,
FILE_BEGIN);
      SetFilePointer(ImgCut, 0, NULL, FILE_BEGIN);

      ReadFile(ImgExpl,  ExplData, (numframes+1)*width,
&BytesRead, NULL);
      ReadFile(ImgDet,   DetData, (numframes+1)*width,
&BytesRead, NULL);
      ReadFile(ImgThick, ThickData, (numframes+1)*width,
&BytesRead, NULL);
      ReadFile(ImgCut, CutData, (numframes+1)*width, &BytesRead,
NULL);

                      ImgOver = CreateFile("overon.img",
GENERIC_WRITE, 0, NULL,

                                           CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

                      //Copy the ELAS header
                      for(count1 = 0; count1 < HEADERLENGTH;
count1++)
        *(CombData+count1) = *(ExplData+count1);

      //But indicate that this is a color file
      *(CombData+24) = 0x03;

      for (count1 = 0; count1 < numframes; count1++)
        for (count2 = 0; count2 < width; count2++)
        {
          curraddr = count2 + (count1+1)*width;

          if ( *(ExplData + curraddr) != 0x00 )
          {
            //Explosives overlap
            *(CombData+(3*count1+1)*width+count2) = 255;  //255
            *(CombData+(3*count1+2)*width+count2) = 0;
            *(CombData+(3*count1+3)*width+count2) = 0;
          }
```

```
          else
            if ( *(DetData + curraddr) != 0x00 )
            {
                 //Detonators overlap
              *(CombData+(3*count1+1)*width+count2) = 0;
              *(CombData+(3*count1+2)*width+count2) = 255;  //255
              *(CombData+(3*count1+3)*width+count2) = 255;  //255
            }
            else
              if ( *(ThickData + curraddr) != 0x00 )
              {
               //Thickness overlap
                *(CombData+(3*count1+1)*width+count2) = 255;
//255
                *(CombData+(3*count1+2)*width+count2) = 255;
//255
                      *(CombData+(3*count1+3)*width+count2) = 0;
              }
              else
              {
                //No overlap
                *(CombData+(3*count1+1)*width+count2) =
*(CutData+curraddr);
                *(CombData+(3*count1+2)*width+count2) =
*(CutData+curraddr);
                *(CombData+(3*count1+3)*width+count2) =
*(CutData+curraddr);
              }
        } //for loop (count2)

      WriteFile(ImgOver, CombData, (3*numframes+1)*width,
&BytesRead, NULL);
      CloseHandle(ImgOver);
      CloseHandle(ImgCut);
      CloseHandle(ImgExpl);
      CloseHandle(ImgDet);
      CloseHandle(ImgThick);

      //Free-up memory
      delete(CutData);
      delete(ExplData);
      delete(DetData);
      delete(ThickData);
      delete(CombData);

    }
  }

  return TRUE;
}
```

GLOBALS.H

```
//Filenames of images returned by the collection program
#define CHAN1COLLECTED                      "one.img"
#define CHAN2COLLECTED                      "two.img"
#define CHAN3COLLECTED                      "three.img"

//Filenames of low energy collected (unprocessed) images
#define XMISSIONLOWCOLLECTED  "v75a.img"
#define BSCATTERLOWCOLLECTED  "v75b.img"
#define FSCATTERLOWCOLLECTED  "v75c.img"

//Filenames of high energy collected (unprocessed) images
#define XMISSIONHIGHCOLLECTED "v150a.img"
#define BSCATTERHIGHCOLLECTED "v150b.img"
```

```
#define FSCATTERHIGHCOLLECTED "v150c.img"

//Filenames of low energy, pre-processed images
#define XMISSIONLOWCUT                      "cut_lt.img"
#define BSCATTERLOWCUT                      "cut_bs.img"
#define FSCATTERLOWCUT                      "cut_fs.img"

//Filenames of high energy, pre-processed images
#define XMISSIONHIGHCUT                             "cut_ht.img"
#define BSCATTERHIGHCUT
       "NOTAVAILABLE"
#define FSCATTERHIGHCUT
       "NOTAVAILABLE"

//
#define EXPLOSIVEBITMAP                             "result.img"
#define DETONATORBITMAP                             "and.img"
#define THICKNESSBITMAP
       "suspect.img"


//Compensation values to use
#define CORRVAL1LOW    0x000
#define CORRVAL2LOW    0x000
#define CORRVAL3LOW    0x000
#define CORRVAL1HIGH   0x000
#define CORRVAL2HIGH   0x000
#define CORRVAL3HIGH   0x000

#define HEADERLENGTH   28
#define NUMFRAMESLOC   12
#define WIDTHLOC                    20
#define NUMCHANLOC            24
```

```
//Global Structures
enum DetectorType {Transmission, Backscatter, ForwardScatter};
enum EnergySource {Low, High};
enum ProcessedType {None, PLow, PHigh, Both};
enum Overlap {On, Off};

struct StructAppStatus {
  DetectorType ImgSource;
  EnergySource ESource;
  Overlap             Over;
  ProcessedType TProcess;            //Indicates whether the
overlaped images have
  ProcessedType BProcess;            //already been processed
  ProcessedType FProcess;
  STARTUPINFO         SInfo;
  PROCESS_INFORMATION  PInfo;
} CurrAppStatus;

HANDLE hXray, hMotor, hDpib;
```

Vita

Panagiotis (Panos) Jason Arvanitis was born in 1972 in Athens, Greece. He graduated from Papagos 2nd Lyceum and moved to the United States in September 1990 for advanced studies. He received the degree of Bachelor of Science in Electrical Engineering from Virginia Polytechnic Institute and State University in December 1994. He continued his studies at the same university to obtain the degree of Master of Science in Electrical Engineering with a Computer Engineering option. He was employed in the Spatial Data Analysis Laboratory from May 1994 until July 1997.

In his spare time, Panos enjoys playing tennis, swimming, listening to music and driving in the countryside.