Spring 2000

A Digital Pressure Sensor Data Acquisition System in a Wind Tunnel Model

John J. Novakoski
Old Dominion University

A Digital Pressure Sensor Data Acquisition System In A Wind Tunnel Model

by

John J. Novakoski
B.S. Computer Engineering, December 1998, Old Dominion University

A Thesis submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

MASTER OF SCIENCE

COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY
MAY 2000

Approved by:

John W. Stoughton (Director)

James F. Leathrum, Jr. (Member)

Linda L. Vahala (Member)

ABSTRACT

A Digital Pressure Sensor Data Acquisition System In A Wind Tunnel Model

John J. Novakoski
Old Dominion University, 1999
Director: Dr. John W. Stoughton

Pressure measurements on wind tunnel models provide an important aid to overall aerodynamic analysis and design of aircraft and vehicles. Most pressure measurements in wind tunnels are made using analog pressure sensors with interfacing electronics that connect to an external data acquisition unit. Some of desirable features of an improved pressure measurement system are that it be: 1) model embeddable, 2) inherently digital in nature, 3) intelligent, and 4) controllable by a remote computer.

An intelligent, model-embedded, eight-channel digital pressure sensor system has been developed and tested in a wind tunnel. The implemented system consists of Micro-Electro-Mechanical System (MEMS) digital pressure sensors that are controlled by a small, Embedded Programmable Logic Device (EPLD)-based electronics module. This module outputs standard RS-232 signals that can be easily read and interpreted by a remote host computer.

This system has been tested in the 16 foot Transonic Tunnel at NASA Langley Research Center with a wireless and hard-wire data transfer system. This implementation requires minimal set-up time and provides a small, re-configurable and fully self-contained digital pressure measurement system. The sensors, electronics, and host configuration/control PC have performed according to the design specifications in the severe conditions of wind tunnel testing.

Details of the digital pressure sensor system and results of the tunnel and laboratory tests are presented in this thesis.

This thesis is dedicated to my wife, Michelle, and my two daughters,
Brooke and Bailey

ACKNOWLEDGMENTS

I would like to acknowledge the many people who have helped me during this project and throughout my graduate studies. First, I'd like to thank my advisor, Dr. Stoughton, for his guidance and his patience. Second, a special thank you to my mentor at NASA, Dr. Seun Kahng, without whose support none of this work would have been possible. Finally, I am indebted to the many people I have worked with at NASA for their advice and help, especially Eddie Adcock, Tommy Jordan, Richard Collins, Mike Holloman and Jim Bartlett.

Most importantly, though, I would like to thank my family. My wife, Michelle, has been a source of unwavering support and encouragement during both my undergraduate and graduate work. Both my wife and my daughter, Brooke, have been very understanding of the fact that I have had to split time between them and my studies. I thank them and the rest of my family for allowing me this opportunity.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE **PAGE**

SECTION ONE
INTRODUCTION

1.0 Wind Tunnel Models

Wind tunnel models provide an invaluable aid for the aerodynamic design of airplanes and spacecraft. These models are scale replicas of either the entire craft or portions of it. The models are put into wind tunnels where the wind speed and angle of attack of the model can be controlled and the effects studied.

In order for these models to be useful, however, accurate and reliable data must be gathered from them. Sensors need to be distributed in the model to measure parameters of interest. These sensors may measure either static or dynamic conditions. Typical types of measurements are pressure, temperature, angle of attack, flow and sheer stress. Many sensors, up to a hundred or more in some cases, need to be read quickly, so the data must be transferred to an off-model host computer and stored so that it may be analyzed at a later date.

1.1 Current Method and Associated Problems

The current method of acquiring pressure data involves measuring small signal voltage outputs from analog pressure transducers. A typical layout is shown in Figure 1.1. This method has some inherent shortcomings. First, the pressure transducers are typically located in a central, electronically scanned pressure (ESP) module. This necessitates the use of long runs of tubing from the actual pressure port to the ESP module location. These long tubes may develop leaks or may not accurately transfer the pressure from the pressure port location. Second, small voltage signals from the transducers are read outside the model over long distances of cabling in an electrically noisy environment. This noise can induce errors in the voltage measurements. Considering that the

---

This thesis style conforms to the *IEEE Transactions on Measurement and Instrumentation.*

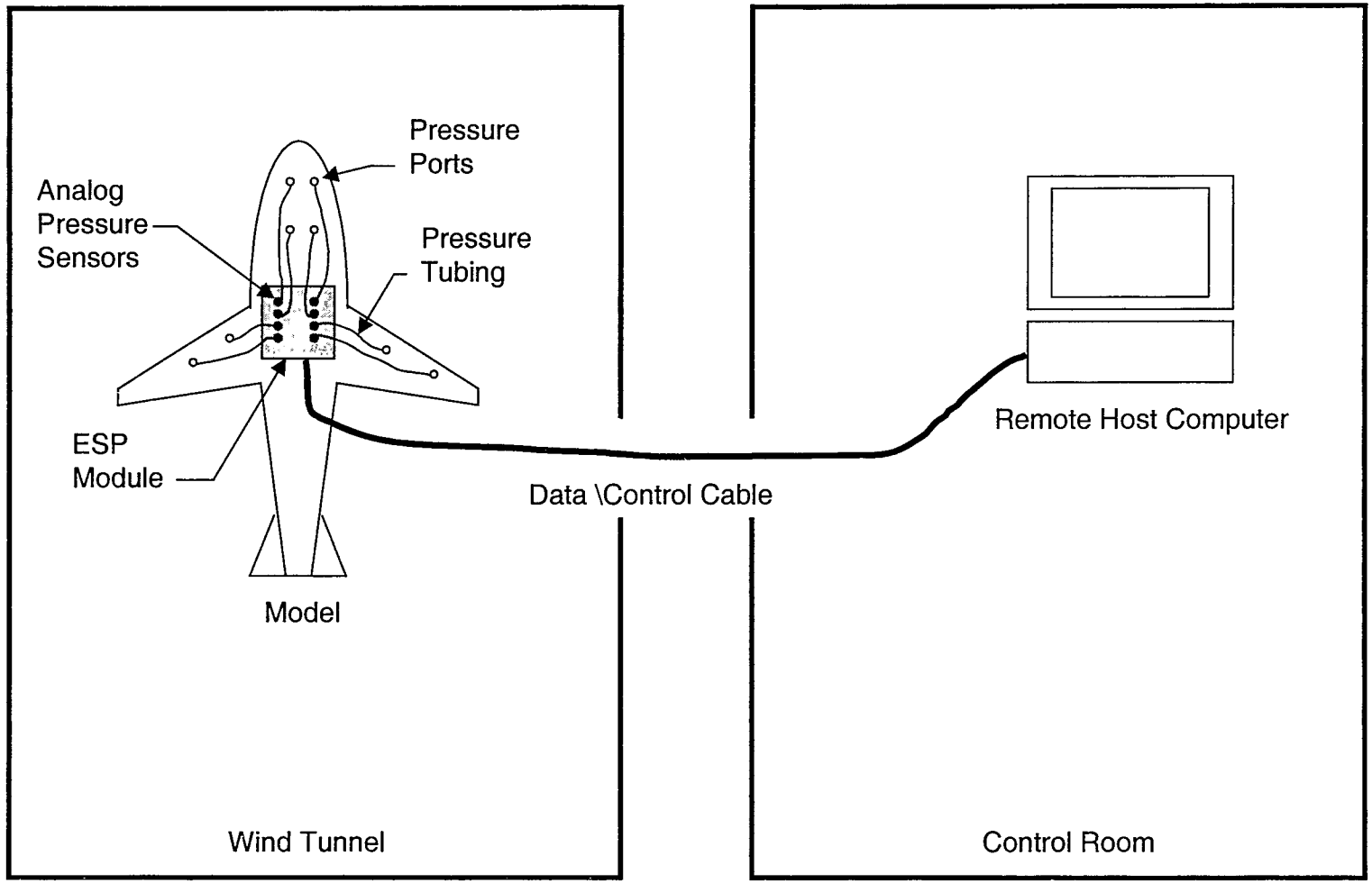


Figure 1.1 Typical Wind Tunnel DAS

signals of interest are typically in the millivolt range, even small levels of noise can be problematic. A design that can alleviate some or all of these problems while still maintaining the required functionality is needed.

1.2 System Objectives

Several modifications of existing techniques are desired for a new data acquisition system. First, an improved system would permit distributed sensor placement to allow analog to digital conversions to take place as close to the pressure source as possible. This would minimize the noise on the analog signals from the pressure transducers and would diminish or eliminate pneumatic problems associated with the runs of long pressure tubing currently used. Second, sensor interfacing and control electronics should be embedded in the model. This would decrease the complexity of needed external system components. Finally, the system should output data in a simple standard protocol. A serial communication protocol at a standard baud rate would allow the data acquisition system to be more readily used with other systems.

Another area of possible improvement is the communication link. New methods of data transfer should be explored to improve speed, signal integrity and ease of installation. Communication via optical fiber would allow faster transmission speeds and be less susceptible to EMI. Wireless RF communication would eliminate the need to install data cabling, thus decreasing installation time.

1.3 Implementation Overview

A new data acquisition system has been designed utilizing an embedded controller module. The data acquisition controller module is designed to receive data from 8 digital pressure sensors and to provide an RS-232 output. The module is contained on a single Printed Circuit Board (PCB) and is built around an Altera Embedded Programmable Logic Device (EPLD). VHDL, an IEEE standard hardware description language, was used to realize the desired digital

architecture on the EPLD. The system PCB is mounted in an aluminum box enclosure with external connectors for the sensor signal lines, the RS-232 output and power.

Additionally, each pressure sensor is mounted on its own, small PCB with its own clock. This enables distribution of the sensors allowing for placement at, or within close proximity to, the actual pressure port. A shielded cable with a 7-pin connector is wired to the PCB. This cable is used for the sensor signal lines as well as for bringing in power for the sensor and clock.

The output of the system is at RS-232 levels, and consists of eight 2-byte packets that correspond to the eight channels of pressure data. The 2 bytes are the 16 bit binary representation of a decimal number. The decimal number corresponds to an absolute pressure reading from one of the KP-100 sensors.

This system can communicate via hard-wire or can be used in conjunction with COTS RF or OF modules for wireless or optical fiber data transfer.

1.4 Thesis Overview

This thesis will describe the design, testing and implementation of a data acquisition system that addresses the issues discussed in this section. The new system layout is shown in Figure 1.2. The sections that follow will provide an in-depth discussion of all aspects of the system design. Section Two lists the specifications of the system and identifies and describes the various system components. Section Three encompasses the design approach, design implementation, selection of components and physical implementation of the data acquisition controller module. Section Four covers the experimental verification of the data acquisition system including testing and evaluation. Finally, Section Five provides a summary of the work done, assesses the effectiveness of the design, lists deficiencies and suggests possible solutions for improvement.

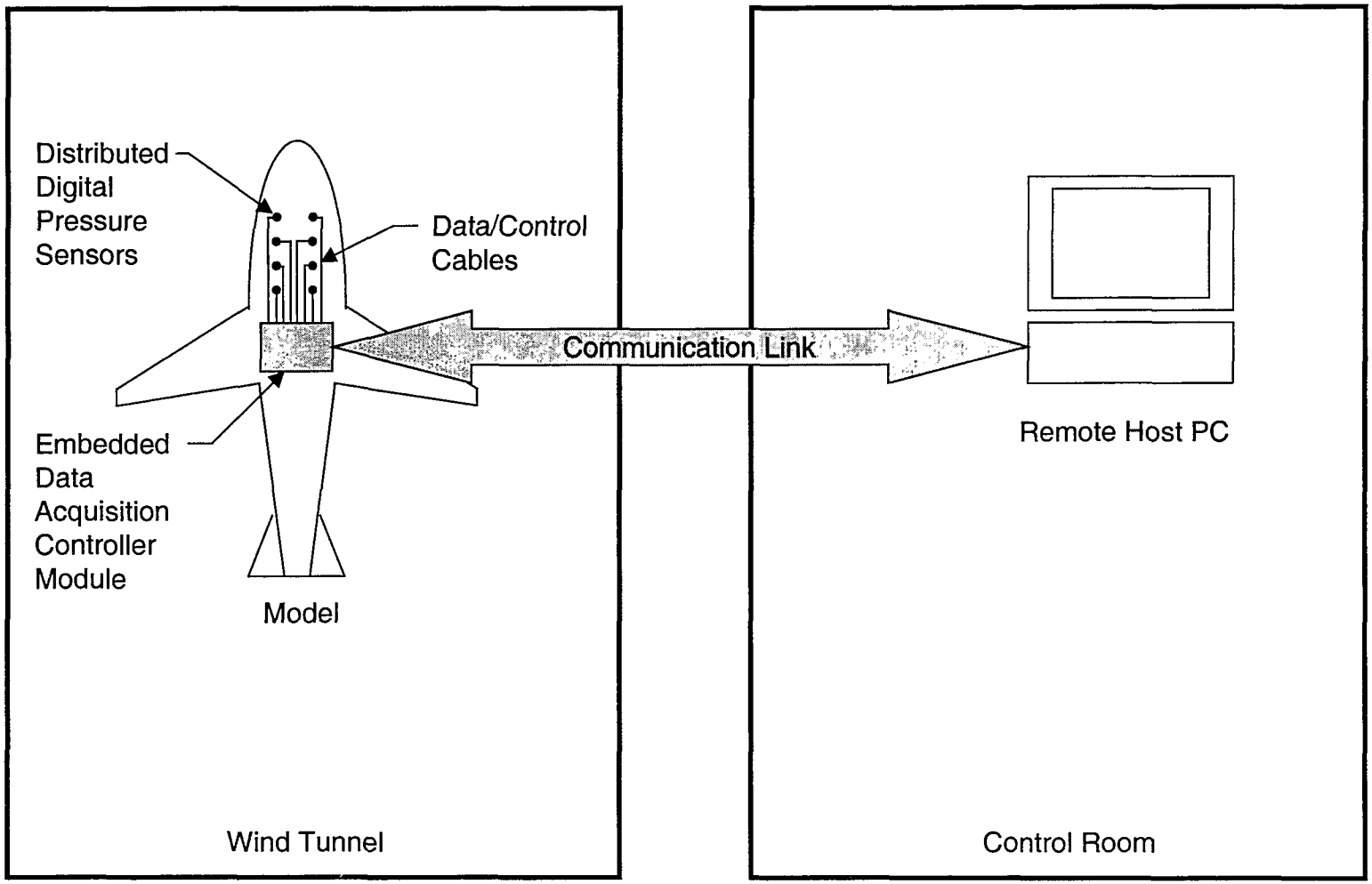


Figure 1.2 Proposed Wind Tunnel DAS

SECTION TWO
SYSTEM SPECIFICATIONS AND OVERVIEW

2.0 Introduction

The system specifications and the elements that comprise the system are discussed in this section. The initial sub-section lists the overall specifications that drove the design and gives justification for the choices that were made. The following sub-section describes, in detail, the individual components that make up the overall data acquisition system. The final sub-section covers the different modes of communication: wireless, optical fiber and hard-wire as well as addresses issues dealing with communication protocol.

2.1 System Requirements

The main design objective was to control and reliably receive data from digital pressure sensors in a wind tunnel model from an external host computer. Several things needed to happen for this to be successfully accomplished. First, sensor control should be handled by the on-board electronics. Second, the sensors and electronics needed to be small enough to be embedded in a model. Third, the communication link needed to perform reliably in the harsh wind tunnel conditions. Last, data needed to be acquired at high speeds to permit more useful analysis.

The original specifications for the overall system were very ambitious. They were modified somewhat as the design progressed to allow the systems involved to be completed on schedule. Because the purpose of this project was to show proof-of-concept for new sensors and communication schemes, the specifics of number of sensors and data transfer rates were not as important as producing a working system. Below is a list of the final required system performance specifications.

1. Number of pressure channels

   The actual number of sensors on hand as well as the overall desired size of the data acquisition module determined selection of the number of channels. The more sensors that were to be used, the more connectors would be needed on the electronics enclosure. This is important because connectors are one of the determining factors of overall component size. The final design was to take in eight channels of pressure data.

2. Output protocol

   A standard communication scheme was needed to allow ease of migration across systems. The RS-232 standard was selected because of its widespread use and simplicity. To accommodate other system needs, including multiple connections on the communication bus and the overall cable length required, an RS-485 data output was required.

3. Baud rate

   High-speed data transfer was desired. The limiting components were the addressable RS-232 to RS-485 converters that were needed so that two data acquisition modules could be on the same RS-485 bus. These converters could only be addressed at 9600 bps. To make the system simpler and ensure completion, a transmission speed of 9600 baud was selected. While this is a relatively slow data transfer rate, it was deemed sufficient for the static condition measurements that were to be taken for this particular test.

4. Temperature survivability

   Expected tunnel conditions were anticipated to be between 20 C and 75 C; thus components needed to be operational in this range.

5. Vibration survivability

   Because models are subjected to severe vibration forces of potentially up to 10 g or more, the components selected needed to be capable of withstanding such conditions.

6. Communication Modes

   Three different communication modes were to be tested: wireless Radio Frequency, Optical Fiber and hard-wire.

2.2 System Overview

An overview of the data acquisition layout can be seen in Figure 2.1. The complete system contains two separate data acquisition controller modules that control different types of sensors. This thesis deals only with the part of the system that is controlling and communicating with the digital pressure sensors. The system components that will be discussed in this sub-section are the sensors themselves, the controller module and the remote host computer and the software used on it.

2.2.1 Siemens Pressure Sensors

The Siemens KP-100 is a surface mount capacitive silicon absolute pressure sensor [1]. It provides a 16-bit digital output via a serial peripheral interface (SPI). The device has 8 pins with functions as shown in Table 2.1.

Each KP-100 sensor is on its own small PCB with an on-board 8 MHz oscillator, as shown in Figure 2.2. The remaining 7 signals are brought in via a 7-conductor shielded cable. The cables from the 8 pressure sensors are connected to 8 connectors on the electronics box. An internal wiring harness connects these individual connectors to a 40-pin header for the signal lines and to power and ground posts for the sensor power and ground lines, as shown in Figure 2.3.

Eight Siemens KP-100 digital pressure sensors were used for this design implementation. These sensors provide a 16-bit digital output that corresponds to the absolute pressure seen at the internal pressure transducer. The operational pressure range is from 60 to 130 kPa, or roughly 8.7 to 18.8 psia. In addition to the transducer, this sensor consists of a sigma-delta modulator, two stages of digital filtering, two shift registers and two clock dividers. Figure 2.4 shows a block diagram of the internal circuitry of the sensor.

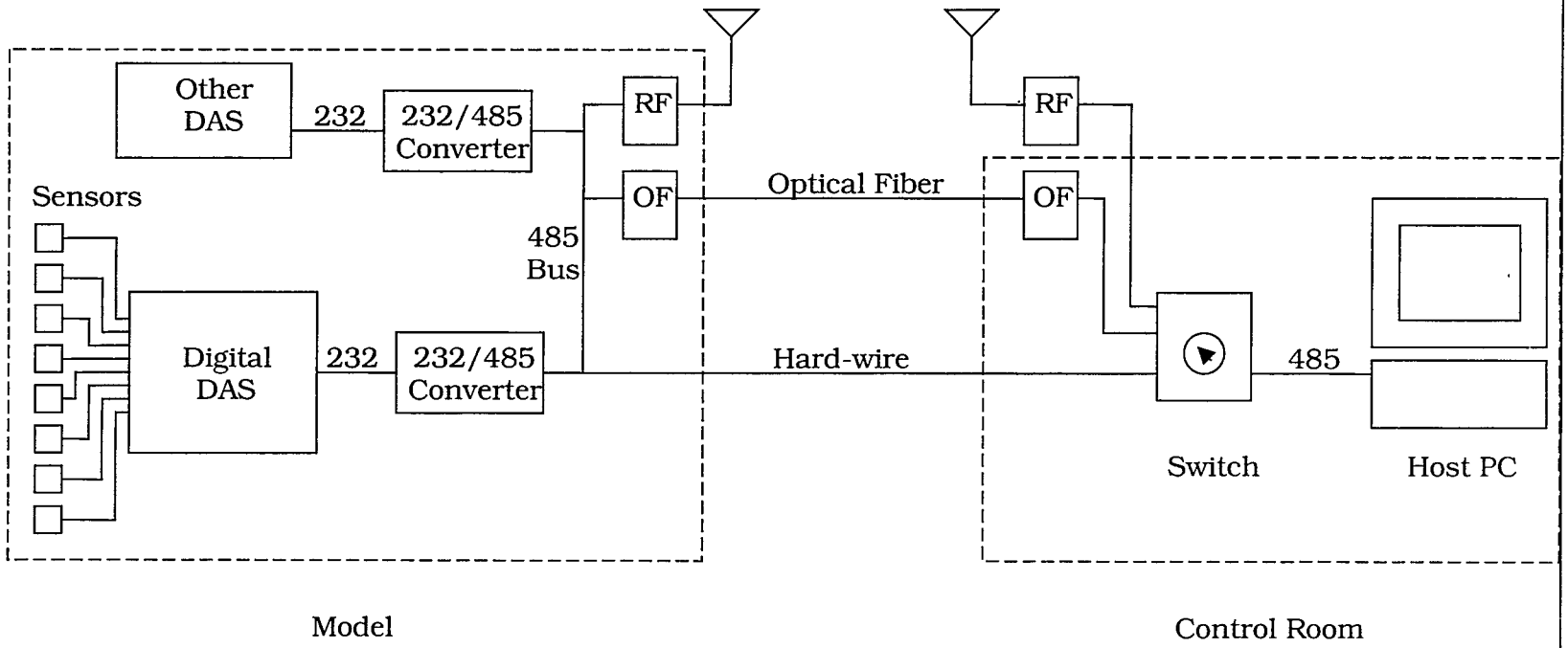


Figure 2.1 System Diagram

TABLE 2.1 Pin Description of KP-100 Pressure Sensors

Pin Number	Symbol	Function
1	CLKS	Input, clock for serial interface
2	CS	Input, chip select, active low
3	DTA_OUT	Output of the serial interface
4	CLK_IN	Input, external clock = 4/8 MHz
5	Vdd	5V power supply terminal
6	DTA_RDY	Data ready signal for serial output
7	DTA_IN	Input for serial interface
8	Vss	0V circuit ground potential

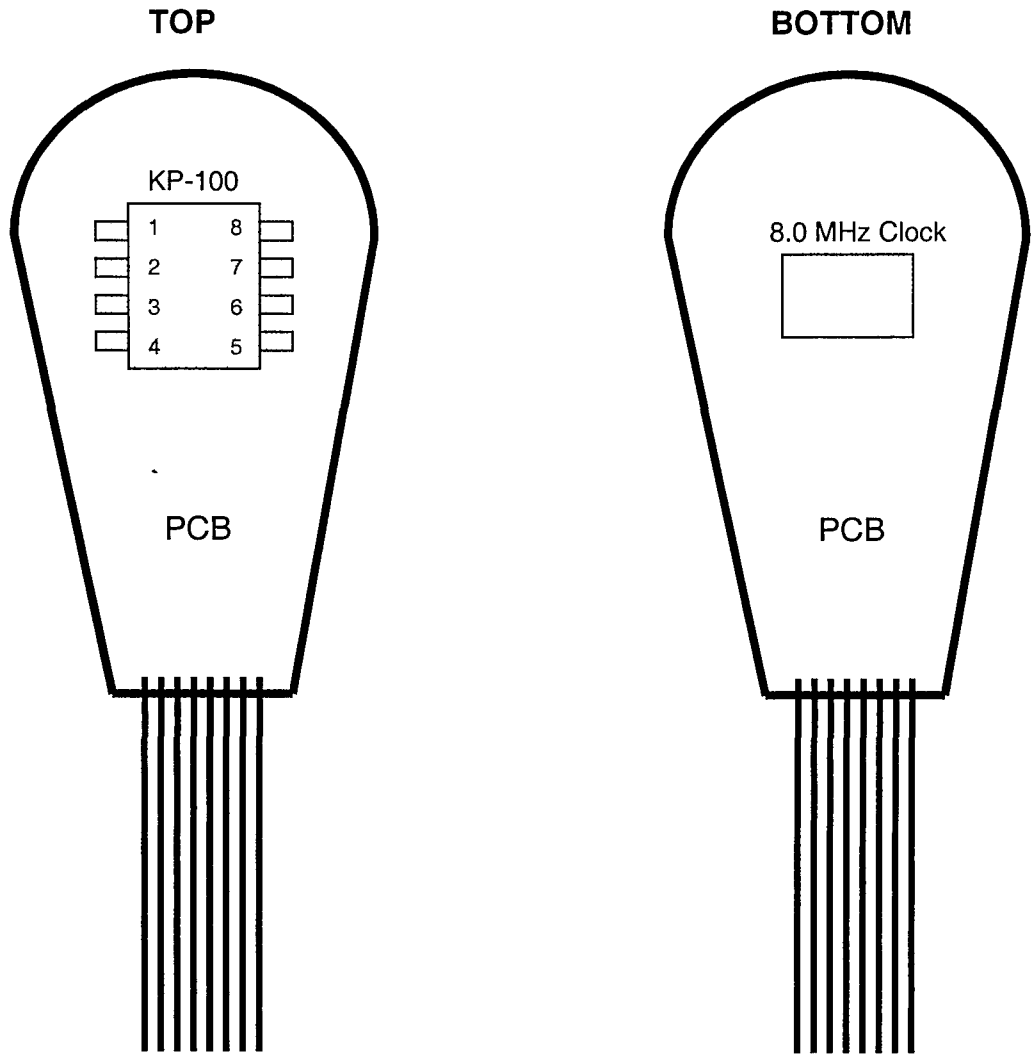


Figure 2.2 Sensor PCB

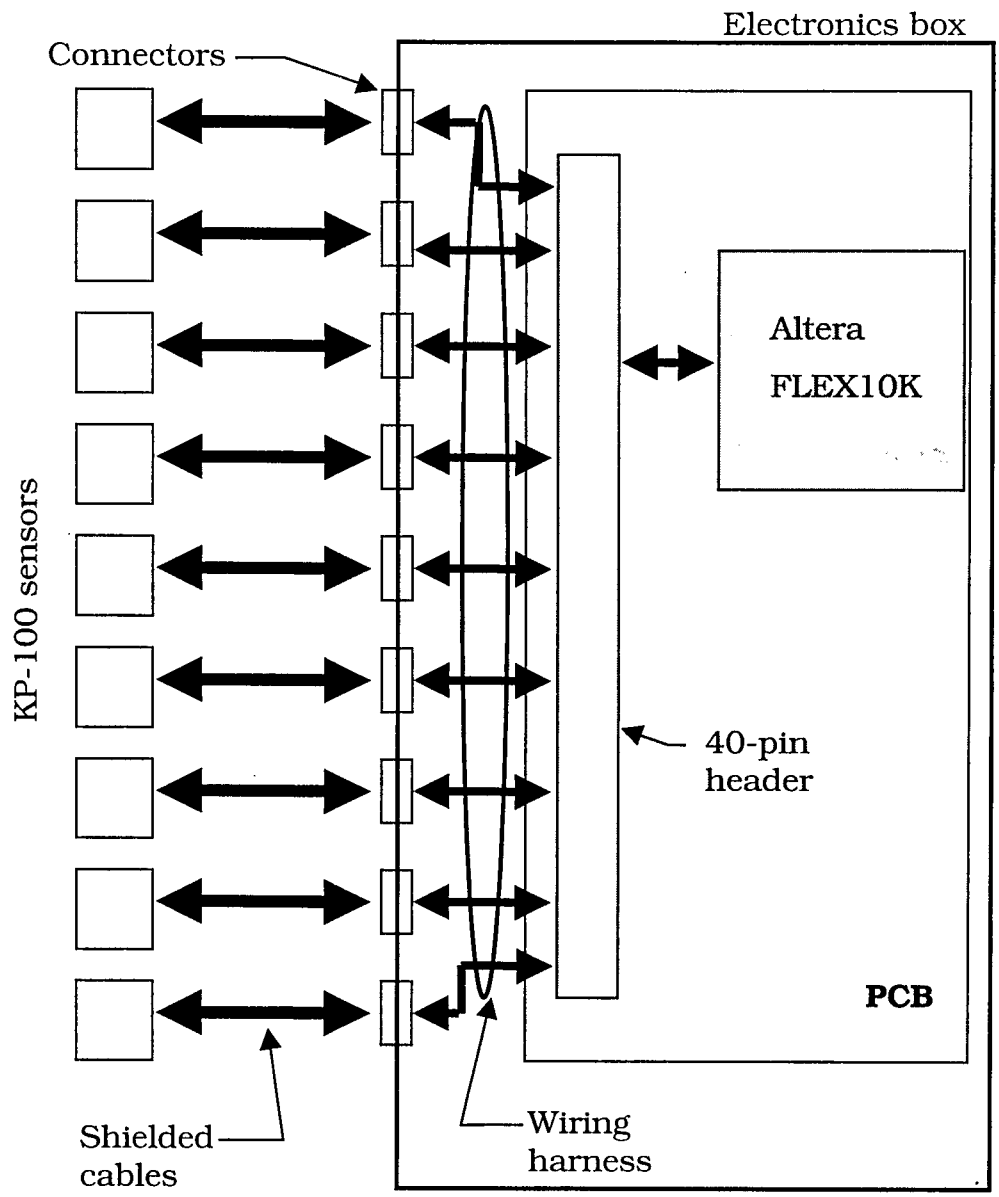


Figure 2.3 KP-100 Signals

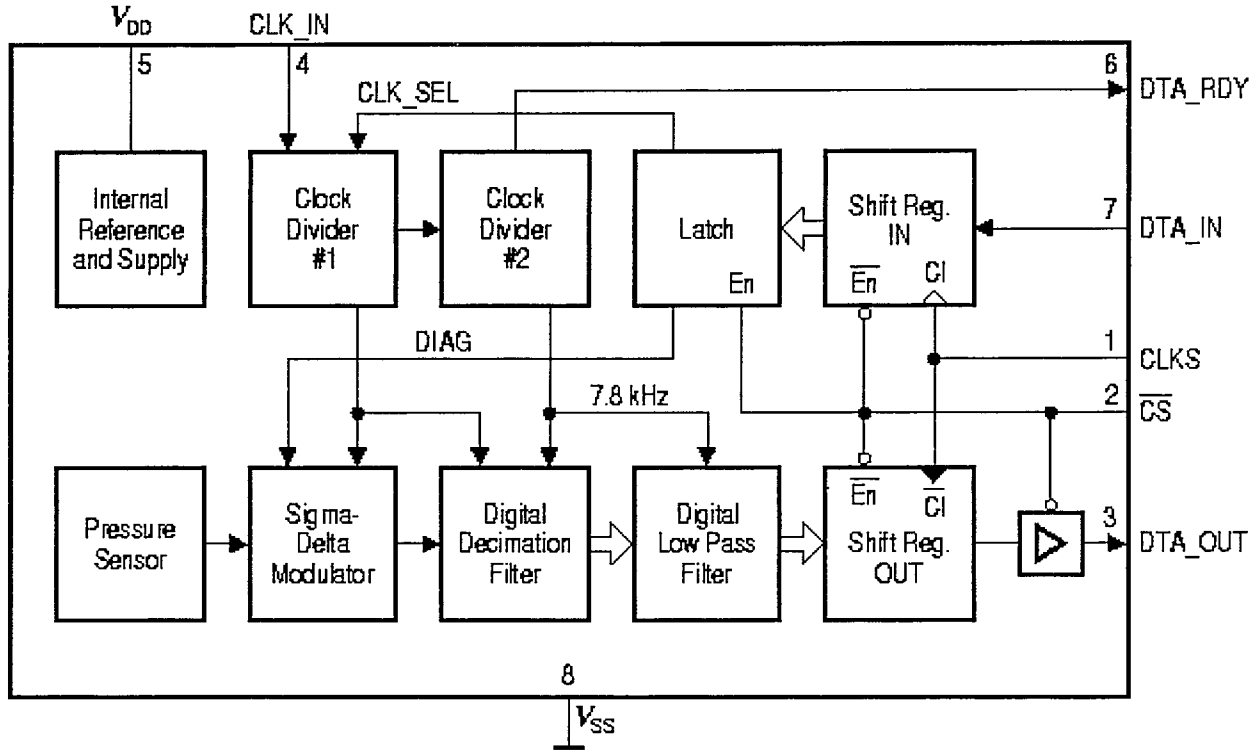


Figure 2.4 Internal Components of KP-100 Sensor

A more in-depth description of the signals to/from the KP-100 follows.

CLKS

This is a clock input that determines the rate of serial data transfer. After the CS line drops low, data is shifted out on the falling edges of this signal. Data is shifted in on this signal's rising edges. The maximum frequency at this pin is 500 kHz. Data is written to an internal shift register every 128 microseconds, so to ensure that all 16 bits of data have time to be shifted out of this register, the minimum frequency is 125 kHz.

CS

Chip select input that, when low, allows data to be shifted in or out. When high, the DTA_OUT pin will show high impedance and the serial interface registers will not shift. A rising edge on this pin latches the data into the input shift register.

DTA_OUT

This is the serial output from the device. After a falling edge of the CS signal, data is shifted out on falling edges of the CLKS line, least significant bit first. When CS is high, this pin will be at a high impedance state and the CLKS clocking signal is ignored.

CLK_IN

This pin receives a clock input of either 4 or 8 MHz. The default mode is 8 MHz but can be changed via the serial input interface. The device's internal clock of 500 kHz is derived off of this input.

DTA_RDY

A rising edge on this indicates that new data is available in the output shift register. This line is asserted every 128 microseconds. Any old data in the output register will be over-written, so a data read should not take be taking place when this occurs.

DTA_IN

Data is shifted into the input shift register from this pin on the rising edge of the CLKS signal when CS is low. However, the data is not latched for use by

the internal circuitry until the rising edge of CS. When this occurs, the last three values shifted in will be latched and used.

2.2.2 Data Acquisition Controller Module

The data acquisition module is responsible for acquiring data from the pressure sensors and sending out RS-232 data packets. This module consisted of a seven-layer EPLD-based PCB in an aluminum box enclosure with eight 7-pin micro-tech connectors for sensor signals, a DB-9 connector for RS-232 signals and a 7-pin Winchester connector for power and ground. The entire enclosure measured 2.75" x 4.25" x 1.0".

This is the only custom-made component. The majority of the research and design effort was put into realizing this part of the system. Section Three of this thesis is devoted to the design if this component.

2.2.3 Host Computer and Software

A Pentium-based computer running the Windows NT operating system was used as the host computer. National Instruments' LabView was used as the communication software. Data transfer was through an RS-485 port that was connected to a switch box and then to either an RF or OF transceiver or directly to the in-model RS-485 bus via a long shielded cable.

The LabView program was run on the host computer with data from both data acquisition systems displayed on the screen, updated once a second. At a data-taking point, a scan of all channels from each system was taken and saved. This was repeated as quickly as possible by the software for the specified number of repetitions. All data from a single point was saved into a single file that could be named automatically or by the user. Figure 2.5 shows the graphical user interface for the LabView program.

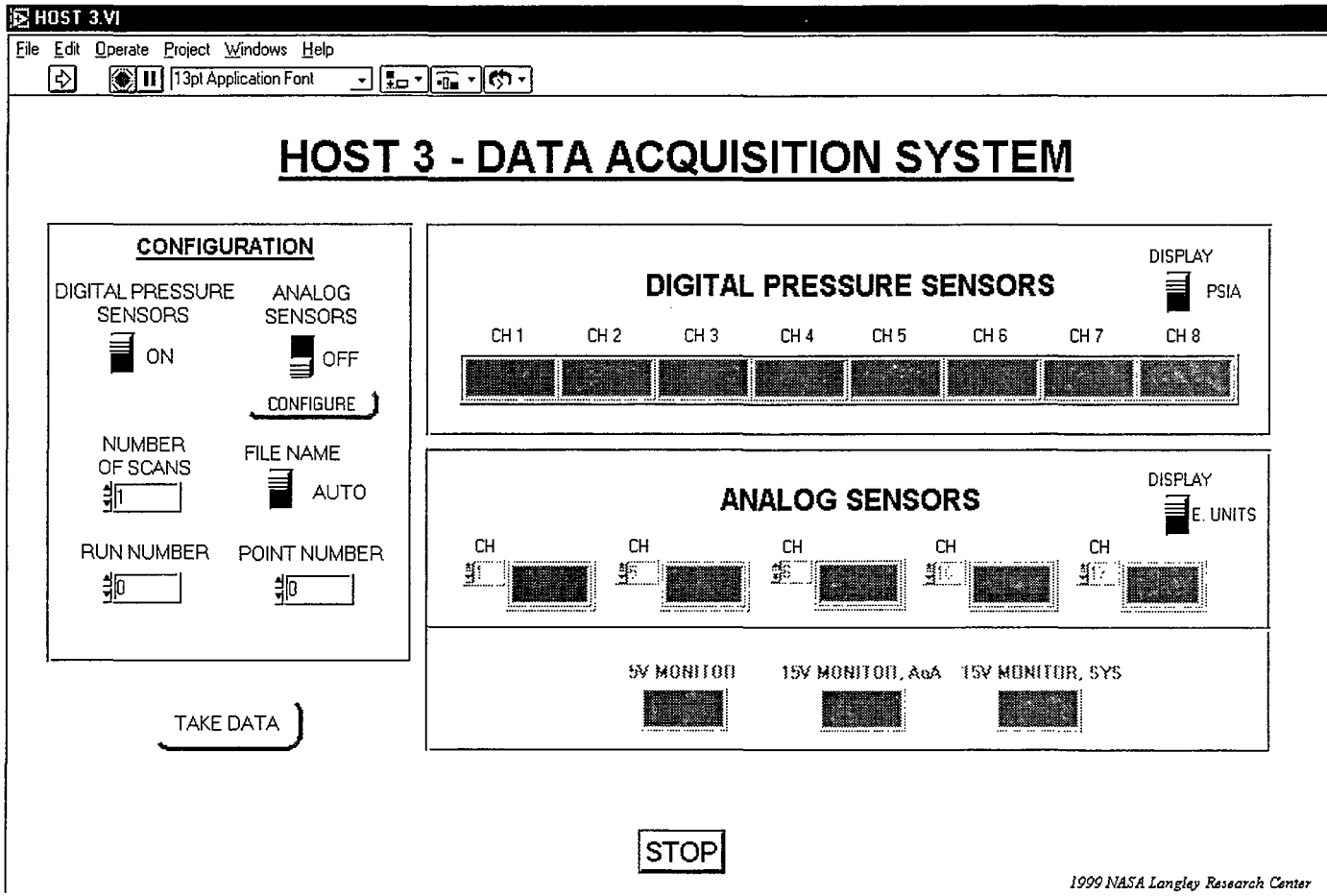


Figure 2.5 LabView Software Interface

2.3 Communication

Three different methods of communication were employed: wireless, optical fiber and hard-wire. A manual switch was used to change between the e different modes. All three communication schemes tapped into the RS-485 in-model bus. The system diagram, Figure 2.1, shows how these components are connected.

2.3.1 Wireless (RF) Modules

COTS RF transceivers were used to implement wireless telemetry capabilities. The transceivers used were the ADAM-4550 Radio Modem Modules made by Advantech. These modules have both an RS-232 and an RS-485 interface. Standard serial communication speeds are software programmable with a maximum transfer rate of 115.2 Kbps. The modules operate on a frequency of 2.442 GHz, Direct Sequence Spread Spectrum with a bandwidth of 22 MHz. Radio transmission power is 100 mW nominal [2].

The RF module was connected to the RS-485 bus in the model. The antenna was mounted outside the model on the sting, a structure to which the model is attached. The other antenna and RF module were mounted in the tunnel plenum about 15 feet from the model behind a plexi-glass window. An RS-485 cable was run from there to the control room.

RF communication performed as expected. Two minor issues surfaced when using this method. First, due to overhead induced by the RF modules themselves, the rate at which data could be taken was slightly diminished. The maximum number of samples per second dropped from 9 over hard-wire to 7 over RF. Second, data transfer was occasionally adversely affected when people or objects passed through the line of site of the antennas. The result was bad or dropped data. This behavior was only seen during set-up of the model, when technicians were moving about, not during the actual tunnel testing.

2.3.2 Optical Fiber (OF) Modules

COTS OF transceivers were used to enable data transfer across optical fiber. The units selected were Telebyte model 272A optoverters. These modules convert two-wire RS-485 signals for fiber optic transmission and can be used

with baud rates up to 2.5 MHz at distances of up to 2 kilometers [3]. One module was placed in the model on the RS-485 bus and the other in the control room.

In the lab, the maximum number of samples per second received was equal to that of the hard-wire link. Due to problems with power to the OF modules as well as with the optical fiber itself, verification of OF communication in the tunnel environment was not possible.

2.3.3 Hard-wire Connection

The third communication link was hard-wire. In this configuration a two-conductor shielded cable was run directly from the host computer's RS-485 port through the switch to the in-model RS-485 bus. This data path served two purposes. First, this was a back-up in the event the other two communications links failed. Second, it provided a baseline for the expected data rate. The data rate achieved from the RF and OF links could be compared to the hardwire data rate to determine transfer rate deviations between the different communication methods.

2.3.4 RS-232 Signals

The output of the system needed to conform to the RS-232 communication protocol [4]. An RS-232 level converting IC, Maxim's MAX232AEWE, was used to convert the 0 volt and +5 volt outputs of the PLD to the –10 volt and +10 volt signal levels required for RS-232. This IC required only a +5 volt supply. External capacitors and internal charge pumps generated the necessary voltages.

The data acquisition module was configured as a DCE device. Sensor data was sent out on pin 2 of the DB-9 connector on the electronics enclosure. This is the RD line for the attached DTE device, which in this case was the addressable RS-232-to -RS485 converter. Data was brought in from the converter on the TD line, pin 3. Figure 2.6 shows a complete view of the function of the RS-232 signals.

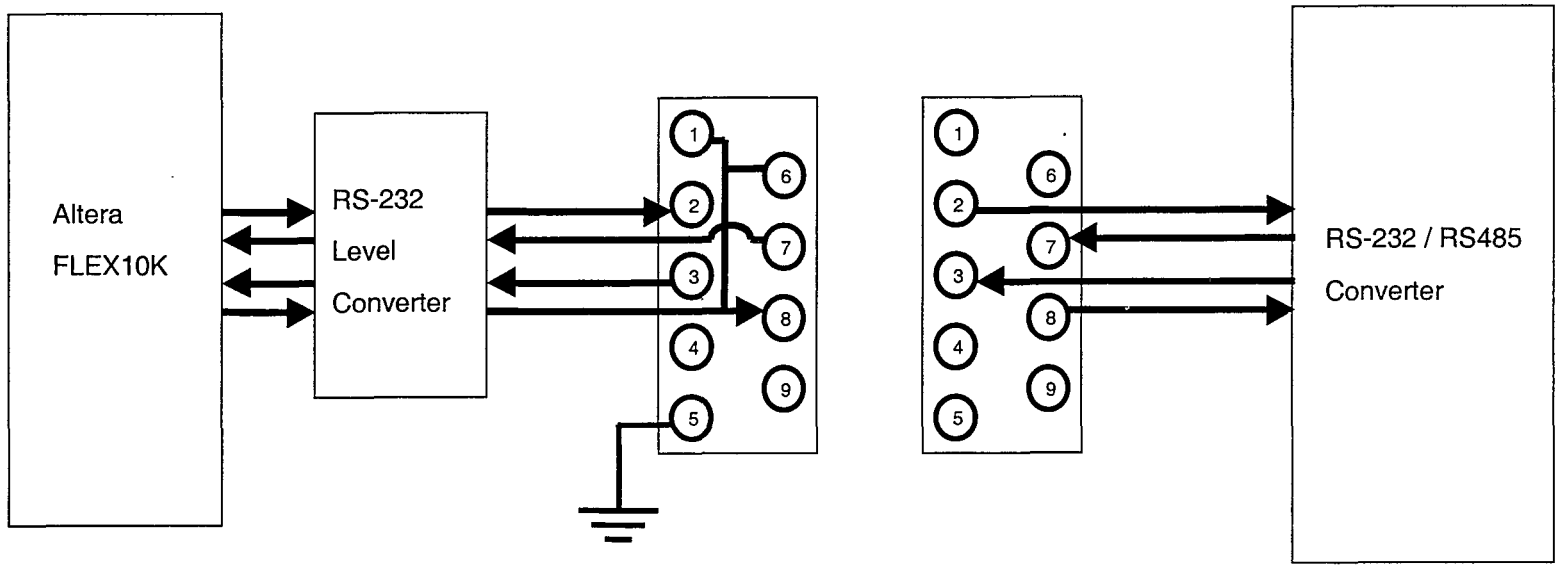
For each scan of the sensors, 8 data packets are sent out. A data packet consists of 2 bytes of data from the sensor with 1 start and 1 stop bit for each

byte. The LSB is sent, then the MSB. A diagram of one sensor data packet is shown in Figure 2.7.

2.3.5 Addressable RS-232-to-RS-485 Converter

Late in the design cycle the RS-232 output was deemed to be inadequate. Because the KP-100 digital system was one of two systems that shared common communication links, a protocol that enabled more than one system on a bus to be addressed was desired. The RS-485 protocol allows up to 32 addressable devices to be simultaneously connected on the bus. Another issue was cable length. Lengths in excess of 100 feet were needed. RS-232 is not recommended for cables longer than 50 feet. RS-485 is rated for cable lengths of 4000 feet [5]. This is the protocol that was ultimately used.

A converter was needed to convert the RS-232 output of the data acquisition module to the RS-485 that was required. The converter selected is addressable at 9600 baud and acts like an open switch when the "on" command is put out on the RS-485 bus with the appropriate address. This allows data to flow from the RS-232 device. The address can be changed with dip switches. An "off" command is put out on the bus to close the switch, preventing any data from flowing through the converter. This gives up control of the bus, allowing another device access.



1) Carrier Detect (CD)
2) Received Data (RD)
3) Transmitted Data (TD)
4) Data Terminal Ready (DTR)
5) Signal Ground (GND)
6) Data Set Ready (DSR)
7) Request To Send (RTS)
8) Clear To Send (CTS)
9) Not Connected (N.C.)

Figure 2.6 RS-232 Signals

Figure 2.7 RS-232 Data Packet

2.4 Summary

This Section has presented an overview of the different components that make up the data acquisition system as well as the specifications that drove the design. A detailed description of the each of these components was presented as well as a discussion of the different communication modes and protocols that were employed.

It is noted that certain system elements have already been presumed in the design, such as the FLEX10K. The design issues and approach for the EPLD-based data acquisition controller are presented in Section Three.

SECTION THREE
DATA ACQUISITION CONTROLLER MODULE DESIGN

3.0 Introduction

The design process, from initial ideas to the final hardware, will be covered in this section. The main design element of this system is the data acquisition controller, so the first sub-section of this section deals with the controller selection. The controller architecture was entered using VHDL code. The VHDL components that comprise the architecture are discussed in sub-section 3.2. Section 3.3 examines the control issues that were involved with system communication. Because this system was to be used in a specific environment, namely a model in a wind tunnel, certain physical constraints had to be met. These are discussed in sub-section 3.4. Finally, components and layout of the controller PCB are covered.

3.1 Controller Selection

A system component that could control the communication between the sensors and the host computer was needed. This controller needed to be small so that it could be model-embedded and flexible to allow for ease of design changes. Since this was to be an entirely digital system, the main design choices were PLDs, FPGAs, or microcontrollers.

3.1.1 Initial Controller Consideration

Several design alternatives were investigated to determine the best solution for meeting system specifications. Some of the options that were examined were Xilinx FPGAs, Altera PLDs and Motorola microcontrollers.

An early prototype utilized a Motorola 68HC11 microcontroller and an Altera 7064 CPLD. The CPLD was responsible for sensor communication with the microcontroller receiving the data via memory-mapped I/O and then sending it to a host computer through its serial port. This approach worked well as a prototype but was deemed excessive because the design goals could be met on a single device, a PLD.

3.1.2 Programmable Logic Device Selection

Ultimately, the Altera FLEX10K10TC144-4 EPLD was selected to be the main component of the data acquisition board [6]. Its selection was based on:

1. 144-pin TQFP package size and availability

This package is a surface mount chip that has 102 user I/O pins and measures approximately 20 mm x 20 mm x 1.5 mm. This package is commonly used and is readily available.

2. Internal gate count

It was important to have a sufficient number of gates to implement the current design as well as future design changes and additions. The FLEX10K10 has a maximum of 31,000 system gates.

3. Operating temperature range

Expected tunnel temperatures were between 20 C and 75 C. The commercial package was used which is operable from 0 C to 85 C.

4. Price

It was important to keep the overall cost of the electronics module down. The FLEX10K10 is relatively inexpensive, about $28.00 as of this writing.

The design fit easily on the FLEX10K10 device. In fact, only 27 % of the available circuitry was used . Of the 102 user I/O pins, 19 input and 16 output pins were used. Of the 16 used output pins, 11 were used solely for troubleshooting: 5 output pins were all that was required. The extra pins and space allow this device to be used for more complicated designs.

3.2 Data Acquisition Controller Architecture

The Altera FLEX10K EPLD is responsible for coordinating and controlling communication with the sensors and with the RS-232 device. Several different controllers, operating at different clock speeds, are used to accomplish this. Signals are routed internally between these controllers and other components on the Altera EPLD.

The individual architectural elements of the EPLD design, as shown in Figure 3.1, were configured using VHDL. VHDL component connection for the

design was accomplished with the schematic entry editor in Altera's MAX+Plus II. This method allows a more intuitive way of setting up and connecting components. The logical connection is specified by the design and is shown in Figure 3.2. Physical synthesis and layout are automatically handled by the MAX+Plus II software.

3.3 VHDL Components

In order to realize the architecture on the EPLD, a hardware description language needed to be used. VHDL, an IEEE standard hardware description language, was selected. The VHDL code was written, compiled and debugged using Altera's MAX+Plus II software.

On the FLEX10K EPLD, the design consists of controllers, shift registers, multiplexers, clock dividers, counters and a timer. VHDL code was written for each of the individual components. The components were connected with the schematic capture capability of the MAX+Plus II software. A detailed description of the individual components follows.

3.3.1 KP_100_CNTRLR

This controller is responsible for the interaction between the PLD and the 8 KP-100 sensors. It is a simple state machine with the appropriate outputs for each state. The controller is asynchronously reset to its initial state by a high RTS line. The RTS line is one of the RS-232 signals and indicates the status of a DTE device. This line should initially be high and drop low when the device is ready to receive data. This controller operates off of a 230 kHz clock with state transitions occurring on the rising edge. The clock is derived from the main clock of 1.8432 MHz.

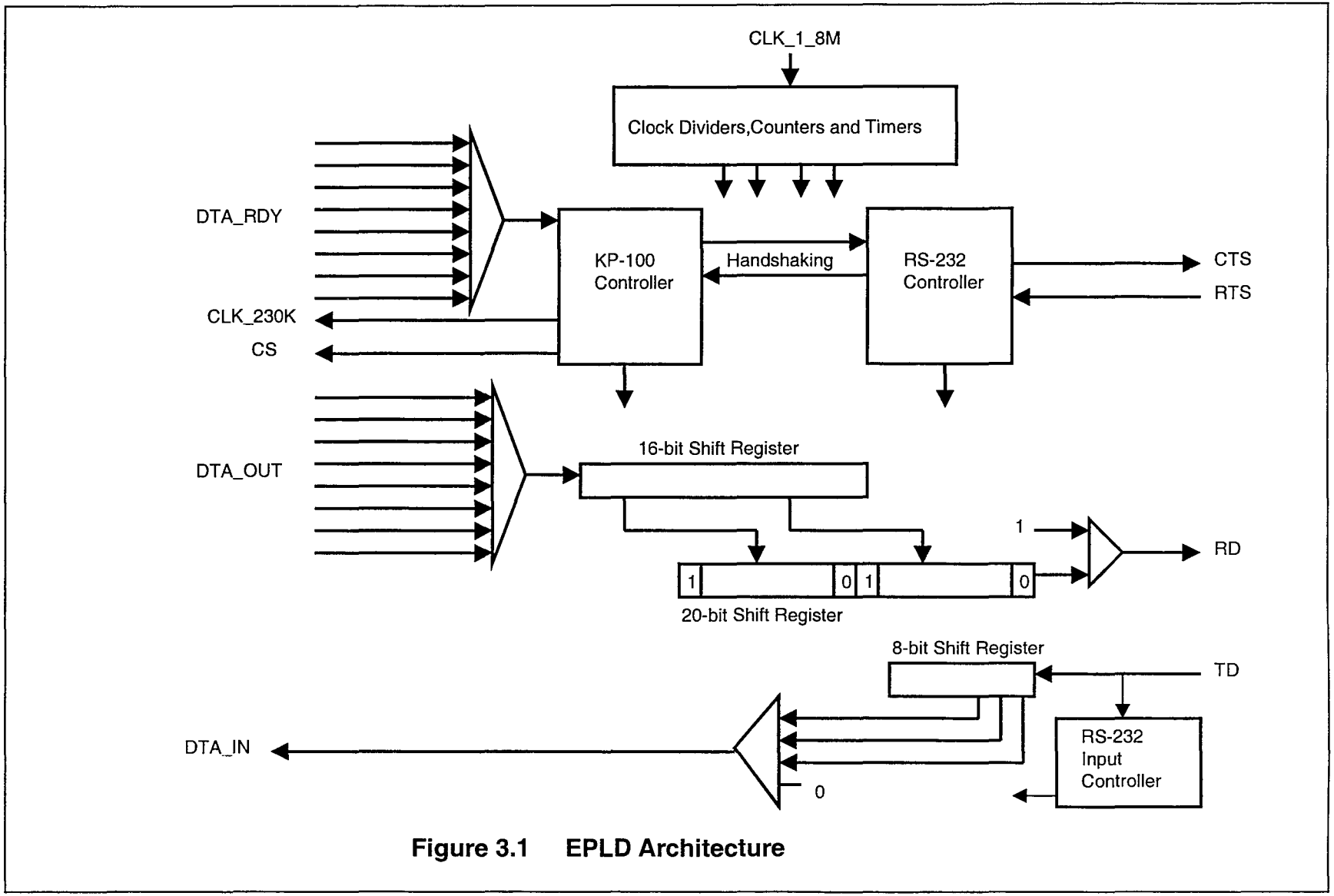


Figure 3.1 EPLD Architecture

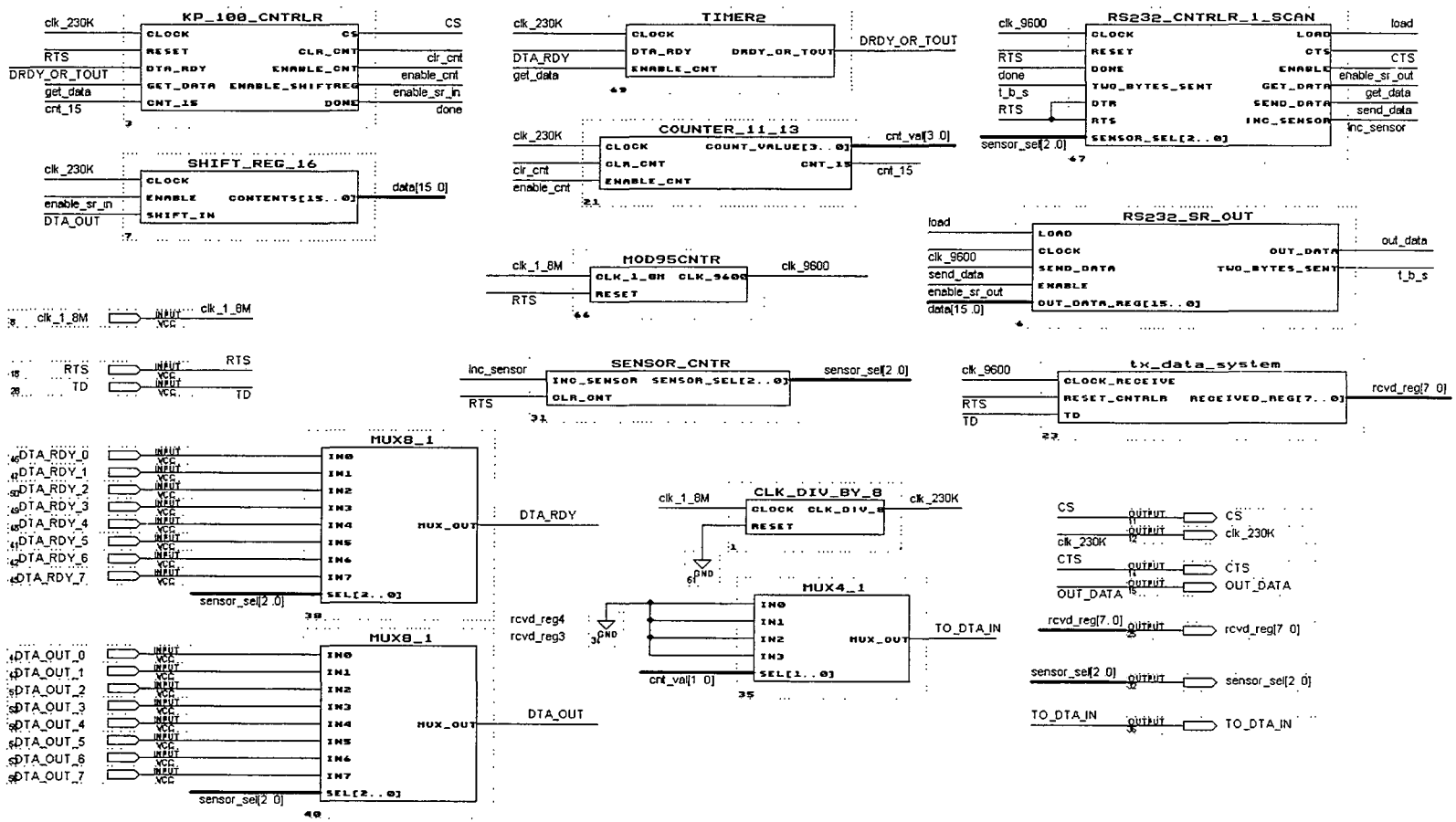


Figure 3.2 Internal EPLD Components and Signals

Timing information for the signals that are used by this controller can be seen in Figures 3.3a and 3.3b. A description of each of the states follows below.

State S0

This is the initial state. The active low CS line, the chip select line for the KP-100 sensors, is set high. A counter to keep track of the number of bits shifted in is cleared and disabled. A handshaking line, DONE, is set low.

Transition to state S1 occurs when both the DTA_RDY line from the selected sensor and the GET_DATA line go high. GET_DATA is a handshaking signal from the RS-232 controller that indicates that data has been requested by a DTE device.

State S1

CS line is dropped low to select a KP-100 sensor. The counter and a 16-bit shift register are enabled.

Transition to state S2 occurs when CNT_15 is raised. This indicates that all 16 bits of data from a KP-100 sensor have been shifted into the shift register, shift_reg_16.

State S2

CS line is raised, deselecting a sensor, and the DONE line is brought high. This raised DONE line signals the RS-232 controller that 16 bits of new data are ready to be sent out of the RS-232 port.

Transition to state S3 occurs after one clock cycle.

State S3

Outputs remain the same as in the previous state.

Transition to initial state, S0, occurs when GET_DATA drops low. This signal from the RS-232 controller indicates that the data has been loaded into the output register. The KP-100 controller can set up for the next reading.

See Figure 3.4 for an ASM chart of this controller component.

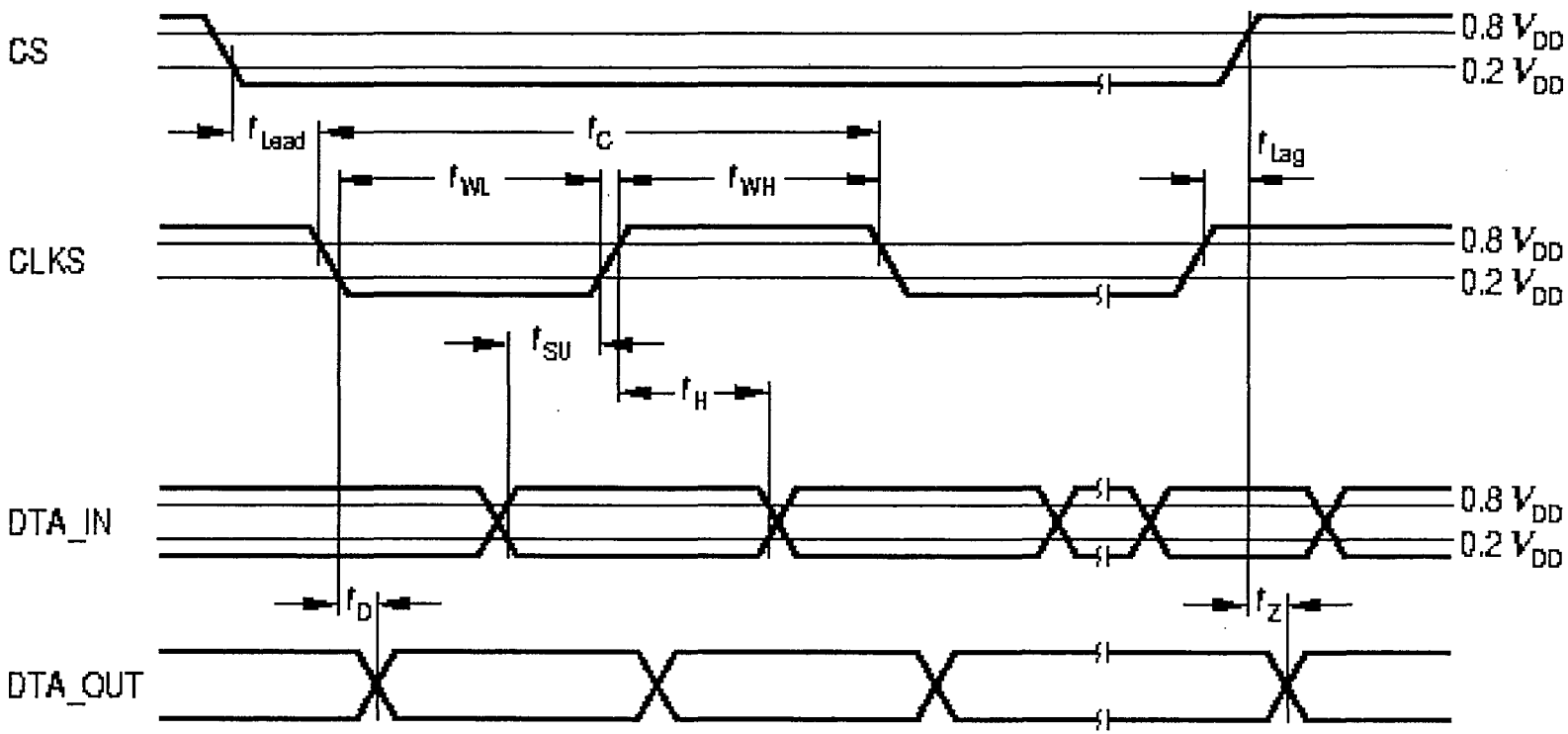


Figure 3.3 KP-100 Timing Diagram

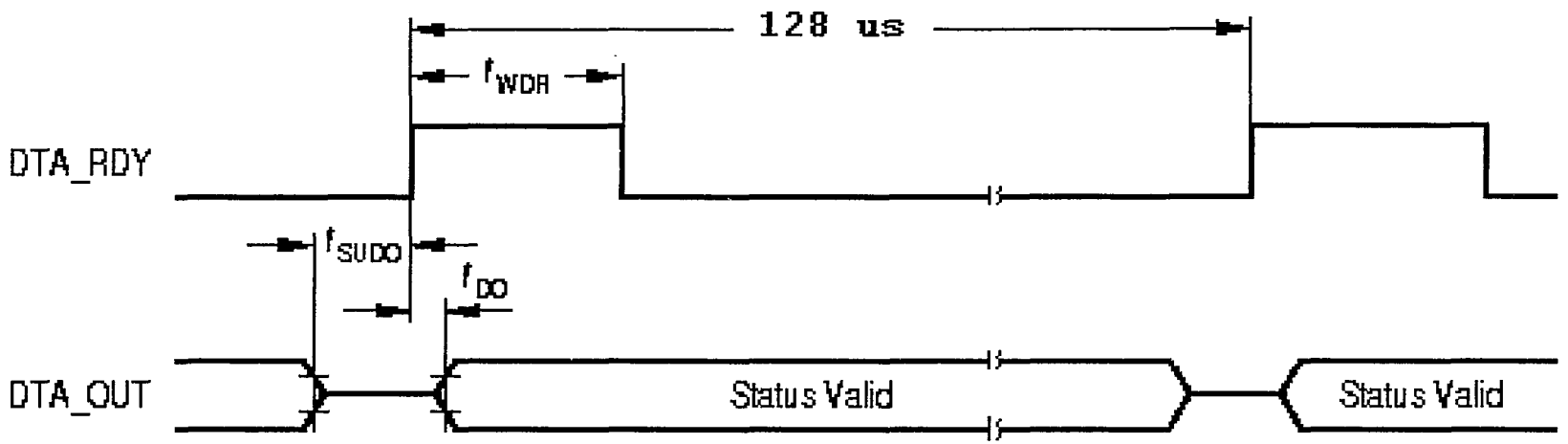


Figure 3.3a Data Timing - KP-100

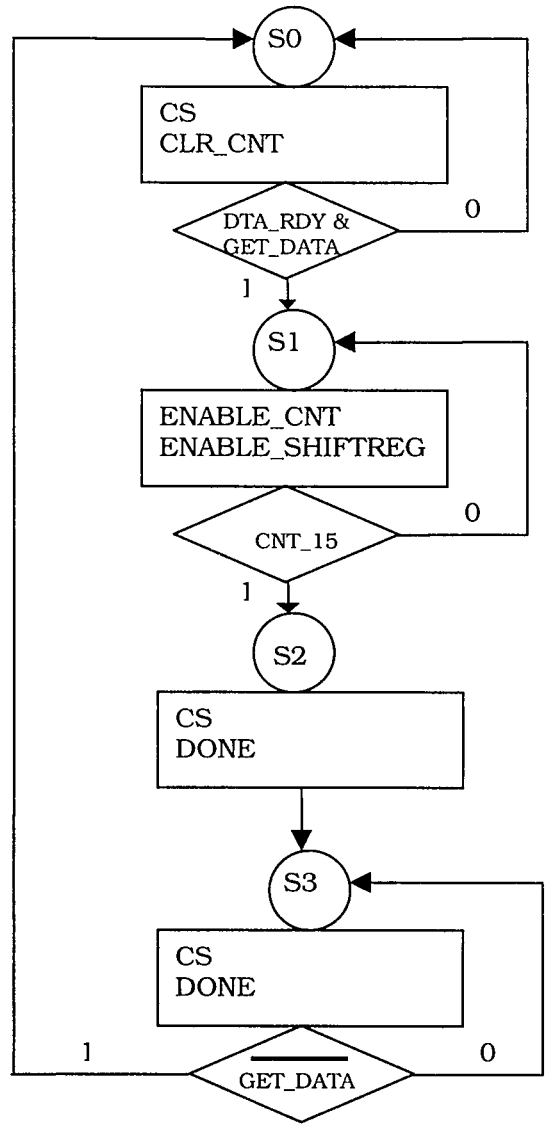


Figure 3.4 ASM Chart – KP_100_CNTRLR

3.3.2 RS_232_CNTRL_1_SCAN

This controller handles the I/O for the RS-232 port. Once the RTS line of the RS-232 port drops, indicating that a DTE device is ready to receive data, the controller communicates with the KP_100_CNTRLR controller to initiate a sensor reading. The output register is then loaded with the 2 bytes of sensor data and is shifted out through the RS-232 port at 9600 baud. A start bit and a stop bit are tacked on to each data byte. Only one scan of all eight channels of sensor data is transmitted. This process will be repeated after the RTS line of the RS-232 port is de-asserted and subsequently reasserted. The controller operates off of a 9600 Hz clock with state transitions occurring on the rising edge.

A description of each of the state of the controller follows below.

State S0

This is the initial state where the entire system is set up for a data request.

Transition to state S1 occurs when the RTS line of the RS-232 port is dropped, indicating a data request.

State S1

GET_DATA line is asserted. This signals the KP_100_CNTRLR controller to get data from one sensor.

Transition to state S2 occurs when the DONE signal from the KP_100_CNTRLR controller is raised to indicate that 16 bits of data are available in the input register.

State S2

GET_DATA line is dropped. INC_SENSOR line is asserted to increment the sensor counter that controls the mux select lines. LOAD line goes high to transfer the sensor data from the input register to the output register.

Transition to state S3 occurs if the RTS line of the RS-232 port is low.

State S3

CTS line is dropped to signal the receiving device that communication may begin.

Transition to state S4 occurs after one clock cycle.

State S4

ENABLE line is raised which enables the output shift register.

Transition to state S5 occurs after one clock cycle

State S5

SEND_DATA line is brought high. This starts the transfer of 20 bits out through the RS-232 port: 16 data bits, 2 start bits and 2 stop bits.

Transition to state S6 occurs when the TWO_BYTES_SENT signal is raised, indicating that both data bytes have been shifted out on the data line.

State S6

CTS line is raised to signal the end of the transmission. Output shift register is disabled.

Transition to state S7 occurs after one clock cycle.

State S7

Outputs are unchanged.

Transition to state S0 occurs if the SENSOR_SEL line is not raised. This allows the entire data taking and sending process to repeat for each of the eight sensors. Once the last sensor's data has been sent, the controller will remain in state S7 until the controller is reset by a high RTS line.

Figure 3.5 shows the ASM Chart for this controller.

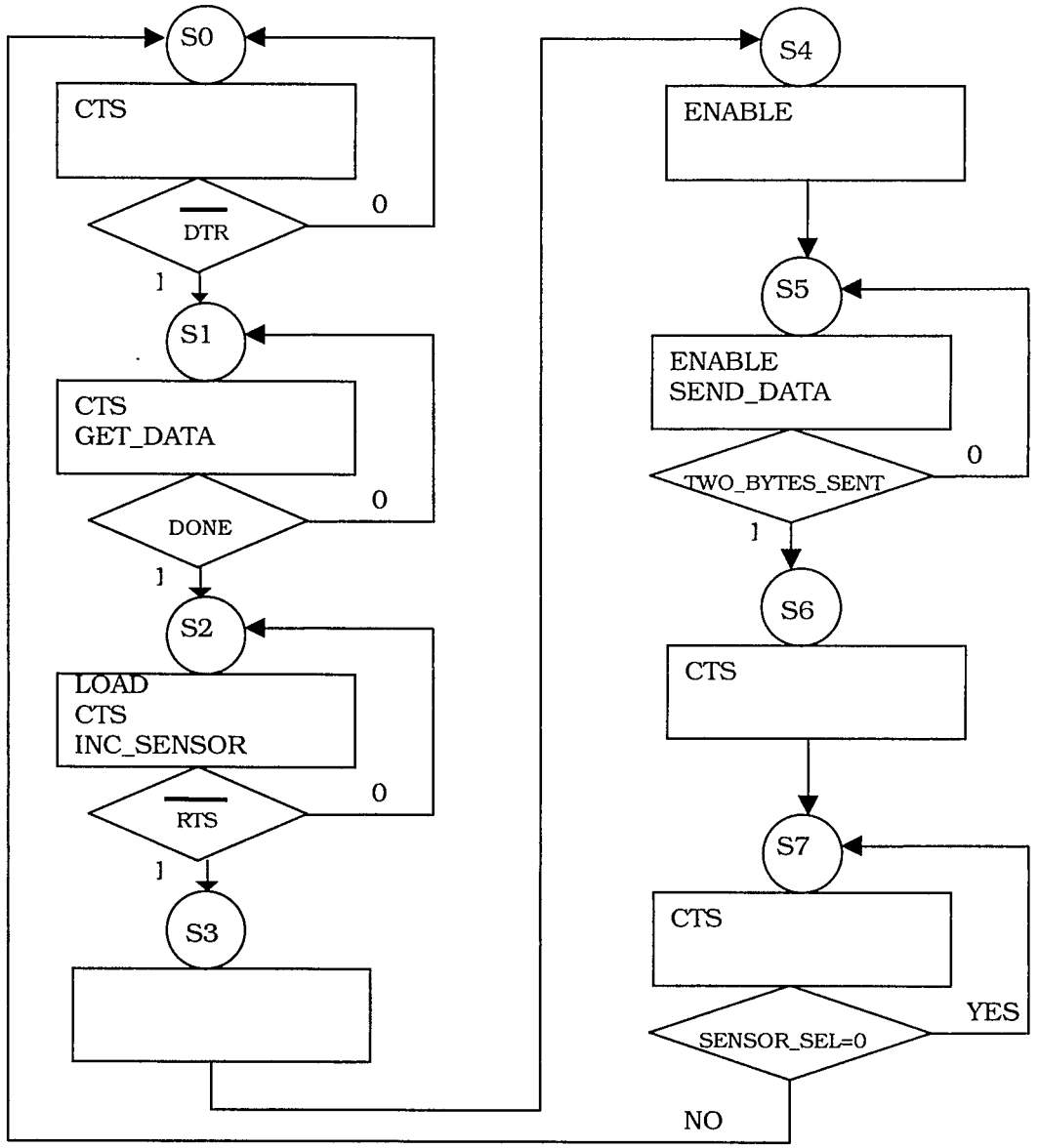


Figure 3.5 ASM Chart - RS_232_CNTRLR_1_SCAN

3.3.4 TIMER2

The KP-100 controller waits for the DTA_RDY signal from a sensor before data is transferred from that sensor. If there is a problem with the sensor and the DTA_RDY signal is never asserted, the controller will be hung up in that wait state indefinitely. The TIMER2 module prevents this from happening. Under normal conditions the DTA_RDY signal is asserted every 128 microseconds. A timer is set to go off after 277 microseconds, allowing enough time for 2 DTA_RDY signals to be received. The KP-100 controller will look for either the DTA_RDY signal from the sensor or the timer signal before initiating a data transfer. Once data transfer begins the timer is cleared. It should be noted that if the timer signal causes the state transition, the data that will be read in from the sensor will be unusable.

3.3.5 SHIFT_REG_16

This is the input shift register. It takes in the 16 bits of data from the KP-100 sensors at 230 Kbps. The KP-100 controller enables this register during a data transfer. The output of this register is connected to the output shift register.

3.3.6 CLK_DIV_BY_8

The main clock oscillator on the PCB is a 1.8432 MHz clock. This module divides that main clock to produce a 230 kHz clock that is used by the KP-100 controller, the timer, a counter and the sensors themselves for the serial data output. The VHDL code for this component was taken from a hardware description language textbook [7].

3.3.7 COUNTER_11_13

This is a 4-bit counter for keeping track of the number of data bits transferred in. The lower 2 bits are also used as mux select lines for output data to the KP-100 sensors, for mode control.

3.3.8 MOD95CNTR

A 9600 Hz clock is needed to drive the RS-232 I/O components. The main EPLD oscillator is a 1.8432 MHz clock. This clock signal is divided to achieve the 9600 clock by using a mod 95 counter. The counter outputs a signal

that is asserted every time the counter rolls over. This signal clocks an inverting flip-flop. The resulting signal from this flip-flop is a 9600 Hz clock signal.

3.3.9 SENSOR_CNTR

This counter is used to generate mux select lines to multiplex in the DTA_RDY and DTA_OUT signals from the appropriate sensor. This counter is incremented by one on the rising edge of the INC_SENSOR signal. The counter can also be cleared.

3.3.10 MUX8_1

This component is an 8-to-1 multiplexer for bringing in the signals from the 8 KP-100 sensors. Select lines come from the SENSOR_CNTR.

3.3.11 MUX4_1

This is the 4-to-1 multiplexer used to send mode control signals to the KP-100 sensors. Select lines come from COUNTER_11_13 module. Appropriate values are taken from the RS-232 input register and put out on the DTA_IN line to the sensors. The last 3 values on that line before the rising edge of the CS signal are latched into the sensor to set the sensor mode.

3. 3.12 TX_DATA_SYSTEM

This component handles the input from the RS-232 port. The TD line of the RS-232 port is monitored for the presence of a start bit. When detected, eight bits are shifted into the input register. Two of these bits can be used to set the KP-100 sensors into one of four operating modes. A mode of "00" is the normal mode while the other three modes are for diagnostic purposes.

The TX_DATA_SYSTEM component is made up of three components: RS232_INPUT_CNTRLR, COUNTER_3BIT and SHIFT_REG_INPUT .

3.3.12.1 RS232_INPUT_CNTRLR

This controller is used for coordinating the transfer of input data from the RS-232 port. It controls a 3-bit counter and an 8-bit shift register. State transitions occur on the rising edge of a 9600 Hz clock signal. A high RTS line resets the controller.

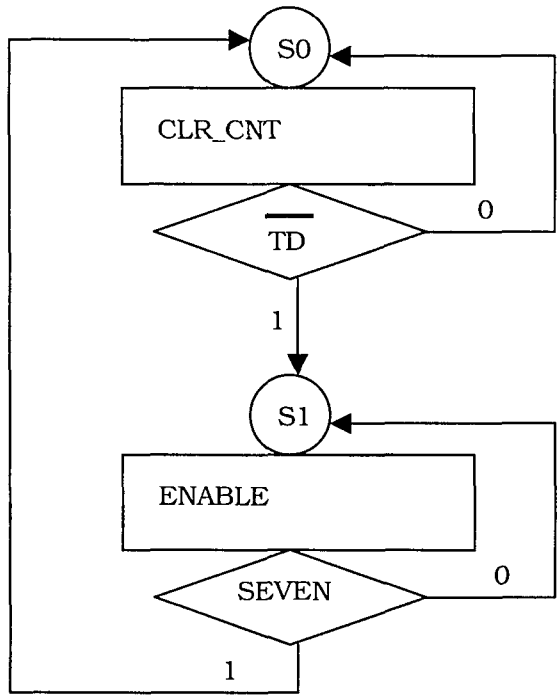


Figure 3.6 ASM Chart - RS232_INPUT_CNTRLR

A description of each of the states follows below.

State S0

This is the initial state. Control signals are output to clear and disable the 3-bit counter and disable the RS-232 input shift register.

Transition to state S1 occurs when the RS-232 line, TD, drops low. This is caused by the transmission of a start bit and indicates that data bits from the DTE will follow.

State S1

In this state, data is read into the input shift register from the TD line.

Transition to state S0 occurs when the signal, SEVEN, is received from the 3-bit counter. This indicates that all eight bits of data have been shifted in.

3.3.12.2 COUNTER_3BIT

This module is a 3-bit counter with a clear and an enable line with an output signal that is asserted when the maximum value, 7 , is reached. It is used to signal the input controller, RS232_INPUT_CNTRLR, that eight bits of data have been shifted in.

3.3.12.3 SHIFT_REG_INPUT

This component is an 8-bit shift register with asynchronous reset and synchronous enable. It is used to shift in data from the DTE device from the RS-232 port.

3.3.13 RS232_SR_OUT

This is the 20-bit output shift register with additional external circuitry. When the register is loaded, the two stop and two start bits are loaded along with the 16 data bits from a sensor. The output of this component is the RD line of the RS-232 port. A multiplexer is used to select either the shift register output or a '1' when data is not being transmitted. '0's are shifted in from the left while the data is shifted out. Since the original left-most bit of the register is a '1', the second stop bit, the only time this register is all '0's is when all of the data has been shifted out. This condition signals the RS_232_CNTRL_1_SCAN controller that both data bytes have been sent out of the RS-232 port.

3.6 Control issues

Several control issues emerged as the design progressed. Among these were communication rate and protocol, addressability of multiple systems and data packet format.

For reasons already discussed, RS-85 communication was used. The two-wire option of RS-485 that was implemented only allows half-duplex communication. The addressable RS-232/485 converters needed to be addressed to be "turned on" and then "turned off" before another converter could be addressed. Because of this, both systems on the bus, once addressed, would send one scan of all channels of data and then stop transmitting. Once the host computer received all the data or enough time had elapsed so that data should have been received, the host would "turn off" the converter and address the other converter. This single scan of data method was needed to ensure that not more than one device would attempt to drive the RS-485 bus at a time.

A standard serial communication data packet was selected: 1 start bit, 8 data bits and 1 stop bit. Again, the baud rate was 9600.

Table 3.1 shows the required steps for receiving data through this system.

Table 3.1 Control Flow of Data Acquisition System		
Ste	**Action**	**Description**
1	Host asserts RTS Line	Indicates host is ready to receive/send data
2	System asserts CTS	Indicates system is ready to receive/send data
3	System waits for DTA_RDY signal from sensor	Indicates that new data is available from the sensor
4	System drops CS lir and reads in data c DTA_OUT line	16 bits at 230 Kbps
5	System sends data o on RD line of RS-23 port	16 data bits 1 start bit, 1 stop bit , no parity (per byte) Baud rate is 9600 RS-232 levels
6	Set up for next sensor	Repeat steps 3 through 5 for remaining 7 sensors
7	Wait for next request	System waits for RTS line to be de-asserted then returns to initial state

3.5 Physical implementation issues

Several things needed to be considered with regard to system placement into an actual model. The main issue was size. Wind tunnel models are scale models and as such, are relatively small. Available space for any on-board electronics is at a minimum. Therefore an electronics package needs to be as small as possible while still performing the necessary functionality.

3.5.1 PCB Size

All of the components used on the main PCB were selected based on size as well as performance. These components were laid out on the PCB to minimize area. The resulting PCB was seven layers, including the ground and power planes, and measured approximately 2 " x 3 ".

3.5.2 Enclosure

The data acquisition PCB needed to be enclosed for protection from the environment and so that connectors could be wired and connected to the board. Again, size was an issue. An aluminum Pomona box just slightly larger than the PCB was selected. This enclosure allowed the PCB to be securely fastened inside as well as allowing for the placement of external connectors and internal wiring. The removable top made making internal changes, like swapping EPROMs, easier and less time-consuming.

3.5.3 Sensor Packaging

Pneumatic and electrical packaging of the sensors was another issue that needed resolution. The KP-100 sensors came in a plastic dual small outline flat package. The individual sensors needed to be placed in an airtight container and have their own 8 MHz clock. This was accomplished by placing each sensor on its own PCB that had an on-board clock. A cap with a small section of metal tubing was epoxied on top of the sensor to the board. Figure 3.7 shows how the sensors were packaged. This configuration allowed short pneumatic tubing to run from pressure ports to the caps containing the sensors. Because each sensor was on its own PCB with its own clock, the sensors could be placed in

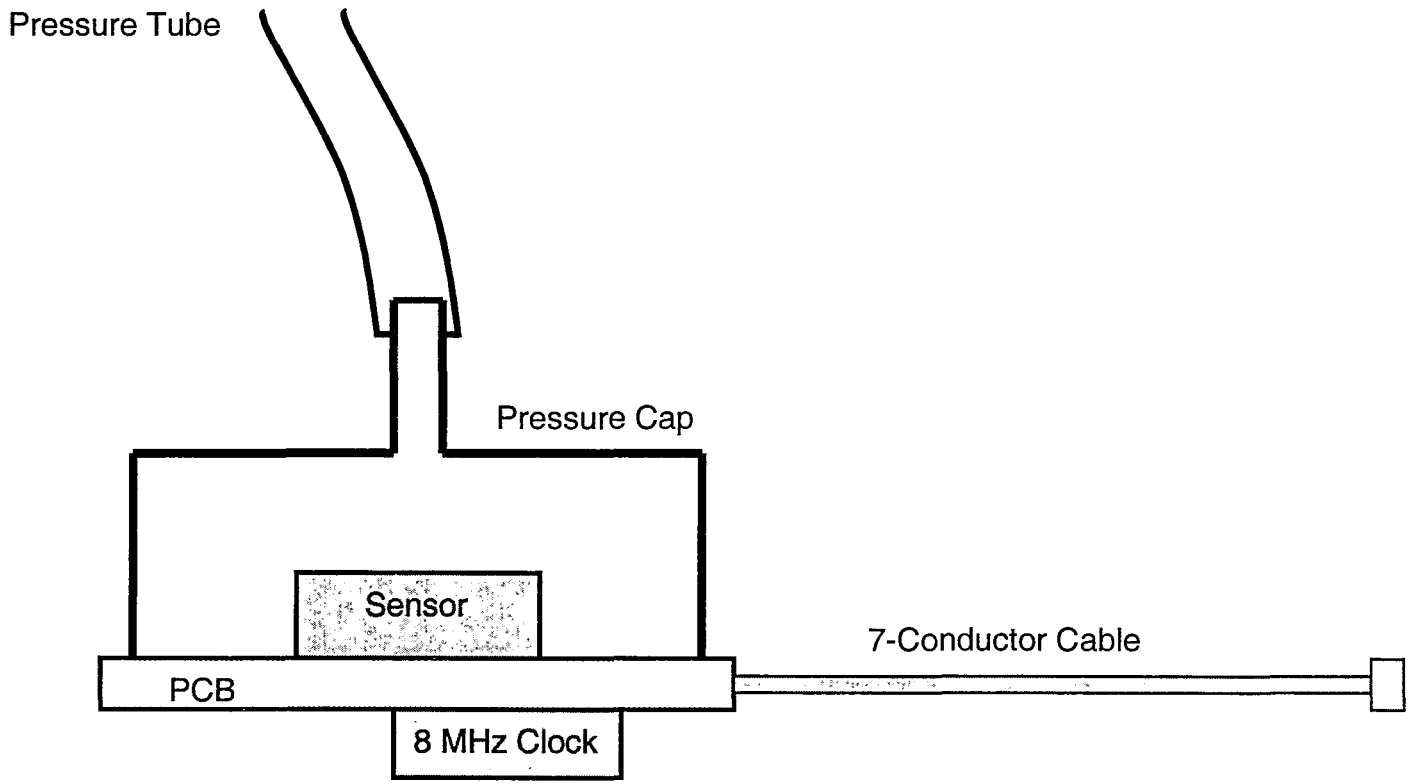


Figure 3.7 Sensor Packaging

close proximity to the pressure ports. Shielded cables carried the remaining seven signals to the electronics enclosure.

3.6 PCB issues

The main signals on the PCB are between the 40-pin header and the PLD for the sensor lines and between the PLD and the 10-pin header via the RS-232 level converter for communication through the RS-232 port. There are several other components on the PCB that are connected to the PLD. Figures 3.7 and 3.8 show how the PCB components are connected and laid out.

The main system clock is a 1.8432 MHz clock oscillator. The output signal of this clock is connected directly to a reserved clock pin on the PLD. The 230 kHz clock for serial communication with the sensors and the 9600 Hz clock for RS-232 communication are both derived internally from this main clock. These derived clock signals are also used by many of the internal components.

The Altera FLEX10K EPLD used must be reconfigured every time it is powered up. Therefore, an external EPROM, Altera's EPC144-1, is connected to the PLD. This EPROM, which is one-time programmable, is programmed with the .pof configuration file produced by the MAX+Plus II software. Upon power-up, the configuration data is serially transferred to the PLD. This process takes less than 320 milliseconds.

Because this was basically a prototype system, an extra 10-pin header was added to allow for PLD reconfiguration with the ByteBlasterMV parallel port download cable. A change in jumper settings changed the configuration signals path from the EPROM to the ByteBlasterMV header. This enabled the testing of different designs to evaluate performance before programming the EPROM.

A few of the signals to the sensors, namely the 230 kHz clock, the chip select lines and the data input lines, required that 8 devices be driven from single PLD outputs. The FLEX10K outputs are unable to drive this many devices. To enable this functionality, a buffer/line-driver IC, 74ACT125, was used with each of the affected outputs.

3.7 Summary

This section has described the design process that was undertaken to realize a functional data acquisition controller module. Controller selection was discussed including initial ideas. The architecture of the design was addressed and detailed descriptions of all VHDL design elements were added. Issues relating to control of the data flow, physical implementation and PCB concerns were brought up here as well. The next section will deal with how this design was verified and the testing that was done on it before actual use in a wind tunnel environment.

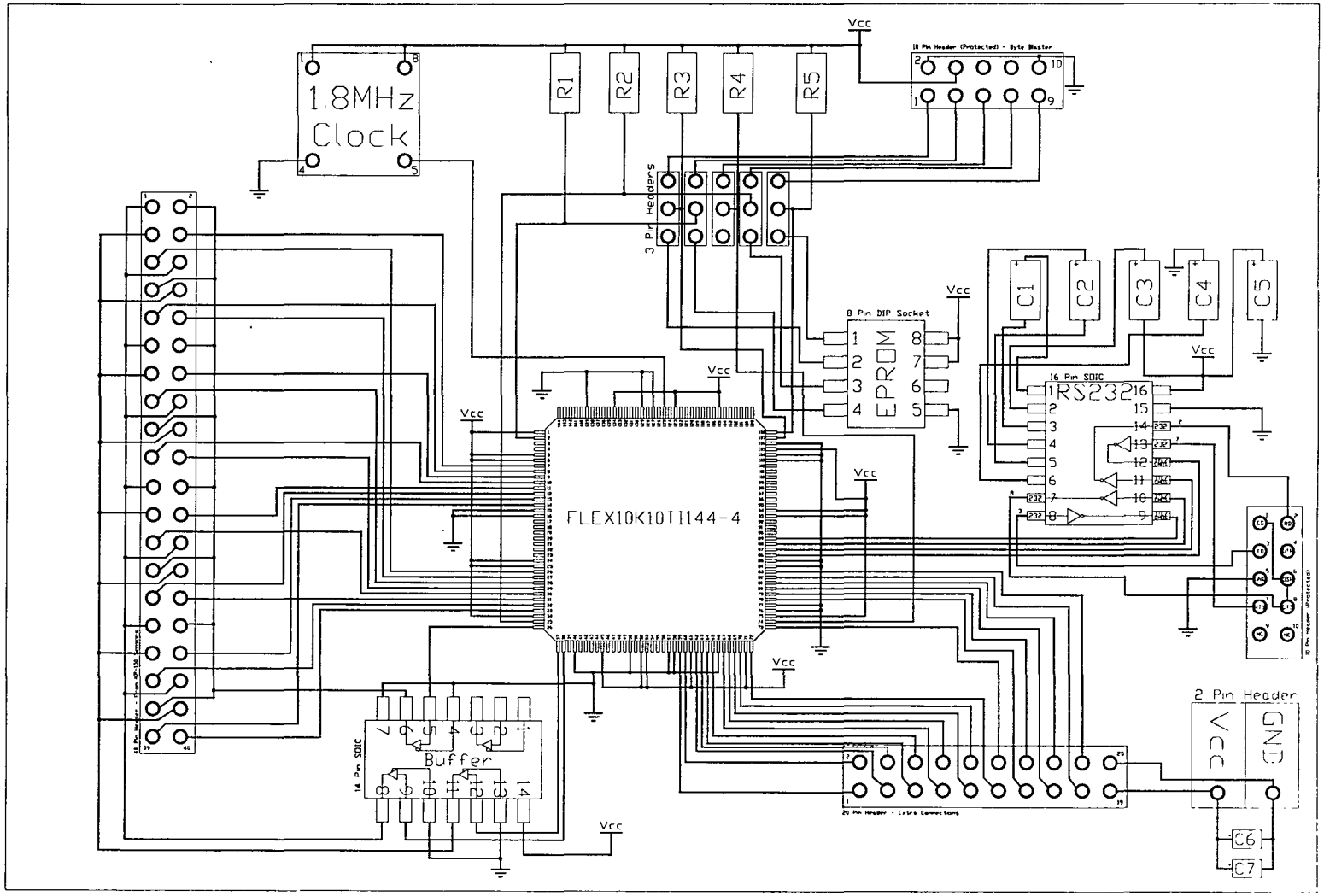


Figure 3.8 PCB Component Connectivity

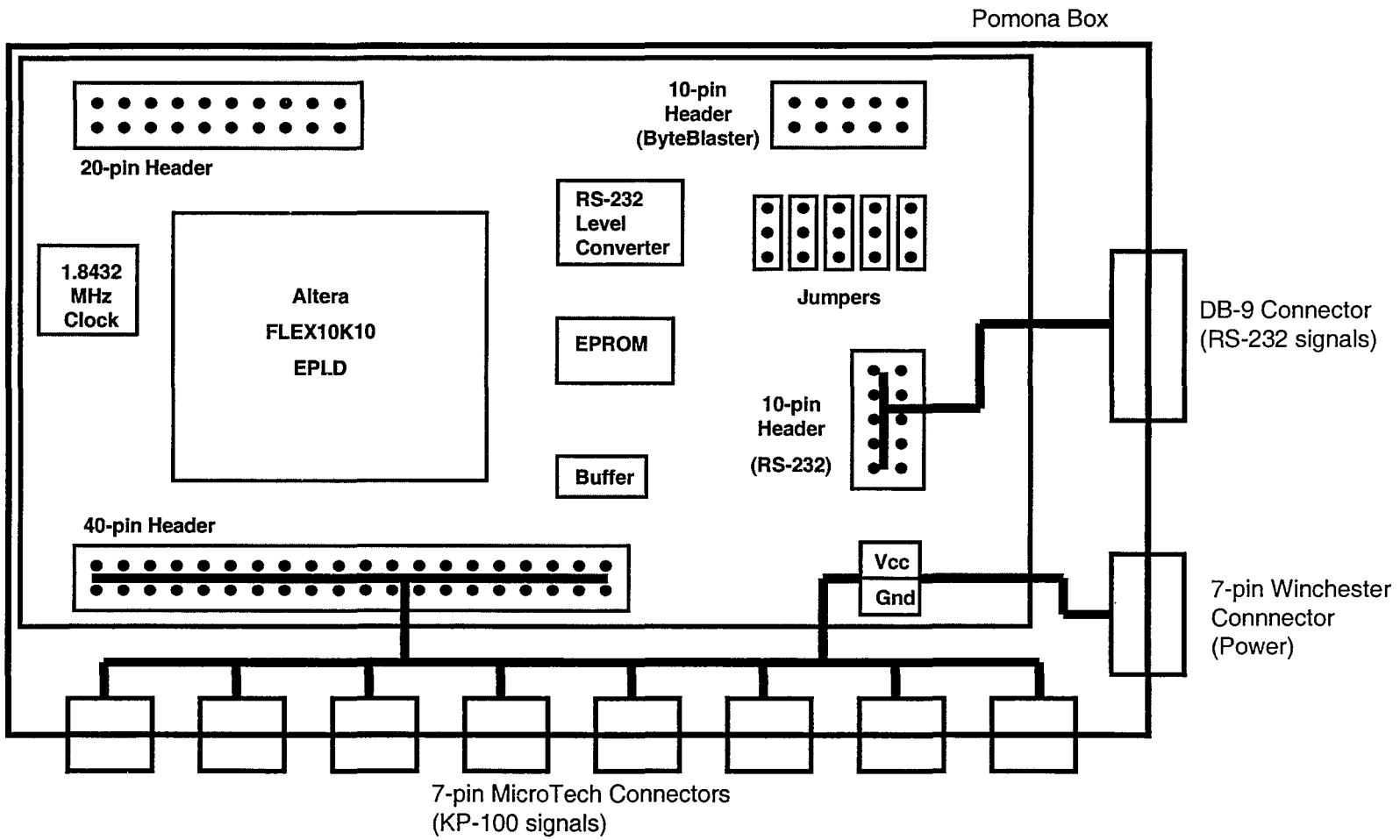


Figure 3.9 Data Acquisition Controller Module Layout

SECTION FOUR
EXPERIMENTAL VERIFICATION AND VALIDATION

4.0 Introduction

This section will discuss the steps that were taken to test and verify system performance. The first sub-section discusses the different stages of digital performance testing. Sub-section 4.2 looks at the different tests that were performed to ensure survivability. The last sub-section examines the data that was gathered from the sensors.

4.1 Testing, Digital

Testing of the hardware design was accomplished in several stages. The performance of the EPLD architecture was first simulated in software to verify correctness. Next, a prototype system was built and tested. Finally, the actual hardware that was to be used was assembled and tested.

4.1.1 Software Simulation

Initially, individual VHDL components were simulated alone. Then, several components that interacted were placed in a design and simulated. Finally, all of the components were put into one design and simulated as a complete system. The waveform editor in MAX+Plus II was used to set up inputs and examine outputs.

The behavior of the KP-100 was modeled in software to verify communication between the VHDL system design and the KP-100 sensors. This was accomplished by writing VHDL code and using parameterized megafunctions to model the inputs and outputs of the KP-100 sensor. This model was placed into the design and simulation was performed using the waveform editor in MAX+Plus II. The waveform was analyzed to ensure correct timing and data flow.

A description of the components that made up this model follows.

4.1.1.1 LPM_SHIFTREG

This is a megafunction supplied with the software. Two instances of this were created as a quick and easy way of creating shift registers to simulate the internal shift registers of the KP-100.

4.1.1.2 LPM_COUNTER

This is another megafunction. This one is a counter and was set to be 11 bits wide with an asynchronous clear input. It was clocked in the simulation by an 8.0 MHz clock. The output was connected to the NUM_CHK component.

4.1.1.3 NUM_CHK

This component took in the output from the counter. If the value of the input was between 895 and 1023 then the DTA_RDY output would be high; otherwise DTA_RDY was low. When the input was 1024, the clear line would be pulsed, clearing the counter. This created the desired 16 microsecond pulse every 128 microseconds.

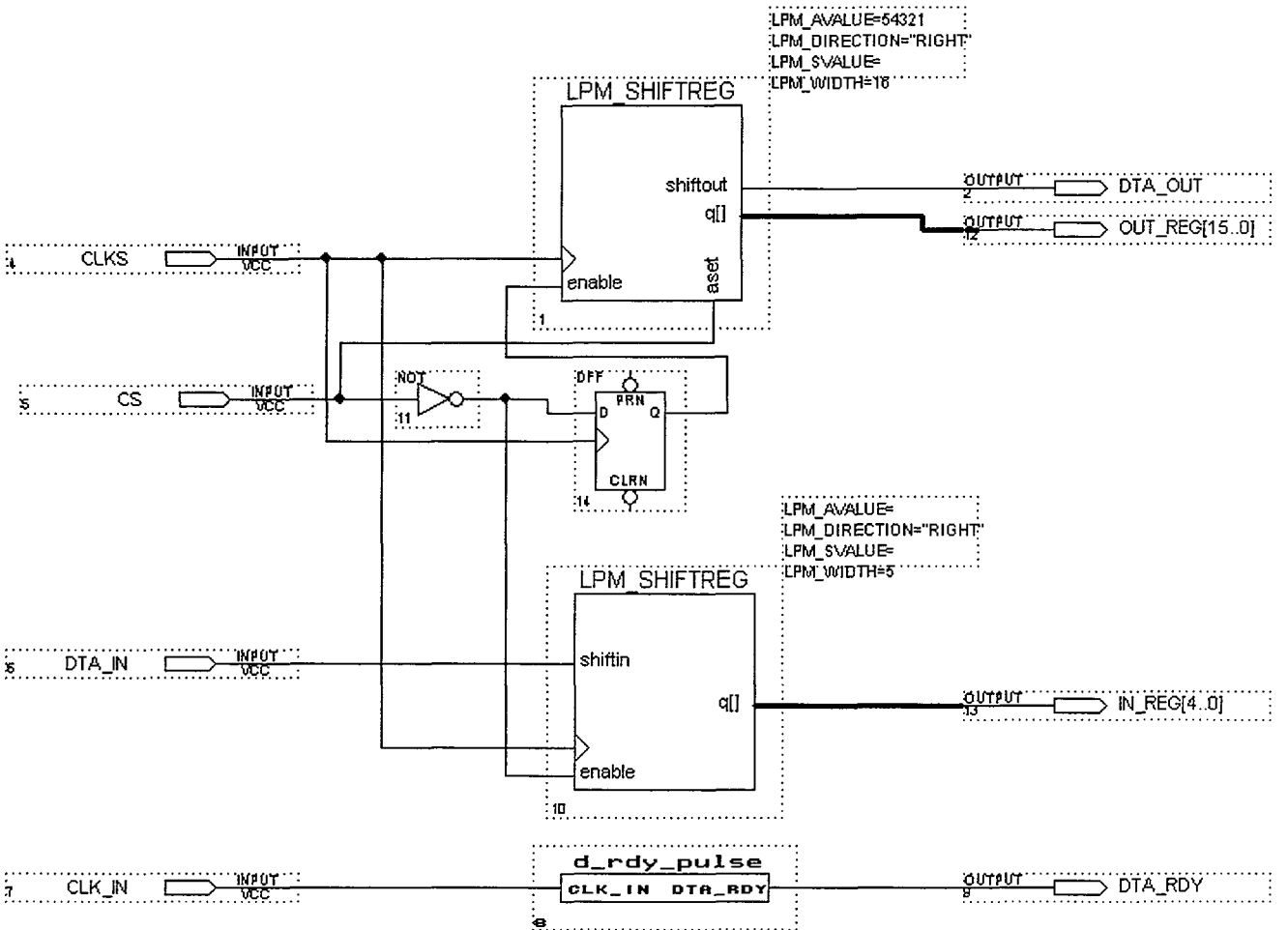


Figure 4.1 KP-100 Simulation Model

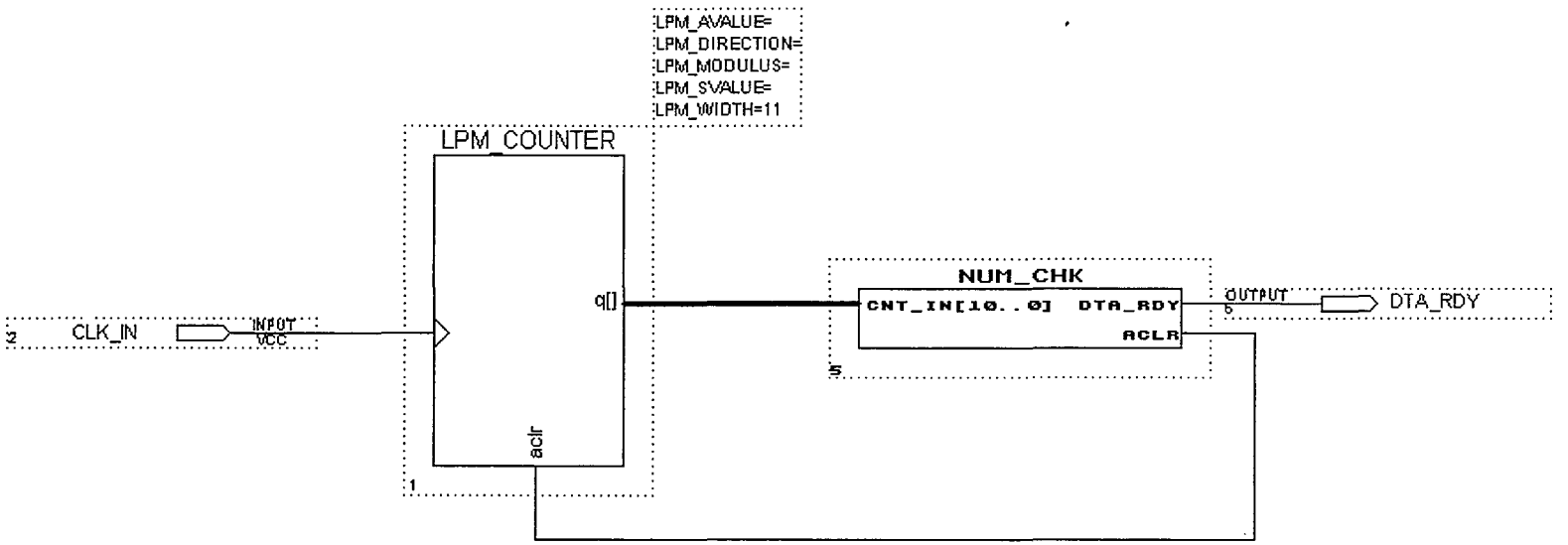


Figure 4.2 D_RDY_PULSE Component

4.1.2 Proto-type

A proto-type system needed to be wired up and tested before committing to a PCB design. The pin size and spacing on the 144-pin TQFP package of the FLEX10K device made accessing the needed I/O pins difficult. To solve this problem, a PCB was made that brought pins from the surface-mount FLEX10K device out to two standard 0.1-inch header strips. This allowed the device to be plugged directly into a standard proto-board. The rest of the components, including sensors, were plugged into the proto-board as well, to make a fully functional system. DIP packages were used for the other components so that no additional adapters were needed. Configuration of the EPLD was accomplished via the ByteBlasterMV cable. A simple communication program was written in QuickBasic to test system functionality.

4.1.3 Final Electronics

Once the proto-type system operation was verified, a PCB was made and populated. This PCB was placed in the enclosure and wired to all of the external connectors. The complete system was assembled and tested, first with the QuickBasic program, then with different LabView programs.

4.2 Testing, Survivability and Characterization

A variety of tests were performed to simulate actual wind tunnel conditions. These tests were done to both verify the ruggedness of the sensors and electronics as well as to evaluate their performance.

4.2.1 Shaker Tests

Models typically will be subjected to severe wind-induced vibration, especially at higher wind speeds and angles of attack. Any on-board sensors and electronics must be able to withstand these types of vibrations. To ensure survivability of the system, all of the digital sensors as well as the digital electronics module were put on a shaker table and were subjected to two-axis vibration forces equal to and greater than those expected in the tunnel environment. Testing was performed in accordance with a military standard for high frequency vibration tests [8]. These tests were performed without the

system being completely assembled or powered-up. After the vibration tests, the system was re-assembled in the lab and its performance evaluated.

One area of concern arose as a result of this testing. The possibility existed for the jumpers to vibrate off of their headers or for the EPROM to vibrate out of its socket. To resolve this, small pieces of non-conducting foam were affixed to the lid of the enclosure in such a way that when the lid was in place, the components in question were prevented from rising up. They were, in effect, wedged into place.

4.2.2 Thermal Tests

Heat is induced by the electronics as well as from the tunnel itself during testing. Reliable performance by the electronics must be assured for temperatures reaching up to 75 degrees Centigrade; the maximum temperature expected during normal tunnel operation. To verify high temperature performance, the sensors and electronics were set up in a temperature control chamber. This chamber was heated to various temperatures, left at those temperatures for a few hours, then allowed to cool back down to room temperature, which was around 25 C. This test routine was performed over a span of several weeks and for different temperatures. The maximum temperature of testing was 75 C.

During these tests the system was fully operational and its performance was monitored. System performance was not degraded in any way during any of the temperature tests.

4.2.3 Pressure Tests

The data sheets for the KP-100 sensors provided a general idea of expected output. However, individual sensor performance varies. This dictates a thorough characterization of each sensor through the full operating pressure range. In addition, because of the sensor temperature sensitivity, sweeps of temperature as well as pressure were performed and the output recorded. An absolute pressure controller was not available, so pressure testing was performed with applied pressures of between 1 atmosphere, roughly 14.7 psia, up to around 30 psia.

Once the data was gathered, curves were fit to the data for different temperatures. These curves enabled translation between the decimal output of the sensors and the engineering unit information, psia, which was desired. Figure 4.3 is an example of one such curve.

4.3 Data

Output data from all KP-100 sensors, including back-ups, was collected during the various phases of testing and sensor characterization. Software was written to maximize data-taking speed. When only the KP-100 system was operated, the maximum sampling rate was 28 scans of the eight pressure channels per second. When both data acquisition systems were used, a maximum rate of 9 scans per second of all channels was achieved.

System performance in the tunnel environment was the same as in the lab. Table 4.1 shows a partial data file from the test in the 16 Foot Transonic Tunnel.

4.4 Summary

This Section dealt with testing that was performed to verify that the design was valid and that the system would survive in the harsh environment of a wind tunnel. The first sub-section discussed the design simulation and the testing of a prototype and the final electronics. Sub-section 4.2 described the physical stresses that the system went through to determine survivability. Finally, the last sub-section examined the data that was received from the sensors, through the electronics.

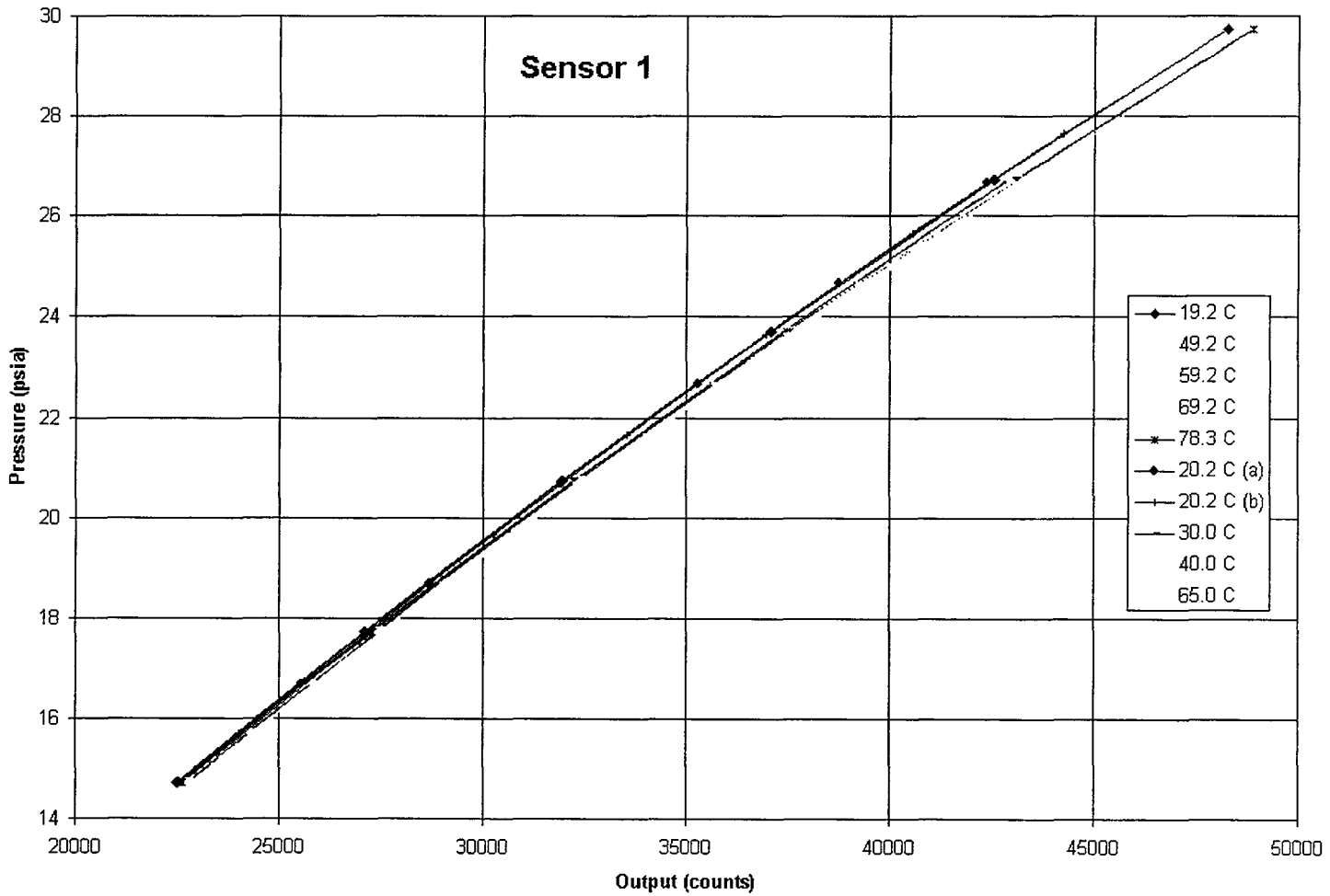


Figure 4.3 Sensor 1 Data Plot

KP100-1	KP100-2	KP100-3	KP100-4	KP100-5	KP100-6	KP100-7	KP100-9	HOUR	MIN	SEC
17193	16451	15941	16662	16739	16314	16972	18650	6	29	26
17152	16401	15893	16626	16686	16275	16960	18650	6	29	26
17156	16411	15909	16655	16727	16297	16959	18658	6	29	26
17164	16419	15924	16678	16754	16336	16953	18543	6	29	26
17185	16446	15941	16686	16741	16310	16989	18675	6	29	26
17158	16394	15911	16649	16706	16287	16974	18694	6	29	26
17214	16459	15952	16686	16752	16306	16951	18622	6	29	27
17204	16453	15982	16723	16795	16356	16983	18608	6	29	27
17217	16478	15976	16716	16791	16360	17003	18668	6	29	27
17206	16451	15960	16670	16760	16331	16966	18635	6	29	27
17210	16468	15980	16710	16772	16362	17001	18637	6	29	27
17227	16472	15980	16727	16791	16362	17008	18648	6	29	27
17217	16446	15947	16686	16752	16335	16966	18642	6	29	27
17217	16476	15978	16708	16778	16342	16985	18622	6	29	27
17200	16468	15974	16735	16793	16342	16999	18656	6	29	27
17225	16474	15995	16729	16793	16362	17005	18662	6	29	28
17221	16470	15982	16710	16766	16354	16997	18654	6	29	28
17217	16449	15945	16686	16751	16314	16959	18610	6	29	28
17212	16457	15989	16735	16793	16375	17010	18664	6	29	28
17169	16429	15934	16668	16739	16321	16964	18627	6	29	28
17199	16438	15928	16666	16712	16277	16964	18679	6	29	28
17181	16432	15930	16670	16739	16301	16981	18696	6	29	28
17214	16463	15962	16680	16741	16301	16964	18637	6	29	28
17217	16453	15960	16716	16758	16327	16959	18610	6	29	29
17191	16446	15958	16727	16785	16356	16989	18618	6	29	29
17179	16438	15932	16666	16739	16333	16989	18660	6	29	29
17212	16449	15945	16686	16739	16335	16970	18637	6	29	29
17236	16482	15997	16725	16772	16358	16991	18664	6	29	29
17193	16455	15968	16708	16766	16323	16947	18608	6	29	29
17204	16436	15922	16656	16721	16312	16972	18658	6	29	29
17173	16434	15937	16666	16723	16306	16987	18681	6	29	29
17179	16403	15905	16632	16704	16301	16945	18639	6	29	29
17223	16472	15976	16708	16770	16362	16993	18631	6	29	30
17193	16451	15952	16701	16760	16335	16995	18683	6	29	30
17197	16442	15947	16682	16745	16297	16962	18660	6	29	30
17185	16455	15956	16670	16737	16321	16964	18648	6	29	30
17195	16440	15964	16686	16747	16338	16981	18640	6	29	30
17202	16451	15976	16708	16772	16336	16964	18574	6	29	30
17206	16461	15970	16721	16795	16362	16995	18637	6	29	30

Table 4.1 Partial Data File From Tunnel

SECTION FIVE
CONCLUSION

5.0 Completed Work

A fully functional system for remotely reading data from digital pressure sensors was assembled. The KP-100 sensors, the EPLD-based electronics module, the RS-232-to-RS485 converter and the RF and OF units were all mounted inside a Boeing T-45 model during a test in the 16-foot Transonic Tunnel at NASA Langley Research Center. This digital system, along with several other data acquisition systems, was in operation throughout testing. Sensor data was recorded via the hardwire and RF communication links at various test points during different tunnel conditions.

5.1 Assessment

The system that was assembled met all of the requirements. The sensors and all of the system components withstood the harsh conditions of a wind tunnel test: high temperatures and severe vibration induced by wind speeds of up to 0.8 times the speed of sound. The hardwire and RF communications links never failed during the tunnel test. One of the optical fibers was crushed during model installation so OF communication was not possible in the tunnel. Also, the final configuration did not make use of the diagnosis modes of the KP-100 sensors. This functionality was not necessary at this early stage of development, so to simplify operation, it was not implemented.

The data that was taken during the tunnel test seemed reasonable. There were no shared pressure ports, so no direct comparisons with a known standard were possible.

5.2 Deficiencies and Future Work

Although the system met all of the specifications and performed as expected in an actual wind tunnel application, improvements could be made to enhance system performance.

5.2.1 Speed

The output data rate of 9600 bps is much too slow for many applications. The determining factor for this data rate was the RS-232-to-RS-485 converter that had to be addressed at 9600 baud. Since each of the 8 KP-100 sensors has an update rate of 7.8 kHz with 16 data bits and 2 start and 2 stop bits per sample, the optimal rate for data transfer would be roughly 1.2 Mbps.

5.2.2 Size

The current system PCB is not optimized for size. A relatively large area is taken up by components that are used solely for prototyping purpose: ByteBlasterMV header and associated pull-up resistors and jumper headers. In a final design, these components could be completely eliminated reducing the board size by 25%. Dual row, 0.1" spacing headers account for a good portion of PCB area. These headers include the 40-pin KP-100 signal header, the 10-pin RS-232 header and an unused 20-pin header for additional connection to the EPLD. This type of header was selected because of its availability and standard use. Much smaller connectors to the PCB could be used, resulting in a substantial reduction in needed PCB size.

5.2.3 Number of Channels

Due to the limited availability of KP-100 sensors, the system took in only 8 channels of pressure data. The FLEX10K EPLD has 102 user I/O pins available with only 35 being used with this design. Another 16 or 24 sensors could easily be added with only minimal design changes.

5.2.4 Output Protocol

The digital electronics module outputs an RS-232 signal, which is what was originally called for. However, the change to RS-485 necessitated the use of an RS-232-to-RS-485 converter. This not only increased the overall size of the system but was the limiting factor in the ultimate data transfer rate. If no other devices were needed on the RS-485 bus, meaning that addressability was not an issue, the RS-232 level converter on the PCB could be replaced with an RS-485 level converter. The electronics module would than have an RS-485 output and the RS-232-to-RS-485 converter would no longer be necessary. This would

eliminate the data transfer bottleneck as well as increase the effective transmission distance.

5.3 Summary

This thesis has described the design process, testing and implementation of a data acquisition system that can be used with digital pressure sensors in a wind tunnel model. The proposed system is 1) intelligent, in that it can respond to commands, 2) small enough to be embedded in a scale wind tunnel model, 3) able to communicate with an external host computer and 4) easily and cheaply re-configurable for different baud rates and number of channels. The system has performed according to all design specifications but is really just the first step towards a smaller and faster final product.

REFERENCES

[1] Data Sheet KP100 : Surface Mount Capacitive Silicon Absolute Pressure Sensor Siemens Semiconductor Group, July 1997

[2] User's Manual ADAM-4550 Radio Modem Module Advantech

[3] Reference Manual Model 272A Optoverter - 2 wire RS-422 to Fiber Optic Line Driver/Converter Telebyte

[4] Strangio, Christopher E. The RS232 Standard: A Tutorial with the Signal Names and Definitions CAMI Research Inc. Lexington, Massachusetts, 1993

[5] B & B Electronics Technical Article #1: Basics of the RS-485 Standard B & B Electronics Manufacturing Company, Ottawa, IL 1994

[6] Data Sheet FLEX 10K : Embedded Programmable Logic Family Altera Corporation, San Jose,CA June 1999

[7] Smith, Douglas J. HDL Chip Design: Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog Doone Publications Madison, AL 1996

[8] MIL-STD-202F : METHOD 204D - VIBRATION, HIGH FREQUENCY 1 APRIL 1980

APPENDIX

APPENDIX A - LIST OF ABBREVIATIONS

bps	bits per second
CD	Carrier Detect
COTS	Commercial Off The Shelf
CTS	Clear To Send
DAS	Data Acquisition System
DCE	Data Communications Equipment
DIP	Dual In-line Pin
DSR	Data Set Ready
DTE	Data Terminal Equipment
DTR	Data Terminal Ready
EMI	Electro-Magnetic Interference
EPLD	Embedded Programmable Logic Device
EPROM	Electrically Programmable Read Only Memory
ESP	Electronically Scanned Pressure
FLEX	Flexible Logic Element MatriX
FPGA	Field Programmable Gate Array
GND	Ground
IC	Integrated Circuit
kHz	Kilohertz
kPa	KiloPascal
LSB	Least Significant Byte
Mbps	Mega bits per second
MHz	Megahertz
MSB	Most Significant Byte
NC	Not Connected
OF	Optical Fiber
PCB	Printed Circuit Board

PLD	Programmable Logic Device
PSIA	Pounds Per Square Inch – Absolute
RD	Received Data
RF	Radio Frequency
RTS	Request To Send
SPI	Serial Peripheral Interface
TD	Transmitted Data
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

APPENDIX B - VHDL FILES

KP_100_CNTRLR

```
PACKAGE define2 IS
  TYPE STATE4 is (s0,s1,s2,s3);
END define2;
--------------------------------------------------------------------------

LIBRARY IEEE, work;
USE IEEE.STD_LOGIC_1164.ALL, work.define2.all;

ENTITY KP_100_cntrlr IS
        PORT(
                    clock, reset, dta_rdy, get_data, cnt_15        : IN
STD_LOGIC;
                    CS, CLR_CNT, ENABLE_CNT, ENABLE_SHIFTREG, DONE
: OUT   STD_LOGIC
                  );
END KP_100_cntrlr ;

ARCHITECTURE KPC1 OF KP_100_cntrlr  IS

        SIGNAL present_state, next_state : STATE4;
 BEGIN

   PROCESS (present_state, dta_rdy, get_data, cnt_15)
        BEGIN

           CASE present_state IS

              WHEN s0 =>

                    CS <= '1';
                    CLR_CNT <= '1';
                    ENABLE_CNT <= '0';
                    ENABLE_SHIFTREG <= '0';
                    DONE <= '0';

                   IF ( dta_rdy = '1' and get_data = '1') THEN
                     next_state <= s1;
                   ELSE
                     next_state <= s0;
                   END IF;
                       -----------

              WHEN s1 =>

                    CS <= '0';
                    CLR_CNT <= '0';
                    ENABLE_CNT <= '1';
                    ENABLE_SHIFTREG <= '1';
                    DONE <= '0';

                   IF ( cnt_15 = '1') THEN
                     next_state <= s2;
                   ELSE
```

```
                next_state <= s1;
              END IF;
                  -----------

          WHEN s2 =>

                  CS <= '1';
                  CLR_CNT <= '0';
                  ENABLE_CNT <= '0';
                  ENABLE_SHIFTREG <= '0';
                  DONE <= '1';

                  next_state <= s3;
                     -----------

         WHEN s3 =>

                  CS <= '1';
                  CLR_CNT <= '0';
                  ENABLE_CNT <= '0';
                  ENABLE_SHIFTREG <= '0';
                  DONE <= '1';

                 IF ( get_data = '0') THEN
                   next_state <= s0;
                 ELSE
                   next_state <= s3;
                 END IF;
                     -----------


      END CASE;

END PROCESS ;

PROCESS (reset, clock)
       BEGIN
              IF (reset = '1') THEN
                 present_state <= s0;
              ELSIF rising_edge(clock) THEN
                 present_state <=next_state;
              END IF;
END PROCESS ;


END KPC1;
```

RS_232_CNTRL_1_SCAN

```
------------------------------------------------------------------------
--
--  Sends out 8 sensor readings: 2 bytes each
--
--
------------------------------------------------------------------------

PACKAGE define1 IS
  TYPE STATE8 is (s0,s1,s2,s3,s4,s5,s6,s7);
END define1;
------------------------------------------------------------------------

LIBRARY IEEE, work;
USE IEEE.STD_LOGIC_1164.ALL, work.define1.all;

ENTITY RS232_cntrlr_1_scan IS
        PORT(
      clock, reset, done, two_bytes_sent, dtr, rts  : IN STD_LOGIC;
      sensor_sel                        : IN   INTEGER RANGE 0 TO  7;

      LOAD,CTS,ENABLE,GET_DATA,SEND_DATA,INC_SENSOR : OUT STD_LOGIC
                );
END RS232_cntrlr_1_scan ;

ARCHITECTURE RSC1 OF RS232_cntrlr_1_scan  IS

        SIGNAL present_state, next_state : STATE8;
 BEGIN

   PROCESS (present_state, dtr, rts, two_bytes_sent,done,sensor_sel)
        BEGIN

           CASE present_state IS

              WHEN s0 =>

                    LOAD <= '0';
                    CTS <= '1';
                    ENABLE <= '0';
                    GET_DATA <= '0';
                    SEND_DATA <= '0';
                    INC_SENSOR <= '0';

                   IF ( dtr = '0') THEN
                     next_state <= s1;
                   ELSE
                     next_state <= s0;
                   END IF;
                       -----------

              WHEN s1 =>

                    LOAD <= '0';
                    CTS <= '1';
```

```
          ENABLE <= '0';
          GET_DATA <= '1';
          SEND_DATA <= '0';
          INC_SENSOR <= '0';

         IF ( done = '1') THEN
           next_state <= s2;
         ELSE
           next_state <= s1;
         END IF;

             -----------

  WHEN s2 =>

          LOAD <= '1';
          CTS <= '1';
          ENABLE <= '0';
          GET_DATA <= '0';
          SEND_DATA <= '0';
          INC_SENSOR <= '1';

         IF ( rts = '0') THEN
           next_state <= s3;
         ELSE
           next_state <= s2;
         END IF;
             -----------

 WHEN s3 =>

          LOAD <= '0';
          CTS <= '0';
          ENABLE <= '0';
          GET_DATA <= '0';
          SEND_DATA <= '0';
          INC_SENSOR <= '0';

          next_state <= s4;

             -----------

WHEN s4 =>

          LOAD <= '0';
          CTS <= '0';
          ENABLE <= '1';
          GET_DATA <= '0';
          SEND_DATA <= '0';
          INC_SENSOR <= '0';

          next_state <= s5;

             ----------

   WHEN s5 =>
```

```
                    LOAD <= '0';
                    CTS <= '0';
                    ENABLE <= '1';
                    GET_DATA <= '0';
                    SEND_DATA <= '1';
                    INC_SENSOR <= '0';

                   IF (two_bytes_sent = '1') THEN
                     next_state <= s6;
                   ELSE
                     next_state <= s5;
                   END IF;

                      -----------

            WHEN s6 =>

                    LOAD <= '0';
                    CTS <= '1';
                    ENABLE <= '0';
                    GET_DATA <= '0';
                    SEND_DATA <= '0';
                    INC_SENSOR <= '0';

                    next_state <= s7;

                      -----------

           WHEN s7 =>

                    LOAD <= '0';
                    CTS <= '1';
                    ENABLE <= '0';
                    GET_DATA <= '0';
                    SEND_DATA <= '0';
                    INC_SENSOR <= '0';

                   IF (sensor_sel = 0) THEN
                     next_state <= s7;
                   ELSE
                     next_state <= s0;
                   END IF;

      END CASE;

END PROCESS ;

PROCESS (reset, clock)
       BEGIN
              IF (reset = '1') THEN
                 present_state <= s0;
              ELSIF rising_edge(clock) THEN
                 present_state <=next_state;
              END IF;
END PROCESS ;

END RSC1;
```

TIMER2

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL, ieee.std_logic_arith.ALL;

ENTITY  timer2 IS

        PORT
        (
                clock                       : IN    STD_LOGIC;
                dta_rdy                     : IN    STD_LOGIC;
                enable_cnt                  : IN    STD_LOGIC;
                DRDY_or_TOUT                : OUT   STD_LOGIC
        );
End timer2;

ARCHITECTURE T1 OF timer2 IS
Signal          internal_count          : INTEGER RANGE 0 TO 127;
Signal          time_out                    : STD_LOGIC;
BEGIN

PROCESS (clock, enable_cnt)

     Variable        my_count           : INTEGER RANGE 0 TO  127;
     Variable        t                              : STD_LOGIC;
     BEGIN
        IF enable_cnt = '0' THEN
            my_count  := 0;
            t := '0';
        ELSIF rising_edge(clock) THEN
              IF (my_count = 63) THEN t := not t;
              ELSE t := t;
              END IF;

              my_count  := my_count+1;
           ELSE my_count := my_count;
        END IF;
        time_out <= t;

 END PROCESS;

DRDY_or_TOUT  <= (time_out OR dta_rdy);

END T1;
```

SHIFT_REG_16

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

ENTITY shift_reg_16 IS
   PORT (
               clock, enable, shift_in       : IN STD_LOGIC;
              CONTENTS                         : OUT STD_LOGIC_VECTOR
(15 DOWNTO 0)
         );
END shift_reg_16;

ARCHITECTURE SR1 OF shift_reg_16 IS

BEGIN

PROCESS(clock, enable)
VARIABLE temp_data  : STD_LOGIC_VECTOR (15 DOWNTO 0) :=
"0000000000000000";

BEGIN
    IF Rising_Edge(clock) THEN
      IF(enable = '1') THEN
         FOR i IN 0 TO 14 LOOP
        temp_data(i) := temp_data(i+1);
         END LOOP;
         temp_data(15) := shift_in;
      END IF;
    END IF;
    CONTENTS <= temp_data;

END PROCESS;

END SR1;
```

CLK_DIV_BY_8

```
LIBRARY IEEE, work;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY clk_div_by_8 IS
        PORT(
                clock, reset            : IN    STD_LOGIC;
                CLK_DIV_8           : OUT   STD_LOGIC
                  );
END clk_div_by_8 ;

ARCHITECTURE CD1 OF clk_div_by_8  IS

        signal Div2, Div4, Div8   : STD_LOGIC;

 BEGIN

   PROCESS (clock, reset, Div2, Div4 )
        BEGIN
           IF reset = '1' THEN
              Div2 <= '0';
           ELSIF Rising_Edge(clock) THEN
              Div2 <= not Div2;
           END IF;

           IF reset = '1' THEN
              Div4 <= '0';
           ELSIF Rising_Edge(Div2) THEN
              Div4 <= not Div4;
           END IF;

           IF reset = '1' THEN
              Div8 <= '0';
           ELSIF Rising_Edge(Div4) THEN
              Div8 <= not Div8;
           END IF;


           -- Resync. with clock
           IF reset = '1' THEN
              CLK_DIV_8 <= '0';
           ELSIF Rising_Edge(clock) THEN
              CLK_DIV_8 <= Div8;
           END IF;

   END PROCESS ;


END CD1;
```

COUNTER_11_13

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL, ieee.std_logic_arith.ALL;


ENTITY  counter_11_13 IS

        PORT
        (
                clock                       : IN    STD_LOGIC;
                clr_cnt                     : IN    STD_LOGIC;
                enable_cnt                  : IN    STD_LOGIC;
                COUNT_VALUE   : OUT   INTEGER RANGE 0 TO 15;
                CNT_15                      : OUT   STD_LOGIC
        );
End counter_11_13;


ARCHITECTURE C1 OF counter_11_13 IS

Signal          internal_count          : INTEGER RANGE 0 TO 15;

BEGIN

PROCESS (clock, enable_cnt, clr_cnt)

     Variable        my_count           : INTEGER RANGE 0 TO  15;

     BEGIN
        IF clr_cnt = '1' THEN
            my_count  := 0;
        ELSIF rising_edge(clock) THEN
           IF (enable_cnt = '1')  then
              my_count  := my_count+1;
           ELSE
              my_count := my_count;
           END IF;
        END IF;

        internal_count <= my_count;
        COUNT_VALUE <= my_count;

 END PROCESS;
```

```
-----------------------------------------------
--15 Detector
-----------------------------------------------

Process (internal_count)

    Variable    hi_cnt    : STD_LOGIC;
Begin
                IF internal_count = 15 THEN
                hi_cnt := '1';
                ELSE
                hi_cnt  := '0';
                END IF;

                CNT_15 <= hi_cnt;
End Process;
END C1;
```

MOD95CNTR

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mod95cntr IS
        PORT(
                  clk_1_8M, reset    : IN     STD_LOGIC;
                  CLK_9600          : OUT    STD_LOGIC
                 );
END  mod95cntr ;

ARCHITECTURE M1 OF  mod95cntr  IS
          SIGNAL ninety_five      : STD_LOGIC;

 BEGIN

   PROCESS(reset, clk_1_8M, ninety_five)
          VARIABLE tmp_cnt   : INTEGER RANGE 0 to 127 := 0;
          VARIABLE temp_95      :  STD_LOGIC;

          BEGIN
                  IF reset = '1' THEN
                     tmp_cnt := 0;
                  ELSIF Rising_Edge(clk_1_8M) THEN
                        IF ninety_five ='1' then
                           tmp_cnt := 0;
                        ELSE
                           tmp_cnt := tmp_cnt + 1;
                        END IF;
                         -----------------
                        IF tmp_cnt = 95 THEN
                           temp_95 := '1';
                        ELSE
                           temp_95 := '0';
                        END IF;

                        -------------------
                  END IF;
                  ninety_five <= temp_95;
END PROCESS;

   PROCESS (reset, ninety_five)
        VARIABLE temp_out    : STD_LOGIC;

        BEGIN
             IF (reset = '1') THEN
                   temp_out := '0';
             ELSIF Rising_Edge(ninety_five) THEN
                   temp_out := not temp_out;
                   ELSE
                   temp_out := temp_out;
             END IF;
            CLK_9600 <= temp_out;
        END PROCESS ;
END M1;
```

SENSOR_CNTR

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL, ieee.std_logic_arith.ALL;

ENTITY  sensor_cntr IS

        PORT
        (
                inc_sensor                  : IN  STD_LOGIC;
                clr_cnt                     : IN  STD_LOGIC;
                SENSOR_SEL                  : OUT INTEGER RANGE 0 TO 7

        );
End sensor_cntr;

ARCHITECTURE SC1 OF sensor_cntr IS
BEGIN

PROCESS (inc_sensor, clr_cnt)

     Variable        my_count           : INTEGER RANGE 0 TO  7;

     BEGIN
        IF clr_cnt = '1' THEN
            my_count  := 0;
        ELSIF rising_edge(inc_sensor) THEN
            my_count  := my_count+1;
           ELSE
            my_count := my_count;
        END IF;

        SENSOR_SEL <= my_count;

 END PROCESS;


END SC1;
```

MUX8_1

```
LIBRARY IEEE, work;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux8_1 IS
        PORT(
             in0, in1, in2, in3, in4, in5, in6, in7  : IN STD_LOGIC;
             sel                : IN    STD_LOGIC_VECTOR(2 downto 0);
             MUX_OUT            : OUT   STD_LOGIC
                 );
END mux8_1  ;

ARCHITECTURE M1 OF mux8_1   IS

BEGIN

   PROCESS (sel)
           .
      VARIABLE temp_out      : STD_LOGIC := '0';
        BEGIN

          CASE sel IS

              WHEN "000" =>  temp_out := in0;

              WHEN "001" =>  temp_out := in1;

              WHEN "010" =>  temp_out := in2;

              WHEN "011" =>  temp_out := in3;

              WHEN "100" =>  temp_out := in4;

              WHEN "101" =>  temp_out := in5;

              WHEN "110" =>  temp_out := in6;

              WHEN "111" =>  temp_out := in7;


              WHEN OTHERS => temp_out := '0';

         END CASE;

      MUX_OUT <= temp_out;

END PROCESS ;

END M1;
```

MUX4_1

```
LIBRARY IEEE, work;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux4_1 IS
        PORT(
                    in0, in1, in2, in3      : IN  STD_LOGIC;
                    sel : IN STD_LOGIC_VECTOR(1 downto 0);
                    MUX_OUT                 : OUT STD_LOGIC
                  );
END mux4_1   ;

ARCHITECTURE M1 OF mux4_1    IS

BEGIN

   PROCESS (sel)

      VARIABLE temp_out      : STD_LOGIC := '0';
        BEGIN

          CASE sel IS

              WHEN "00" =>  temp_out := in0;

              WHEN "01" =>  temp_out := in1;

              WHEN "10" =>  temp_out := in2;

              WHEN "11" =>  temp_out := in3;

              WHEN OTHERS => temp_out := '0';

         END CASE;

      MUX_OUT <= temp_out;

END PROCESS ;

END M1;
```

RS232_SR_OUT

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY RS232_sr_out IS
        PORT(
                load, clock, send_data, enable     : IN  STD_LOGIC;
                out_data_reg    : IN STD_LOGIC_VECTOR(15 downto 0);
                OUT_DATA, TWO_BYTES_SENT          : OUT   STD_LOGIC
                 );
END  RS232_sr_out ;

ARCHITECTURE R1 OF  RS232_sr_out  IS

        SIGNAL contents      : STD_LOGIC_VECTOR(19 downto 0);
        SIGNAL shift_out      : STD_LOGIC := '0';

BEGIN        .

   PROCESS (load, clock, enable)
           VARIABLE temp      : STD_LOGIC_VECTOR(19 downto 0);

        BEGIN
           IF Rising_Edge(clock)  THEN

              IF load = '1' THEN
                 temp := '1' & out_data_reg(15 downto 8) & "01" &
         out_data_reg(7 downto 0) & '0';
              ELSIF enable = '1' THEN
                    shift_out <= temp(0);
                    FOR i IN 0 to 18 LOOP
                    temp(i) := temp(i+1);
                    END LOOP;
                    temp(19) := '0';
                 ELSE
                    temp := temp;
              END IF;

            ELSE
                   temp := temp;
             END IF;

           contents <= temp;

   END PROCESS ;

--------------------------------
-- MUX

   PROCESS (shift_out, send_data)
        VARIABLE x    : STD_LOGIC;

        BEGIN
             IF (send_data = '1') THEN
                   x := shift_out;
```

```
            ELSE
                  x := '1';
            END IF;

            OUT_DATA <= x;

  END PROCESS ;

-------------------------------------------------------------------------
-- ALL ZERO CHECKER - When register contains all zeros, 20 bits have
been sent

   PROCESS (contents)
        VARIABLE y    : STD_LOGIC;

        BEGIN
            IF (contents = "00000000000000000000") THEN
                  y := '1';
            ELSE
                  y := '0';
            END IF;

            TWO_BYTES_SENT <= y;

  END PROCESS ;


END R1;
```

COUNTER_3BIT

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL, ieee.std_logic_arith.ALL;

ENTITY  counter_3bit IS

        PORT
        (
                clock                       : IN    STD_LOGIC;
                clr_cnt                     : IN    STD_LOGIC;
                enable_cnt                  : IN    STD_LOGIC;
                SEVEN                       : OUT   STD_LOGIC
        );
End counter_3bit;

ARCHITECTURE C1 OF counter_3bit IS
Signal          internal_count          : INTEGER RANGE 0 TO 7;

BEGIN

PROCESS (clock, enable_cnt, clr_cnt)

     Variable        my_count           : INTEGER RANGE 0 TO  7;

     BEGIN
        IF clr_cnt = '1' THEN
            my_count  := 0;
        ELSIF rising_edge(clock) THEN
           IF (enable_cnt = '1')  then
              my_count  := my_count+1;
           ELSE
              my_count := my_count;
           END IF;
        END IF;

        internal_count <= my_count;

 END PROCESS;

-----------------------------------------------
-- 15 Detector

Process (internal_count)

    Variable    hi_cnt    : STD_LOGIC;
Begin
                IF internal_count = 7 THEN
                hi_cnt := '1';
                ELSE
                hi_cnt  := '0';
                END IF;

                SEVEN <= hi_cnt;

End Process;
END C1;
```

RS_232_INPUT_CNTRLR

```
PACKAGE define3 IS
  TYPE STATE2 is (s0,s1);
END define3;
-------------------------------------------------------------------

LIBRARY IEEE, work;
USE IEEE.STD_LOGIC_1164.ALL, work.define3.all;

ENTITY RS232_input_cntrlr IS
        PORT(
                  clock, reset, td, seven      : IN    STD_LOGIC;
                  CLR_CNT, ENABLE              : OUT   STD_LOGIC
                );
END RS232_input_cntrlr ;

ARCHITECTURE RSIC1 OF RS232_input_cntrlr  IS

        SIGNAL present_state, next_state : STATE2;
 BEGIN

   PROCESS (present_state, td, seven)
        BEGIN

           CASE present_state IS

              WHEN s0 =>

                    CLR_CNT <= '1';
                    ENABLE <= '0';

                   IF ( td = '0') THEN
                     next_state <= s1;
                   ELSE
                     next_state <= s0;
                   END IF;
                       -----------

              WHEN s1 =>

                    CLR_CNT <= '0';
                    ENABLE <= '1';


                   IF ( seven = '1') THEN
                     next_state <= s0;
                   ELSE
                     next_state <= s1;
                   END IF;

                       -----------


       END CASE;
```

```
END PROCESS ;

PROCESS (reset, clock)
        BEGIN
                IF (reset = '1') THEN
                   present_state <= s0;
                ELSIF rising_edge(clock) THEN
                   present_state <=next_state;
                END IF;
END PROCESS ;


END RSIC1;
```

SHIFT_REG_INPUT

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

ENTITY shift_reg_input IS
   PORT (
               clock, enable, shift_in, reset      : IN STD_LOGIC;
               CONTENTS        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
         );
END shift_reg_input ;

ARCHITECTURE SRI1 OF shift_reg_input  IS

BEGIN

PROCESS(clock, enable)
VARIABLE temp_data  : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";

BEGIN
    IF reset = '1' THEN
       temp_data := "00000000";

    ELSIF Rising_Edge(clock) THEN
      IF(enable = '1') THEN
         FOR i IN 0 TO 6 LOOP
        temp_data(i) := temp_data(i+1);
         END LOOP;
         temp_data(7) := shift_in;
      END IF;
    END IF;
    CONTENTS <= temp_data;

END PROCESS;

END SRI1;
```

NUM_CHK

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY num_chk IS
        PORT(
                  cnt_in             : IN    INTEGER RANGE 0 to 2047;
                  DTA_RDY, ACLR      : OUT   STD_LOGIC
                 );
END  num_chk ;

ARCHITECTURE N1 OF  num_chk  IS

BEGIN

   PROCESS(cnt_in)

   VARIABLE tmp_out, tmp_clr : STD_LOGIC;

    BEGIN

     IF (cnt_in > 895 and cnt_in < 1023) THEN
        tmp_out := '1';
     ELSE
        tmp_out := '0';
     END IF;

     IF (cnt_in = 1024) THEN
        tmp_clr := '1';
     ELSE
        tmp_clr := '0';
     END IF;

     DTA_RDY <= tmp_out;
     ACLR <= tmp_clr;

  END  PROCESS;

END N1;
```

VITA

John J. Novakoski
41 Barnard Hill Road
Dunbarton, NH 03045

B.S. Computer Engineering, Old Dominon University, Norfolk,VA 1998
M.S. Computer Engineering, Old Dominon University, Norfolk,VA 2000

Currently working as a Digital Design Engineer at Sanders, a Lockheed Martin company in Nashua, New Hampshire. Worked part-time at NASA Langley Research Center in Hampton, Virginia from June 1998 through December 1999.