Overview and Introduction

This project will bring together several pieces of software and draw on knowledge gained in several of the previous projects in this class. The overall objective is to demonstrate a closed loop control system. A closed-loop control system consists of an input variable that specifies a set point, an output variable that is controlled, using a feedback system, to maintain the output value as a function of the input set point. In this particular project, the objective is to implement a closed loop control system that will detect and track a light source as it is moved around. Accomplishing this overall objective will require completing a number of subtasks, and then bringing each of these pieces together to complete the system.

The system will be made up of the following primary hardware components:
1) A servo motor which can rotate to and hold a particular position.
2) A servo motor controller that can be commanded via software running on a PC to control the servo position. The servo motor controller is implemented in the gate array on a Digilab XL board.
3) An optical sensor head that allows determining the brightness in five different directions. This sensor head will be attached to the servo motor so that software can control the direction that the optical sensors look.
4) An analog to digital conversion module that can convert the brightness levels detected by the optical sensor head into digital values that can be read by software on the PC. This module is implemented using a microcontroller on a board that plugs into a Digilab XL board.

The system will also be made up of the following primary software components:
1) Low level drivers to send servo position commands to the servo controller.
2) Low level drivers to communicate with the analog to digital conversion module and obtain light intensity information.
3) Timing software (via the system timer interrupt) to control the timing of operations in the various software modules.
4) The closed loop control software, which uses the above components to implement the overall control task.

The light intensity information is obtained from the optical sensor array attached to the moving part of the servo motor. The optical sensor array contains five photo-transistors arranged in a pattern which detects light intensity in five different directions simultaneously. The position of the sensor array is controlled by controlling the position of the servo motor. The brightness information from the optical sensor array is conveyed as five analog voltage levels that are converted by an analog to digital conversion module into
digital values. The analog to digital conversion module communicates via an RS-232 serial port to the control software running on a PC. The control software communicates with the analog to digital conversion module via a command/response protocol transmitted over the serial port.

The servo motor position is controlled by a pulse width modulator circuit configured in the gate array of the Digilab XL board. This servo motor controller is capable of simultaneously controlling four servo motors. However, only one is used for this project. The servo motor position is controlled by writing position values to a control register in the servo controller. The Digilab XL board is connected to the PC via the parallel port, and the control software writes position values to the servo controller via the parallel port.

The control software will be responsible for sweeping the optical sensor head back and forth until a bright light source is detected. It will then center the sensor array on the bright light source, tracking it as it moves.

**Supporting Documentation**

In addition to this document, the following documents contain information needed to understand the various pieces of the system and to implement the solution:

- A/D Converter Firmware User’s Guide
- R/C Hobby Servo Controller User’s Guide
- Interfacing the Standard Parallel Port (from the [www.beyondlogic.org](http://www.beyondlogic.org) web site)
- Interfacing the Enhanced Parallel Port (from the [www.beyondlogic.org](http://www.beyondlogic.org) web site)
- Interfacing the Serial/RS-232 Port (from the [www.beyondlogic.org](http://www.beyondlogic.org) web site)

**Pre – Lab: Part 1**

This pre-lab assignment is due at the beginning of the first lab period for this project.

Get your team together to determine which tasks need to be done and which person will do each task. Determine how you will organize the subtasks and when they will be brought together to solve the whole problem. Present your plan to the laboratory instructor. A measure of the work includes how well you organize and break the project into appropriate pieces.

The hardware interface, via the PC parallel port, for the servo controller is designed to use EPP (Enhanced Parallel Port) mode transfers. Refer to the parallel port documentation on the [www.beyondlogic.org](http://www.beyondlogic.org) web site for information on the control signals and timing for EPP mode transfers. EPP mode hardware provides a hardware handshaking protocol for automating data transfers across the port. The parallel ports on the machines in the lab only support SPP (Standard Parallel Port) mode. It is possible to emulate the EPP handshaking protocol in software using an SPP mode parallel port. The timing can’t be duplicated exactly, but the sequence of signal levels on the control lines can be duplicated, and this works reliably to communicate with an EPP mode device connected to an SPP mode port. Sketch the timing diagrams for the four types of EPP mode cycles (Address Read, Address Write, Data Read, and Data Write). Once you understand the handshaking sequences involved, determine the sequence of values that need to be read and/or written on the SPP mode data and control ports to simulate the EPP mode cycles. Design the low level i/o routines that will be used to implement this sequence of events, to accomplish the EPP mode transfers.

Note: Implementation of the read cycles (Address Read and Data Read) requires a bi-directional parallel port and it is necessary to disable the output drivers on the PC port before starting the read cycle. This is necessary to prevent both the PC and the Digilab board from trying to drive the data lines simultaneously. At best, this contention will prevent reliable data transfer, and could potentially damage the drivers on the PC parallel port.
Once the above has been accomplished, you will understand how to transfer data from the PC to the servo controller in the Digilab board. Now, referring to the servo controller documentation, determine what sequence of address and data writes required to update the servo controller channel with a new position value. Servo controller channels 1, 2 or 3 may be used. You decide which one to use and how to update the position control register for the selected channel. Design the subroutines that will be needed to implement this level of the servo positioning software.

**Lab Procedure: Part 1**

For this part of the lab procedure, you need to write a program to step the servo motor back and forth from one extreme to the other. To accomplish this, several steps are required.

1) Implement the i/o routines to write position values to the position registers of the servo controller. The lowest level routines will perform EPP mode Address write, Data write, Address read and Data read cycles. (The address and data write routines will definitely be needed. Depending on what you think will be needed later, you may decide not to implement the address read and data read operations). In addition to the low level EPP mode access routines, you should also write the subroutine for commanding the servo to move to a new position. This routine will call the low level routines implemented above. Once these routines are implemented, you should write a simple program to prompt the user to enter position values and then move the servo to the entered position. Using this program, determine the range of input values that need to be used to move the servo to its extreme clockwise and counter-clockwise positions.

Note: It is important not to attempt to move the servo beyond its limits by writing out of range values to the position register. If the servo is left in this condition for a long time it could damage the motor or control electronics in the servo.

2) Each time that the servo is instructed to move to a new position, it will take time for the head to rotate to the new position and stop. When the closed loop control part of the project is implemented later, it will also be necessary to leave the head at a given position for a specific amount of time before moving to another position. To implement these kinds of time delays, a timer interrupt routine should be implemented to allow delays to be timed accurately in 1/18 second increments. It will also be necessary to perform some experiments to determine how much time delay needs to be allowed for the servo to move from one position to another. The time needed, will, of course, depend on how far the new position is from the old one. Using the program you wrote in part 1) above, determine the amount of time needed for the servo to move from one position to another. Ideally this routine will compute an amount of time based on how far the servo needs to move to get to the new position.

3) It will also be necessary to determine what control values to write to the servo controller to position the head to specific locations. Perform some experiments to determine what position values correspond to particular angular positions of the head and then write a subroutine that takes an angular position as an argument, calculates the correct position value to write to the servo controller, writes the value out to servo controller, and then waits the appropriate amount of time for the servo to have reached the new position. Modify the program you wrote in Item 1) above to take position values from the user in degrees and move the servo head to the specified angular position.

4) Write a program to sweep the servo head from one extreme position to the other at a particular rate. For example: Start at the extreme counterclockwise position, step to the extreme clockwise position in ten degree increments, pausing for ¼ second at each position. When it reaches the extreme clockwise position step back to the extreme counterclockwise position, and repeat. Modify the program that you wrote above to allow the user to enter the angular step amount and the time delay to pause at each position.

**Pre – Lab: Part 2**

This pre-lab assignment is due at the beginning of the second lab period for this project.
In Lab Part 1, you learned how to control the servo motor position, and wrote the functions necessary to handle that part of the project as well as the timing functions that will be needed. In this part of the project, you will learn how to communicate with the analog to digital conversion module.

The analog to digital conversion module is implemented using a microcontroller. A microcontroller is a complete computer; including CPU, program memory, data memory, and peripheral devices, implemented on a single integrated circuit. The particular microcontroller used in this project has five analog inputs, a 10-bit resolution analog to digital converter, and an asynchronous serial port. It contains 2048 instructions of program memory, and about 192 bytes of data memory. It also has several additional digital inputs and outputs as well as various other peripherals, but these features are not used in this project. The software running on a microcontroller is generally referred to as firmware. Review the documentation on the A/D conversion module hardware and firmware and become familiar with its operation.

The interface between the host PC and the microcontroller is via an RS-232 serial interface, and for this reason, it will be necessary to implement serial i/o routines. You should be able to use the serial i/o routines developed in Lab 8 previously as a starting point, though they will probably have to be modified to work successfully for this project.

The information passed via the serial port between the host PC and the microcontroller consists of commands sent by the host computer to the microcontroller and response packets sent by the microcontroller to the host PC. When a protocol, such as this, is used to communicate between two computers, there are numerous ways for errors to occur. The microcontroller might fail (or the cable become disconnected), in which case, the host CPU will stop receiving responses when it issues commands. In a noisy environment, bytes might be lost or bit errors introduced, which could cause invalid commands to be sent to the microcontroller, or invalid response packets to be received by the host PC. For the software to be robust, it is necessary for an error detection and recovery strategy to be employed to detect when these errors occur and to prevent the host PC software from failing because of them.

For example: A naive implementation might always assume that a response will come back after a command is issued. If the i/o routine simply waited until the right number of bytes were received before returning, the machine would lock up if the cable were accidentally disconnected. A more robust implementation would have the i/o routine time out and report an error after a certain amount of time had elapsed without a response.

Another, unfortunate aspect of using electronic devices (such as microcontrollers) in systems involving actuators (such as motors or solenoids) is the huge current loads that these devices can demand. The starting current for a motor can be five to ten times the running current. Unless extraordinary measures are taken in the design of the power supply, these current spikes can cause large amounts of noise and/or voltage dropouts on the power supply going to the microcontroller. If not taken into account, this can cause problems such as software glitches, and even failure of the devices to occur.

In the hardware setup for this project, the power supply on the Digilab board is used to provide power to the microcontroller and supply power to the servo motor. Even though the electric motor in these servos is a very small motor its starting current is still larger than the power supply on the Digilab board can supply. Using an oscilloscope, voltage dropouts of as much as two volts have been observed to occur on the power pins of the microcontroller when the servo motor moves to a new position. A voltage dropout of this magnitude can cause the microcontroller to reset. Even worse, the rise time of the power supply voltage following the dropout doesn’t meet the rise time specifications for the microcontroller, and therefore, it isn’t guaranteed to reset cleanly. This is a fact of life that must be dealt with when designing this kind of project. It must be dealt with in a suitable way. In the case of this lab project, if the microcontroller malfunctions, no great harm can be done. In other circumstances, such as the microcontroller controlling a heart defibrillator, a problem such as this that isn’t dealt with correctly could kill someone.

1) Determine the appropriate serial communication protocol information (e.g. baud rate, number of data bits, etc) used by the microcontroller. Referring to the PC serial port documentation, determine the
values that need to be programmed into the 16550 UART registers to configure the PC serial port to match the requirements of the microcontroller serial protocol.

2) Determine what changes will need to be made to the serial i/o routines you will use. Consider the need for the ability to have a serial read request, time out if no byte becomes available within a reasonable amount of time.

3) Study the protocol implemented by the microcontroller and determine what command packets need to be sent to the microcontroller to read each analog input channel. What will the corresponding response packet sent by the microcontroller look like? What will your software have to do to translate the packet sent by the microcontroller into information that will be usable by your software.

4) Determine the possible ways that errors can occur in the command/response protocol between the host PC and the microcontroller. Design an error detection and recovery strategy for dealing with the possible errors that could occur. Are there ways that undetectable errors could occur? Consider how to deal with this in the higher level software so that an undetected error will not cause the control software to malfunction.

5) Design a strategy for dealing with the fact that the microcontroller may be randomly reset each time that the servo motor is commanded to move to a new position. Notice in the documentation for the servo controller that there is a microcontroller control register.

Lab Procedure: Part 2

For this part of the lab procedure, you need to write a program that can communicate with the A/D conversion module. It must be able to establish communications with the microcontroller, issue commands to read the analog inputs, receive the response packets and then print the result in a meaningful way to the user.

1) Implement the modifications to the serial i/o routines (or write new ones) that support the requirements for this project.

2) Write the functions necessary for issuing commands to the microcontroller and receiving the response packets. Make sure that you implement the error detection and recovery strategy you designed in the Pre-Lab in these routines. At the very least, these routines need to detect the errors and communicate the error information so that higher level routines can perform the recovery.

3) Write a program that sends commands to the A/D conversion module to read the values on each of the light sensors, get the response packets, translate the response packets into binary values, and then print the resulting values.

4) Using the program that you wrote for item 3) above, perform some experiments to determine what range of values will be received for various light sources at various distances. You will need to have an idea about what range of values to expect for various conditions of the light source when you implement the closed loop control program for the final part of this project. For the purpose of this exercise, a small flashlight works well as a light source.

Pre – Lab: Part 3

There are no special pre-lab requirements for this part of the project.

Lab Procedure: Part 3

This is the final part of the project. In this part, you will make use of the components that you built in the previous two parts. You will implement the closed loop control program. Your control program should have the following features:

1) Initially, it should sweep the sensor head back and forth looking for a bright light source
2) When a bright light source is detected, it should center the sensor array on the bright light source. (i.e. the center light detector should point at the light source)

3) If the light source moves, the sensor head should track its movements, keeping the sensor head centered on the light source at all times. Note: because of various delays inherent in the system, it will probably not be possible to track the light source if it moves very quickly.

4) If the light source goes away, the program should go back to the mode where it sweeps the head back and forth looking for a new bright light source.

5) A graphic display should be maintained on the PC monitor that shows a graphic indicator of the direction that the sensor head is pointing, and a bar graph display showing the light intensity being detected by each optical sensor. This display should be updated continuously.

**Lab Report**

In the final report, elaborate on the design requirements for the project, the division of labor, and project management. Be sure to follow the standard report format.
Appendix A:

Servo Motor Controller Reference Guide

Introduction

The servo motor controller is implemented in the Xilinx FPGA on the Digilab board. Before using the servo controller, it will be necessary to configure the gate array with the correct design file. An configuration utility program will be provided for configuring the gate array.

The R/C hobby servos used in this lab project use a Pulse Width Modulated (PWM) signal to indicate the position that the motor should hold. The pulse with varies from approximately 0.5 milliseconds to 1.5 milliseconds, and is repeated at about a 50 Hz rate. A pulse of 0.5 milliseconds duration corresponds to the motor being all the way clockwise. A pulse of 1.5 milliseconds corresponds to the motor all the way counter-clockwise. The motor is able to travel approximately 190° from its clockwise limit to its counter-clockwise limit.

The servo motor control hardware configured into the gate array provides four channels of PWM control that can control up to four servo motors simultaneously. In the hardware configuration available in the EE314 lab, only three of the four control signals are brought out to the interface board, and so channels 1, 2 or 3 are usable. The pulse width, and hence, the servo position for a given channel is controlled by writing a position value into the control register for the channel. The position value is a 16 bit binary number in the range 170-1700. A value of 170 corresponds to approximately a 0.5 ms pulse, a value of 1700 corresponds to approximately a 1.5 ms pulse. A position value of 0 written to a control channel will cause the channel to stop generating servo control pulses.

The interface to the servo controller is through an EPP mode parallel port from an IBM PC type computer. The EPP interface uses an address register and multiple data registers. The address register is written using an EPP Address Write transfer cycle. An EPP Data Write cycle will cause the data to be written to the servo controller register specified by the last address write.

Servo Controller Register Map:

<table>
<thead>
<tr>
<th>Address Register</th>
<th>Data Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Channel 1 low byte</td>
</tr>
<tr>
<td>1</td>
<td>Channel 1 high byte</td>
</tr>
<tr>
<td>2</td>
<td>Channel 2 low byte</td>
</tr>
<tr>
<td>3</td>
<td>Channel 2 high byte</td>
</tr>
<tr>
<td>4</td>
<td>Channel 3 low byte</td>
</tr>
<tr>
<td>5</td>
<td>Channel 3 high byte</td>
</tr>
<tr>
<td>6</td>
<td>Channel 4 low byte</td>
</tr>
<tr>
<td>7</td>
<td>Channel 4 high byte</td>
</tr>
<tr>
<td>8</td>
<td>Microcontroller control register</td>
</tr>
</tbody>
</table>

Register Description

Servo Position Registers:

Each servo channel requires a 16 bit position value. The low byte of the position value is written to the low register, the high byte of the position is written to the high register. To write a control value to a given channel perform the following operations:

1) Write the address of the position low register to the address register via an Address Write cycle
2) Write the low byte of the position value via a Data Write cycle
3) Write the address of the position high register to the address register via an Address Write cycle
4) Write the high byte of the position value via a Data Write cycle

Microcontroller Control Register

The register at address 8 is a control register for controlling the microcontroller on the A/D converter card plugged into the Digilab board. The following describes the control bits in this register:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

X - undefined
Reset - This bit is used to reset the microcontroller. A value of 1 in this bit will cause the microcontroller to be held reset. A value of 0 will allow the microcontroller to run.
Appendix B:

**A/D Converter Firmware User Guide**

Rev. 1.1: 04/10/2002  
Gene Apperson, WSU

**Introduction**

The ADCONV module is a low-speed analog input module that works in conjunction with a Digilab XLA board. It provides a serial interface for communications with control software running on a host PC, and supports five analog inputs and 6 digital input/outputs. Commands can be issued via the serial port to read the analog channels.

The ADCONV module is implemented using a Microchip PIC 16F870 microcontroller. The 16F870 provides 5 analog input channels, 12 configurable digital i/o’s, three timer channels and an asynchronous serial port. The analog channels provide 10-bit resolution, with an input range of 0 to 5 volts.

**Interface Protocol**

The host computer communicates with the ADCONV device by issuing commands via the serial interface and receiving responses over the serial interface. The command protocol is defined so that all commands are sent as single ASCII characters. The responses depend on the command the certain internal mode settings, but are generally sequences of ASCII characters.

The serial port is configured for 9600 baud, 8 data bits, 1 stop bit, no parity.

**Commands**

Each command to the microcontroller is a single ASCII character sent over the serial interface. For each command sent, the microcontroller will respond with a response string. The following describe the actions and responses of each command:

@ Verify Status

This command is used to verify that the microcontroller is connected to the serial interface and receiving commands. The microcontroller will respond with a ‘!’ character to indicate that it received the command.

A Read Analog Channel 1  
B Read Analog Channel 2  
C Read Analog Channel 3  
D Read Analog Channel 4  
E Read Analog Channel 5

These commands will cause the microcontroller to read an analog sample value from the specified analog channel and report the result. The format of the response string depends on the current analog reporting mode.

In ASCII HEX reporting mode, the microcontroller will return the analog sample value in a packet with the analog value encoded as a 3 digit ASCII hex number. The packet will be preceded by a channel indicator and terminated by a ‘@’ character. The channel indicator will be a lower case version of the command character for the read command. For example, the response to an A command would look like: a01F@. The ‘a’ indicates that the following value is a channel 1 value.
The ASCII hex digits 01F give the sample value for that channel, and the ‘@’ character indicates the end of the transmission.

In Binary reporting mode, the microcontroller will return a two byte binary number, with the least significant byte sent first, followed by the most significant byte. There is no channel number indicator or end of transmission indicator.

F  Set Binary Reporting Mode

This command will place the microcontroller in binary reporting mode. All future Read Analog Channel command will respond with a two byte binary number for the channel value. The controller will respond with a ‘@’ character to acknowledge receipt of the command.

G  Set ASCII Hex Reporting Mode

This command will place the microcontroller in ASCII hex reporting mode. All future Read Analog Channel command will respond with a 5 character ASCII hex response string as described above under the Read Analog Channel command descriptions. The controller will respond with a ‘@’ character to acknowledge receipt of the command.

H  Query Analog Reporting Mode

This command can be used to question the microcontroller as to which reporting mode is currently active. The controller will respond with an ‘A’ character if it is in ASCII hex mode, or a ‘B’ character if it is in Binary mode.

I  Reset

This command will cause the microcontroller to jump to its power on initialization code. This will result in all internal memory locations and modes to be reset to the power on state. This command is not acknowledged directly, but the normal reset sequence causes the microcontroller to send a ‘!’ character after it has performed all initialization. The receipt of the ‘!’ character indicates that the microcontroller received the command and has completed the reset process.

L  Loopback Test

This command causes the microcontroller to enter loopback test mode. This mode can be used to verify that the microcontroller is active and that the serial interface is working. On receipt of the ‘L’ command, the microcontroller will send a ‘>’ character to acknowledge receipt of the command and entry into loopback mode. While in loopback mode, the microcontroller will send every character it receives on its serial input back out on its serial output, echoing each character back to the sender. Loopback mode can be ended by sending an ESC character (hex value 01Bh). On receipt of the ESC character, the microcontroller will send an ‘@’ character to acknowledge, and then return to normal operating mode.