

A TRIDENT SCHOLAR PROJECT REPORT

NO. 243

"Distributed Processing Using Single-Chip Microcomputers"



UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

This document has been approved for public
release and sale; its distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE

1996

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

Distributed processing using single-chip microcomputers

5. FUNDING NUMBERS

6. AUTHOR(S)

Pritchett, William C.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION
REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

United States Naval Academy
Annapolis, MD 21402

10. SPONSORING/MONITORING
AGENCY REPORT NUMBER

USNA Trident Scholar
report; no. 243 (1996)

11. SUPPLEMENTARY NOTES

Accepted by the U.S.N.A. Trident Scholar Committee

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

UL

13. ABSTRACT (Maximum 200 words)

This project investigates the use of single-chip microprocessors as nodes in a token ring control network and explores the implementation of a protocol to manage communication across such a network. A control network is useful when the event to be controlled is located at some distance from the inputs required to control it; likewise, a control network is useful when an application receives inputs from more sources than a single microprocessor is capable of handling. Such a network allows nodes to share only the information that is essential for each to perform, eliminating the need for a powerful and costly computer. This makes it extremely effective in a wide variety of applications ranging from missiles to home security systems to "smart" automobiles. One type of control network is the token ring network, where each node is connected serially with the node immediately following it and the one preceding it. Its efficiency, simplicity, and determinacy make it an excellent choice in a small control network. A specific scenario is examined where the position of a marble along a motor-driven track is controlled using inputs from a user operating a PC as well as a microcomputer-driven interface module, an optical encoder mounted on the motor, and a camera located above the track. Using the information of the state variables as well as preferences of the user, a digital control system is developed to move the marble to the proper position.

14. SUBJECT TERMS

Networks; Distributed processing; Token ring; Microprocessors; Serial communications; Control systems

15. NUMBER OF PAGES

87

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT

UNCLASSIFIED

18. SECURITY CLASSIFICATION
OF THIS PAGE

UNCLASSIFIED

19. SECURITY CLASSIFICATION
OF ABSTRACT

UNCLASSIFIED

20. LIMITATION OF ABSTRACT

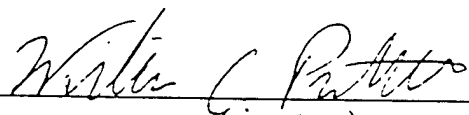
UL

U.S.N.A. --- Trident Scholar project report; no. 243 (1996)

"Distributed Processing Using Single-Chip Microcomputers"

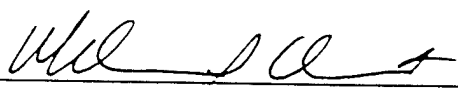
by

Midshipman William C. Pritchett, Class of 1996
United States Naval Academy
Annapolis, Maryland


(signature)

Certification of Adviser Approval

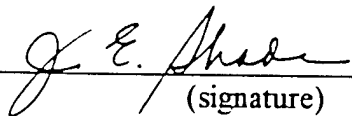
Assistant Professor William I. Clement
Department of Weapons and Systems Engineering


(signature)

1 MAY 1996
(date)

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Chair, Trident Scholar Committee


(signature)

1 May 1996
(date)

ABSTRACT

This project investigates the use of single-chip microprocessors as nodes in a token ring control network and explores the implementation of a protocol to manage communication across such a network.

A control network is useful when the event to be controlled is located at some distance from the inputs required to control it; likewise, a control network is useful when an application receives inputs from more sources than a single microprocessor is capable of handling. Such a network allows nodes to share only the information that is essential for each to perform, eliminating the need for a powerful and costly computer. This makes it extremely effective in a wide variety of applications ranging from missiles to home security systems to "smart" automobiles. One type of control network is the token ring network, where each node is connected serially with the node immediately following it and the one preceding it. Its efficiency, simplicity, and determinacy make it an excellent choice in a small control network.

A specific scenario is examined where the position of a marble along a motor-driven track is controlled using inputs from a user operating a PC as well as a microcomputer-driven interface module, an optical encoder mounted on the motor, and a camera located above the track. Using the information of the state variables as well as preferences of the user, a digital control system is developed to move the marble to the proper position.

Keywords:

- networks
- distributed processing
- token ring
- microprocessors
- serial communications
- control systems

Table of Contents

Chapters	Page
1. Problem Statement.....	3
2. Background.....	5
2.1 Networking.....	5
2.2 Serial Communication.....	8
2.3 DC Motors.....	10
2.4 Microcontrollers.....	12
2.5 Image Processing.....	13
3. Dynamics and Control.....	15
4. Token Passing and Network Protocol.....	20
5. Individual Nodes.....	28
5.1 Node 2: Interfacing.....	28
5.2 Node 3: Image Processing.....	29
5.3 Node 4: Motor Control.....	34
6. Conclusions.....	39
7. References.....	41
 Appendices	
A1 Calculating System Equations.....	42
A2 Integrator State Feedback Design.....	45
A3 Additive Clock Operations.....	47
A4 Node 2 Source Code.....	48
A5 Node 3 Source Code.....	65
A6 Node 4 Source Code.....	73

1. Problem Statement

The purpose of this project is to create a serial network of microprocessors capable of performing control operations using sensors from remote locations. This network will be involved in controlling the position of a spherical marble along a tilting beam driven by a DC motor. The desired position of the ball can be specified by the user using some manner of input, and the actual position will be determined using image data from an analog camera. A microprocessor responsible for controlling the motor will operate using a control law also specified by the user. This will require that one node in the network be a PC on which the user may test his control law before applying it.

There is a need for a protocol designed with the control network in mind, ready to handle frequent bursts of data. A major goal in this project is to explore a common protocol whereby any number of single-chip microprocessors may communicate with one another, making the project more universal in scope. This protocol should make it easy to add or remove nodes anywhere in the network as different sensor output stations are needed. The number of microprocessors in the network will vary according to the requirement of the specific problem at hand. This number will grow as the project grows in scope and complexity. The exact number of nodes is not important, but it must be a number great enough to test the network's capabilities of communication.

An important issue in the project is to make certain that each microprocessor is able to continue with its assigned task even as it receives data from other nodes in the network. The purpose of this project is not only to create a message protocol, but to allow microprocessors to perform useful work and still participate in this network. In all of the microprocessors, it takes some time to receive and process an incoming message and prepare and transmit an outgoing one. The critical issue is what a microprocessor is to do if it is about to perform an important function and receives an incoming message. For example, the microprocessor responsible for monitoring the position of the ball will take position data at a periodic rate and may be forced to miss a pass if it is processing information from the network. Perhaps it is acceptable to miss a pass and use the data obtained on the last pass, or perhaps it is not. This issue must be dealt with for each of the microprocessors in the network.

The specific problem to test the ability of this network protocol is the position control of a ball which moves along a tilting track. Figure 1.1 shows the track and ball combination with the two position states, x and θ , labeled. The user can specify a desired position either through the use of a personal computer or through the turning of a shaft with a shaft encoder attached. By using position and velocity data of the ball from a camera positioned above the track and angular position and velocity data from a shaft encoder attached to the motor, a control law using state feedback will be designed to achieve proper response from the motor in order to balance the ball at the desired position.

Four nodes will interact in this network: three single-chip Peripheral Interface Controller (PIC) microprocessors and one personal computer (PC). The PC will allow the user to view information on a monitor as well as to update the desired position of the ball or change the control law that the problem is following. One PIC will be dedicated to controlling the motor which turns the track. A second PIC monitors data from the

camera, while a third serves as a link between the network and the PC and as a secondary input source for the desired position. If these four nodes are able to act successfully and effectively control the position of the ball, then the network protocol will have proven itself effective.

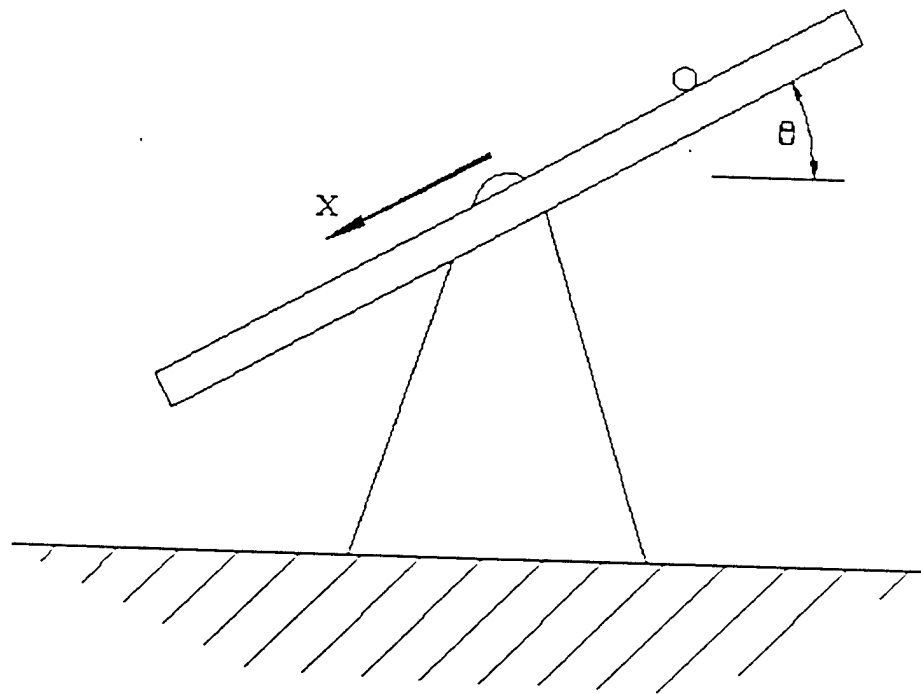


Figure 1.1. Diagram of Mechanical System

2. Background

2.1 Networking

A computer network is a system consisting of two or more individual stations, or nodes, configured in such a way as to allow data to be exchanged between the nodes. This information sharing is governed by a network protocol, a set of rules which govern all facets of the communication — it “defines connectors, cables, signals, data formats, and error-checking techniques as well as algorithms for network interfaces and nodes, allowing for standard — to within a network — principles of message preparation, transfer, and analysis on different levels of detail” [1].

The choice of a protocol for a particular network involves many factors, which will be discussed later. In general, there are two different kinds of networks: centralized networks and decentralized (or distributed) networks. Centralized networks contain a master node, which governs all traffic on the network. Typically each of the other nodes, termed slaves, can communicate only with the master node and not among themselves. The master node is often a much more powerful processor than the slave nodes because it is responsible for performing most of the work. In a distributed network, on the other hand, each of the nodes has the same right to use the network as any other node. The processors used in each node are usually very similar in power, as each node is responsible for performing a relatively equal amount of work.

A comparison between centralized networks and distributed networks reveals advantages and disadvantages of each. The protocol for centralized networks is usually much simpler than the one for distributed networks [1]. On the other hand, the response times of distributed networks are typically faster because each node can communicate directly with the other nodes instead of passing all information through a master node. Distributed systems also allow each function to be performed by a processor which has the necessary capabilities for that specific function instead of a single complex computer performing all operations, thus avoiding unnecessary complexity for simpler operations [2].

A network can also be categorized according to the geographical distribution of its nodes. A wide area network, or WAN, spans great distances and may contain nodes scattered across a country or a continent. A local area network, or LAN, on the other hand, is a network in which there is a much shorter distance between each node, often on the order of several kilometers. Before the 1970's most LANs used as data networks were centralized systems. Since then many different distributed network protocols have been introduced, such as Xerox's Ethernet, Datapoint's Arcnet, and IBM's Token Ring [1].

LANs may further be divided into two very distinct categories according to the type of operation the network is to perform. These categories are data networks and control networks. A data network is what is often pictured when the term network is mentioned—several PC's sharing files in an office via a central server, for example. This type of network differs drastically from the network that is useful to systems engineers —

the control network. In general, data networks send large packets of data infrequently, and require a high data transmission rate when sending these data. Control networks, on the other hand, receive huge bursts of data packets, dubbed 'ordered traffic' [3]. These bursts occur very frequently and are very short in length, usually less than 20 bytes. A control network must operate with nodes which are performing time-critical functions and cannot pause indiscriminately for data reception or transmission.

The networks mentioned above—Arcnet, Ethernet, and Token Ring—were all introduced for use in data networks, but similar frameworks have also been used in control networks. There are several factors to consider in determining the framework to use in a control network. These factors include interoperability, efficiency, determinacy, robustness, and cost per node.

Interoperability refers to the ability of the network to perform when not all nodes or connections are alike. "The network protocol must be open. Its availability to anyone on equal terms is crucial to the ability of products from different manufacturers to work together on the network" [3]. This requires an acceptance of a standard protocol by the manufacturers. A second issue of interoperability is the need to work in varied operating environments with mixed media access. "For example, portions of a system may require expensive fiber in noisy environments, while other portions can tolerate low-cost twisted pair wires in benign environments" [4].

The protocol efficiency of a control network refers to the ratio of the number of message bits delivered compared to the total bandwidth of the network. Since only some of the bits in the message are data bits and the others are overhead bits used for message routing and other network tasks, an obvious way to optimize efficiency is to reduce the number of overhead bits required to send a constant number of data bits. Efficiency is also a function of packet size since the overhead is often a fixed length. For example, a thirty-two bit message may contain only eight bits of data and twenty-four bits containing node address and protocol instructions. Protocol efficiency is generally divided into two categories: heavy traffic efficiency and light traffic efficiency. As will be seen, some protocols work well in one but not the other.

A third property of a control network that demands attention is determinacy. "Determinacy, or the ability to calculate worst-case response time, is important for meeting the real time constraints of many embedded control applications" [4]. Most systems contain some sort of prioritization scheme where nodes that control tasks requiring immediate execution may take temporary control of the network to send their traffic. Determinacy is extremely important so that a network architect will know whether a likelihood exists for the entire network to freeze as a result of multiple nodes attempting to gain access to it.

Many network applications require robust operation in order for success. A network is robust if it can respond easily to unforeseen and unwanted changes in the network such as an added node, deleted node, or lost token, for example. It is also desirable for a network to be able to respond quickly in the event that a power surge or glitch causes a reset.

Finally, the cost per node is one of the paramount considerations in network design. Different topologies and protocols require differing amounts of hardware and software resources depending on the simplicity or complexity of the protocol. For cost-

sensitive high volume applications a simpler protocol is often desirable; a more complex protocol is useful when application growth is expected [4].

It was mentioned above that no definite standards for control networks have yet been introduced. However, there has been experimentation in several broad areas including polling, time division multiple access (TDMA), token ring, token bus, binary countdown, carrier sense multiple access with collision detection (CSMA/CD) and carrier sense multiple access with collision avoidance (CSMA/CA). These seven protocols each have advantages and disadvantages according to the application of the network [4].

Polling is a popular method for control networks because of its simplicity and determinacy. One processor (node) typically polls each of the other processors in turn to determine if they have traffic to send and then gives them permission to do so. This is ideal for applications with centralized data acquisition where prioritization and node-to-node communication is not required. Polling, however, requires a large amount of network bandwidth due to its two way communications and is therefore unacceptable for high speed applications [4].

The TDMA protocol is used extensively in aerospace operations. Similar to polling, a TDMA network functions by a master node sending a synchronization signal to each of the other nodes so they are in essence running on the same clock. Each node is then assigned a time slice when it may transmit. TDMA uses much less bandwidth than polling, but node costs are increased because each node requires a stable clock for synchronization. An additional disadvantage is the need for fixed-time transmissions so the nodes do not exceed their allotted time [4].

A token ring network consists of nodes connected to one another in a ring shape. There is no common bus in the token ring protocol; each node is connected only to two other nodes. A token message is sent from node to node around the ring until a node has something to send. This node then replaces the token with its message which is transferred from node to node until it reaches its destination. The determinacy of this protocol is very good since worst-case token passing time can easily be calculated. Throughput efficiency during both heavy and light traffic situations is very good since idle token passing decreases as traffic increases. Looking to the future, the token ring network's point-to-point connections adapt easily to the growing field of fiber optics. One important disadvantage is that a failure of any one node in the network may cause the entire network to crash. This may create problems that are more difficult to detect than when one node on a common bus is malfunctioning [4].

A token bus network is very similar to a token ring network in that a token is passed from node to node in a virtual ring. The nodes are actually connected by a common bus so each message is sent to all the nodes before the next message is sent. This requires a great deal more time than the process of direct token passing in a token ring network. It is superior in that if one node fails the network may still function [4].

In the binary countdown protocol, each node waits for a clear channel before transmitting. Each node is assigned a certain binary number corresponding to the priority of the traffic it sends. When it transmits, the node first sends this information on the bus. Two nodes attempting to transmit at the same time resolve their conflict by sending out one bit at a time of their assigned number to determine which message has priority. Typically, a binary 1 indicates a higher priority than a binary 0. For example, imagine

three nodes with priorities 110, 111, and 010 attempting to transmit at the same time. They each transmit their first bit, and the network sees that two are transmitting 1's so it locks out the 010 node because its traffic has a lower priority. It then continues checking the bits of the remaining nodes until it reaches a point where all nodes are locked out but one. This protocol, also called the bit dominance protocol, is used infrequently because of the difficulty in adding new nodes and the required complexity of the connections [4].

The CSMA/CD protocol allows an almost unlimited number of nodes connected on a common bus. New nodes may be added or deleted without new initialization. Each node simply sends a message across the bus when it needs to. If two nodes attempt to transmit at the same time a collision occurs. Analog circuitry detects this collision, and each node transmits its data again after a period of time. This protocol is very inefficient under heavy traffic conditions, and its determinacy is very poor since multiple collisions may keep occurring, effectively freezing the network. Ethernet, the popular data network protocol, is based on this protocol. Very similar is the CSMA/CA protocol, which combines facets of the CSMA/CD and TDMA protocols [4].

2.2 Serial Communication

Whatever protocol is used in the network, there must also be a standard for communication of information between the nodes. Fortunately, there are several standards to which most existing applications adhere. There are still, however, choices to be made when dealing with intercomputer communications.

There are two different methods by which one can communicate information. The first is called parallel communication. In parallel communication, whole words of information are sent at one time from one computer to another. A computer word is usually 8 or 16 bits long so two computers would need 8 or 16 connections for one to transmit a word in parallel. Serial transmission, on the other hand, sends out one bit of information at a time instead of one byte at a time. Therefore, only one data path is required between nodes. There are several advantages and disadvantages to both. Parallel communication is much faster than serial communication since information is essentially moving eight times as fast. There is a price for this speed increase, however. Eight times as many connections are required between computers. This can become bulky and very costly, especially in a noisy environment where a quality medium such as fiberoptic cable is used. In general, parallel communication is reserved for short paths such as nodes in the same room, whereas serial communication is more practical for longer distances such as those between nodes in a typical control network.

The problem with transmitting information serially is that since there is only one data path for information to travel, a large number of bits must travel along the same wire at different times. This results in a problem if the receiving node misses one or more of the bits in the transmission process and then decodes the message incorrectly. On a broader note, the receiving node must know when the first bit of the message is going to be sent. The communications interface can accomplish this in two ways. First, the processors can agree that the transmission will occur at a specific time, every millisecond for example. This requires that one of the processors, termed a master node, provide a

clock signal to the slave node and send or receive data on the rising or falling edges of this clock. This is called synchronous transmission. The other way for the receiving node to know data is coming is for the transmitting node to send a start bit immediately prior to sending data. The processors here do not necessarily need to be running at the same clock speed, but must be capable of operating at a common speed (i.e., at the lower of the two). This is known as asynchronous transmission. Because of the fewer connecting wires that it requires, asynchronous transmission is more practical over long distances, and therefore more widely used in network applications.

As mentioned, asynchronous serial transmission works by one processor signaling another with a start bit that it is going to begin transmitting. This start bit is typically a low state. The other processor, which knows how fast the information is coming and how many bits of data are coming at a time (usually eight), reads each bit. After all of the data bits are transmitted, the transmitting processor sends a stop bit of opposite state as the start bit to let the receiver know the transmission is complete. This stop bit lasts for an indefinite period, with a minimum time specification to allow the receiving computer time to store data before the next byte arrives. These ten or so bits of information—the start bit, data bits, and stop bit—are known as a frame [5]. A typical frame is shown in Figure 2.1 below.

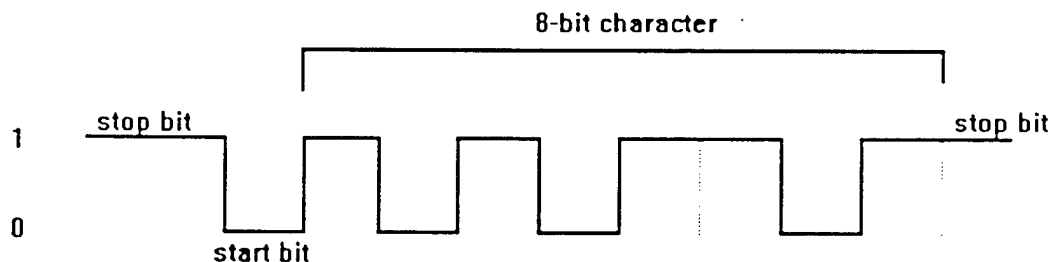


Figure 2.1. A Sample Frame

If the start bit or stop bit are not received correctly, a framing error occurs. The receiver throws out the entire character because it could not be sure it received the right data. This may not be the end of the problem, however. If a start bit is received incorrectly as a high instead of low, the first bit of data that is low will be assumed to be the start bit and the wrong character will be sent. This usually happens if a glitch on the line generates a false start. If the bit after the stop bit is also high, the receiver will think this data is valid and accept it. This is known as a synchronization error.

There are many other errors that can occur as a result of garbled transmissions. If one of the data bits is not transmitted properly, the receiver will simply receive one bad character. This is known as a single bit error. One way to catch this error is by performing a character parity check [5]. In a parity check, a parity bit is transmitted immediately after the data bits and tells whether there were an even or odd number of 1's in the data. The receiver looks at the data and calculates this value also and then compares its result with the parity bit. This is one of the simplest error checking protocols, but also one of the most commonly used. A single parity bit will detect an odd

number of bit errors, but cannot determine which bits were in error. It is often called a “single-error” detecting scheme. It is impossible to detect all possible sources of error, but it is important to realize that any communication error may result in a momentary or total failure of the control network.

2.3 DC Motors

A DC motor is a device that converts electrical energy to mechanical energy by current flowing through a magnetic field. A model of a DC motor is shown below in Figure 2.2. The components L_a and R_a represent the inductance and resistance of the armature, respectively.

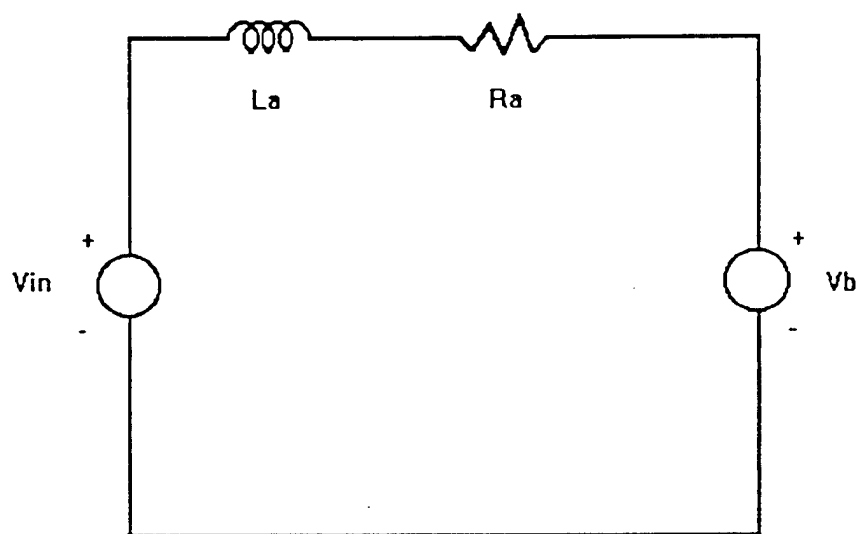


Figure 2.2. A Model of a D.C. Motor

As the applied voltage V_{in} increases, the current that flows through the armature increases as well. Since a direct relationship exists between the armature current and the output torque of the motor, this will cause an increase in motor velocity as well. This is typically how a DC motor is controlled — by varying the voltage applied to its armature. This may be done with a transistor or rheostat. The problem with this is when the transistor operates in its linear region. If the motor is to run at half the applied power, the other half of the power must be dropped across the transistor and therefore wasted [6].

A better way to control the voltage applied to a DC motor is by using the transistor merely as a switch and sending a pulsewidth modulation (PWM) signal. The PWM signal can be varied in duty cycle to send a fraction of the power to the motor without wasting any power. Instead of half the power being sent to the motor and half

dissipated across the transistor, the full power of the battery is sent to the motor for half the time.

The theory behind the PWM signal is the Fourier Transform. By analyzing a train of pulses with frequency $f = f_0$, we obtain the frequency domain spectra shown in Figure 2.3 (a). As can be seen, the signal contains a DC value (which happens to be equal to the average value of the function) and discrete values at whole number multiples of the frequency of the pulse train. A motor acts as a first order low pass filter (shown in Figure 2.3 (b) below) and removes any frequencies higher than its cutoff frequency, f_c . What is left, assuming f_0 is greater than f_c , is a single DC value equal to the average value of the pulse train. The inverse Fourier Transform of this is simply a step function of height equal to the DC value. This step function then drives the motor as if a constant voltage equal to the value of the step function were applied directly.

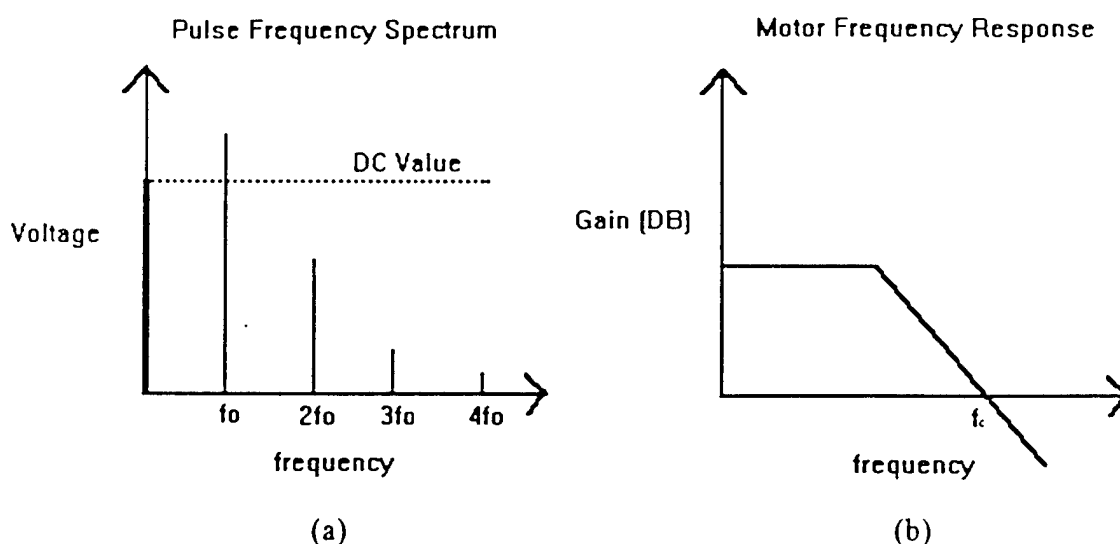


Figure 2.3. Frequency Spectra of PWM Signal and Motor Response

An equally important consideration as controlling the speed of the motor is controlling its direction. This involves four switching devices set up in an H-bridge configuration, with two being turned on at any one time. Fortunately there exists several H-bridge motor drivers with this circuitry built in. All of these drivers are fairly similar in terms of the signals required to control them. Each requires a power source (usually 12-24 V) and ground as well as a Phase and Enable signal. The phase signal tells the motor driver which direction the motor will turn, and the enable signal allows the power signal to be connected to the motor armature, i.e., enables the motor to run.

There are two ways to drive a motor in this fashion. The first, called the sign/magnitude mode uses the Phase bit to determine direction and the Enable bit to determine speed. In this mode, the PWM signal is sent to the Enable bit. The duty cycle of the PWM signal determines the DC value sent to the motor. It can vary over its full

resolution, controlling the speed from full on (100% duty cycle) to full off (0% duty cycle). The motor is thus only turned on as needed [6].

The alternate mode is the Locked Antiphase. In this mode, the Enable signal is left at high—the motor is “full on.” What is varied using PWM is the Phase or direction signal. A PWM input of 50% duty cycle will enable the motor half the time in each direction at full speed, thereby canceling one another. Any PWM duty cycle greater than 50% will cause the motor to turn in the positive direction, with 100% duty cycle causing the motor to actually turn at full speed in the positive direction. The same is true for the negative direction in the lower half of the PWM values. The advantage of this method is that full torque always exists on the motor. This allows the motor to respond quickly to a change in commanded velocity. The disadvantage is that this method has only half the resolution of the sign/magnitude mode since the duty cycle ranges only 50% from full off to full on. Also, greater power consumption results because current is always flowing, even when the motor is stopped. The reason for this effect is that the Enable bit is always set. This effectively sends no information to the motor driver, where before, the setting or clearing of the Enable bit determined the speed of the motor [6].

2.4 Microcontrollers

For embedded applications there are several choices of microprocessors for use, but they can essentially be divided into two groups: 1) microprocessors from makers such as Intel or Motorola, which are adapted by third party vendors for use in embedded applications and 2) dedicated microcontrollers which are not capable of running an entire PC. The real division between a microprocessor and a microcontroller is that the former requires external RAM and ROM to be added while the latter already possesses required memory on a single chip. Each has its own advantages and disadvantages.

The Intel 80x86 family of microprocessors is the same line which make up the heart of most personal computers in the world. These are very powerful and can often be run at clock speeds exceeding 100 MHz. They also possess, in their embedded mode, capability to perform many of the same functions as when they run a PC. For example, many include connections for a floppy disk or VGA display. Another advantage is that familiar compilers that run on PC's will compile programs for these embedded processors as well. This often saves the user from having to learn another language for his embedded applications. The main disadvantage of such chips is the high cost. Most are in the range of \$400 to \$600. For a multi-node network, this can quickly make a project unfeasible. The boards on which the Intel or Motorola dedicated chips come are also very bulky in size. This is because the Intel chip itself is only a microprocessor and needs supporting chips such as A/D converters and display adapters. Finally, it is difficult to exploit the full potential of an Intel computer in embedded applications. This often makes the high cost not very worthwhile.

Compared to the Intel line of microprocessors, dedicated microcontrollers seem at first severely lacking in power. Their maximum clock speeds usually do not exceed 25 MHz. This is a bit misleading, however. With careful assembly language programming, they can often run the same operations as a more powerful computer in fewer instructions.

Most are also typically smaller than comparable Intel-based microcontrollers, with the smallest taking up no more space than a single chip. This makes these dedicated microcontrollers much more desirable for space-conscious applications. They also typically cost less than one tenth the amount of Intel microprocessors, making them a good choice for applications requiring many nodes.

At this time, one of the best of these microcontrollers is the Peripheral Interface Controller (PIC) family of microcontrollers. The PIC16Cxx series is representative of a new breed of microcontroller. These particular devices are 18- to 40-pin chips and require only one external crystal oscillator to become fully functioning computers. The RISC (Reduced Instruction Set Computer) architecture contains only 35 single-word instructions, each of which executes in one clock cycle (except the branching instructions such as "goto" and "call" which require two cycles). The speed and simplicity of this design makes these processors ideal for high-speed control applications. In addition, this particular family of processors includes several varieties which have onboard analog-to-digital (A/D) capability, thus further increasing their usefulness in the control field. Often, control problems require that analog data be collected and control actions be based upon such data. The PIC16C71 and PIC16C74 are ideal for such applications. Each has a single A/D unit which is multiplexed to multiple pins on the chip, thus allowing sequential reading of four or eight analog values, respectively. The PIC16C71 operates on a 16 MHz clock frequency, executing each instruction in 250 ns and is capable of A/D conversions at the rate of 30 kHz. The PIC16C74 operates at 20 MHz and has an A/D converter which can perform 62,500 conversions per second (16 μ s each) [7]. These processors range in price from approximately \$10 to \$50 and are the processors chosen for this research.

2.5 Image Processing

An image is defined as a "two-dimensional light intensity function $f(x,y)$, where x and y denote spacial coordinates and the value of f at any point (x,y) is proportional to the brightness (or gray level) of the image at that point" [8]. Images are analog in nature; the brightness f may take on any value at any point and the difference between the values at different points may be infinitely small. To make image data useful to computers, an image must be discretized or digitized so that information about the brightness at any point can be communicated digitally to the computer. The result of this operation is known as a digital image.

A digital image processor is the core of an image processing system and contains components that perform four basic functions—image acquisition, storage, low-level processing, and display. The module responsible for acquiring and digitizing the image is known as a frame grabber, so named because modern image processors are capable of digitizing a TV image in one frame time [9]. The image data are then stored in memory called a frame buffer. From here a processing module performs low level arithmetic and logic operations on the image before it is sent to the display device.

A digitizer divides the two dimensional image into an array of pixels, thereby discretizing the number of possible coordinates x and y . Each digital coordinate is known

as a pixel. In order to speed calculations, the array is often square with the number of rows and columns being a power of two, for example 512 X 512 pixels. The image is further discretized in the number of brightness levels allowed each pixel. This also typically is a power of two. A binary image is a digital image with only two brightness levels—0 and 1. With the smallest amount of information per pixel, a binary image is the simplest, yet lowest clarity digital image. It is essentially a black and white image, with no other gray levels. A 256 gray level image, on the other hand, contains much more information per pixel and is therefore much clearer. It however requires more storage and communication time to make use of this data.

The image on which a frame grabber operates is the output of a camera. Both VIDICON and Charged Coupled Device (CCD) are common, with the latter being preferred due to size and weight. The camera is capable of sending analog information out one wire at speeds sufficient to display without flickering. The information is sent line by line with all the odd lines being sent and then all the even lines following. This is known as interlaced scanning [10]. Between each line a 0-V horizontal sync pulse is sent to let the image processor know a new line is coming. After a complete picture has been scanned, a longer vertical sync pulse is sent to indicate the completion of a frame.

3. Dynamics and Control

In order for a control system to be developed for the specific application, a complex study of the physical system is necessary. There are several ways of doing this. For example, the motor may be given a step input and have its output measured. This might give a fairly accurate system model. A different approach is to obtain a model by modeling the dynamics of the system itself. However, because this system is highly nonlinear, this may prove problematic.

One solution to the trouble of modeling nonlinear equations is to make use of Lagrange's method, which uses the concept of potential and kinetic energy instead of forces. Because of the amount of mathematical calculations required by this method, it is convenient to use a computer to find the solution. In Appendix A1, a MATLAB program for calculating the system dynamics is shown. By applying Lagrange's method to the system at hand, assuming the marble rolls without slipping and the angular velocity of the track has a negligible effect on the velocity of the marble, the following system of equations is obtained.

$$-B_x \dot{x} = 7/5 m_1 \ddot{x} - m_1 x \dot{\theta}^2 - m_1 g \cos(\theta) \quad (3.1)$$

$$T_m - B_\theta \dot{\theta} = 2m_1 x \ddot{x} + (m_1 x^2 + 1/2 m_2 L^2) \ddot{\theta} - m_1 g x \cdot \cos(\theta) \quad (3.2)$$

where:

- m_1 = mass of ball
- m_2 = mass of beam
- B_x = linear damping coefficient of ball
- B_θ = angular damping coefficient of beam
- T_m = motor torque
- x = linear position of ball
- $\dot{x}$ = linear velocity of ball
- $\ddot{x}$ = linear acceleration of ball
- θ = angular position of beam
- $\dot{\theta}$ = angular velocity of beam
- $\ddot{\theta}$ = angular acceleration of beam

As can be seen, these equations are both nonlinear, which makes them very difficult to use in any attempt to develop a control solution. The equations can, however, be linearized. By examining values that the state variables are most likely to take, approximations can be made which will allow for four linear state equations which are an accurate model for the nonlinear system. In order to linearize the system, the amount of accuracy is determined by how many values the state variables are allowed to take. For example, one linearization would assume $\theta \approx 0$, for the purposes of simplification since the track is not expected to turn at great angles. This would allow changes to be made to

the above equations that remove nonlinearities (for example, the $\cos(\theta)$ in (3.1) becomes a constant value of 1. Another, more complex linearization algorithm might have several values that the state variables may take on and use a lookup table for each value. Whatever system is being used, it is important to note that if the actual variables take on values too different from the assumptions, the whole linear model becomes useless.

For this specific problem, both position and angular position variables are linearized about 0. This is legitimate since, as mentioned earlier, the track should not turn at a great angle. Also, the position will never be more than .5 meter in either direction since the beam is only one meter long. A few other assumptions must be made in order to completely linearize the system. Since angular velocity is expected to be rather small, any term with $\dot{\theta}^2$ is assumed to be zero as well. By making these assumptions, a linear state matrix can be created. This matrix is shown in (3.3).

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -5B_r/7m_1 & 5g/7 & 0 \\ 0 & 0 & 0 & 1 \\ m_1g/D & 0 & 0 & -B_\theta/D \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1/D \end{bmatrix} \cdot T_m \quad (3.3)$$

where: $D = m_1 L^2 / 2$

Once a state matrix is formed, a control solution can be begun using MATLAB. The simplest way to accomplish this is by using an integral plus gain state feedback design. A model of this design is given below, in Figure 3.1

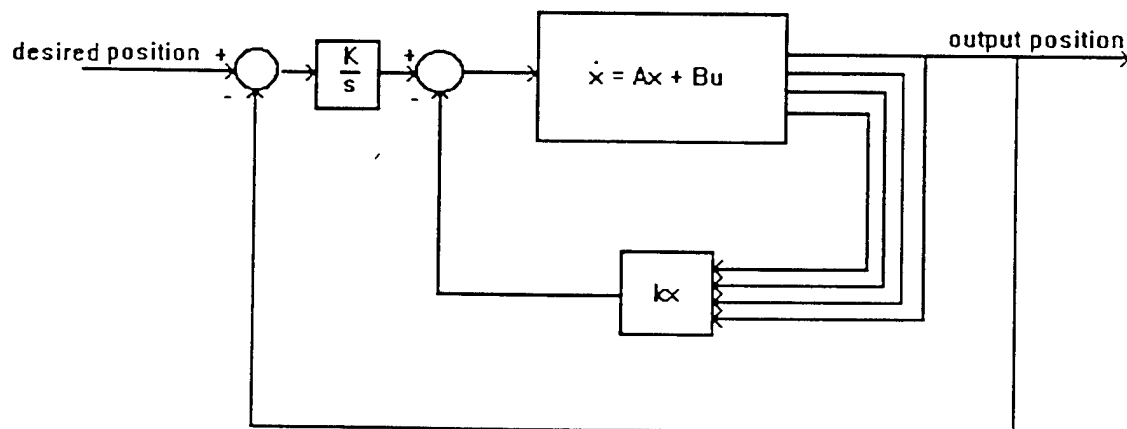


Figure 3.1. State Feedback Integral Design

The integrator will assure that the error goes to zero even if there are variations between the actual system and the modeled system. Appendix C shows a MATLAB program written to perform this function. By adjusting only one line in the program, a set

of desired poles can be chosen. Changing these poles should change the step response of the system. Since no overshoot at all is acceptable in the case when the marble is directed to move near the end of the beam (if any overshoot existed, the marble would fall off the track), poles near the origin are chosen initially. The program has the capability to allow the user to print these gains in a binary file which may be used as a source of input data to the PC running in the network. This will allow the user to change the poles using MATLAB and witness immediately the effect of the new gains on system response.

In order to test both the accuracy of the linear model as well as the effectiveness of the pole placement on step response, it is necessary to simulate the actual system. This is accomplished using Simulink, a program which runs from within MATLAB that can accept data from MATLAB. Here, the actual nonlinear system is simulated according to the equations obtained using Lagrange's method. At the same time, the linear model was simulated, and the same input was applied to both systems. The results of this are shown in Figure 3.2, where the response to both the linear and nonlinear systems are plotted on the same graph.

The approximated linear response is very similar to the expected actual response from the system. This means that the assumptions made when linearizing the system were good ones. The responses are also acceptable in terms of overshoot and settling time. This indicates that the poles selected were also a good choice. The required input torque can also be plotted using the data from Simulink and MATLAB. Figure 3.3 shows the torque of the motor in both the linear approximation and nonlinear model. Again, there is a very good relationship between the two. What remains to be seen is whether the motor is capable of exerting this much torque.

If different poles are chosen by the user, the position graphs began to differ. For example, if the poles are moved to speed the peak time up, two things happen. First, the overshoot reaches an unacceptable level (this is expected). What is not expected is the much larger difference between the modeled response and actual response. This indicates that the system was forced past the point where the linearization assumptions were valid. If the user really wanted to make use of this pole placement, the entire linear model would need to be changed. Fortunately, the desired response from Figure 3.2 remains within these parameters. These state feedback gains seem therefore a good choice for the design.

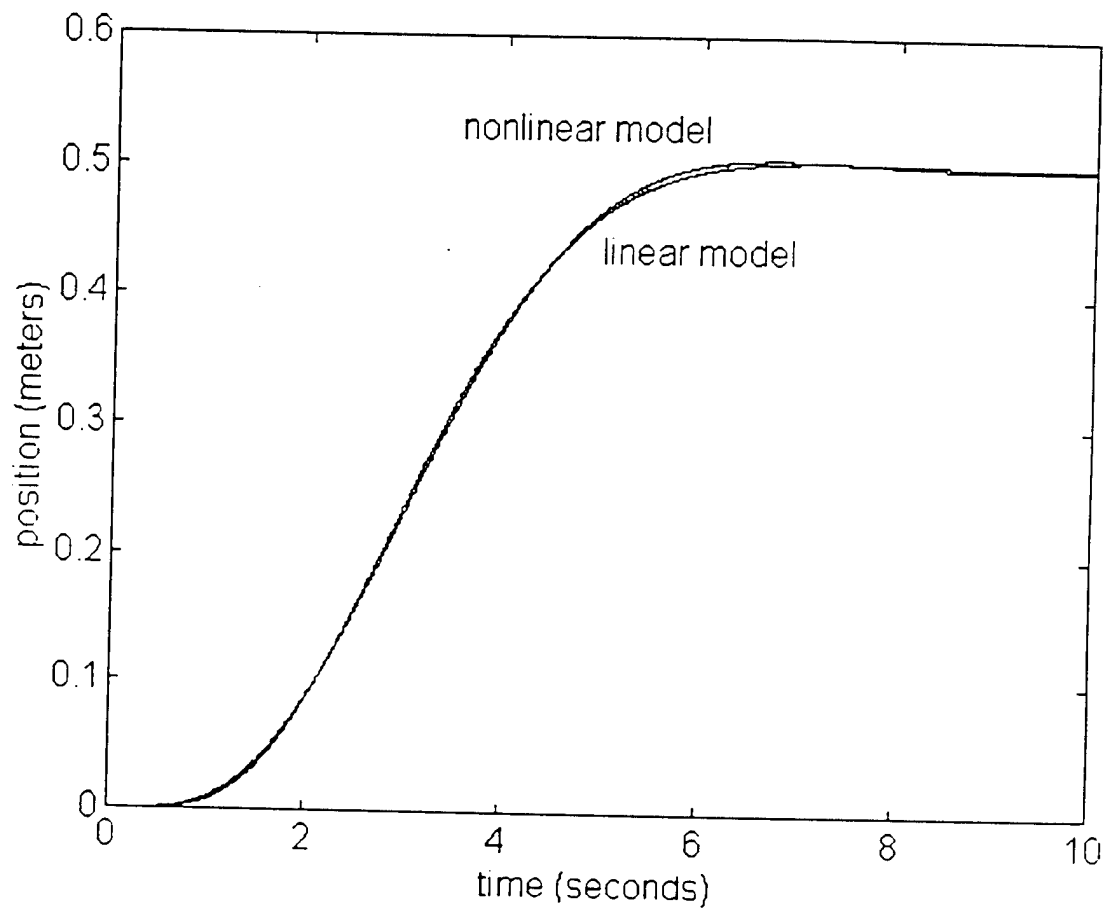


Figure 3.2. Position Response Comparison of Linear and Nonlinear Models

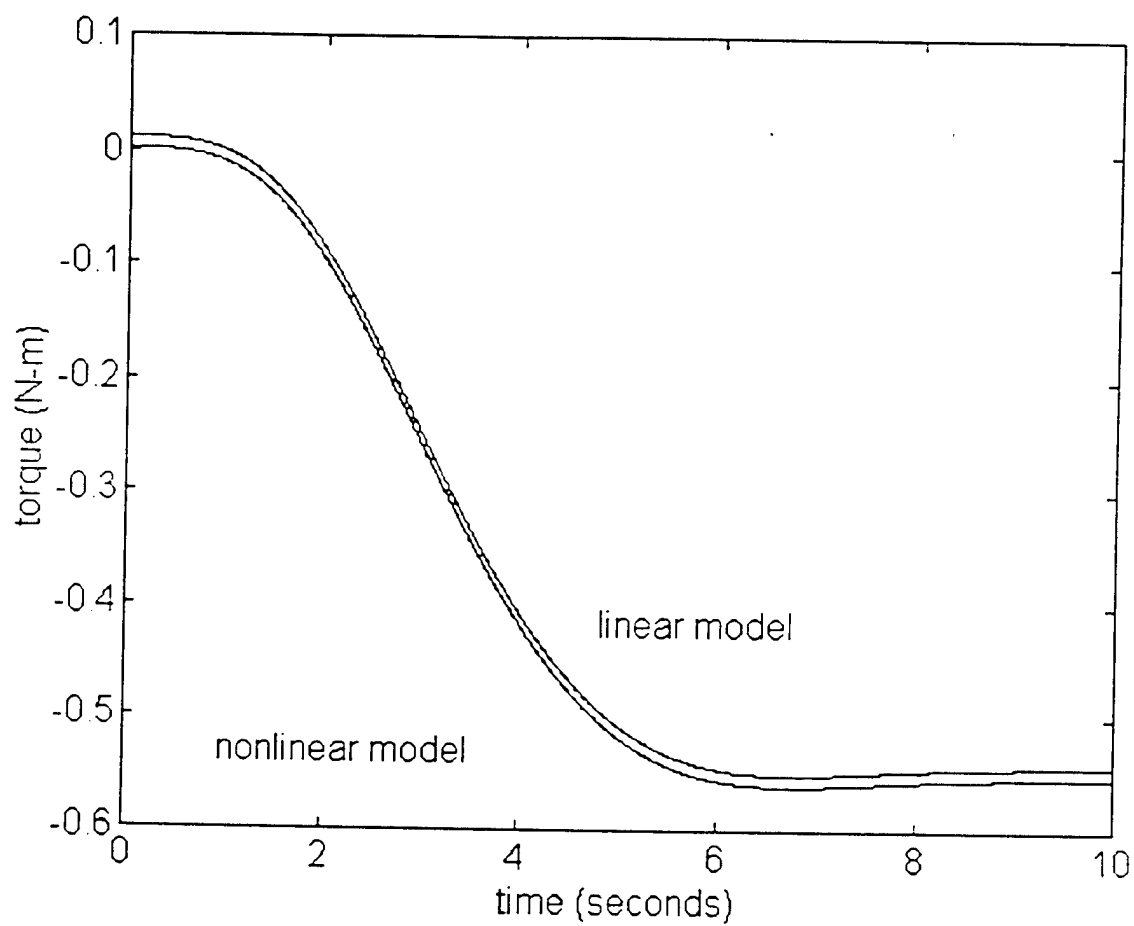


Figure 3.3 Torque Response Comparison of Linear and Nonlinear Models

4. Token Passing and Network Protocol

By far the most important aspect of the project is developing a message passing protocol that will be successful in a generalized control network. The protocol primarily must assure that each node that needs to send data will have some chance to send it. Secondly, it must allow for the most desirable transmission of data possible. Desirability varies depending on the network properties that the application requires and includes such factors as speed, efficiency, and determinacy. Finally, any protocol should consider that future additions may someday be made and maintain a flexibility to handle these additions.

With these factors in mind, the actual network must be examined. The heart of any network is the message passing protocol. As mentioned earlier, a token ring network is simply a group of microprocessors connected in series, with each microprocessor directly connected only to two others. The first message sent is a startup message and must be initiated by the user. After this, the network is constantly running itself, with messages being passed from one node to another, whether or not they contain actual data. If one node has a message to send, it waits for a token which indicates the network is free and then grabs the token and sends its message. The message is coded for the address that is meant to receive the message so that if there are any nodes between the sender and receiver, they will recognize that the message is not for them and pass it on. When a node receives a message that is addressed to it, it will read the message and perform the operation indicated. It may then send a message of its own or send a token, depending on whether it has something to say or not. This process is repeated ad infinitum. If a point is ever reached where not a single node has a message to transmit, then tokens will still be passed from one node to another continuously.

The communications format for the messages is a standard serial 9600 baud (bits per second) transfer. Each of the nodes communicates information by passing several bytes of data one bit at a time over two wires (one wire contains the information and the other is a common ground). One byte (or character) consists of 8 bits of information. Each bit of information is assigned either the value 1 or 0. Depending on the communication standard used, the bit values are assigned a voltage level. For example, in the RS-232 standard, by which most PC's communicate, a 1 is assigned the value of -12 V, and a 0 is assigned the value of +12 V. The standard used in the network of PIC processors by necessity needs to rely on TTL voltage levels. Thus, a 0 is assigned the value of 0 V, and a 1 the value of +5 V. A sample byte is shown below in Figure 4.1.

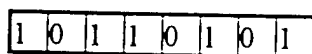


Figure 4.1. A Sample Character

This character may be represented as the binary number 10110101_2 . This number may then, for the sake of convenience, be converted to its equivalent decimal representation, in this case 181. Using this convention, then, each byte of every message is a decimal number from 0 to 255 (00000000_2 to 11111111_2). As mentioned before,

each message consists of two or more bytes of information communicated serially across one wire.

The way that each node generates these characters varies slightly because each node has different capabilities. They all generate the same characters, just in a different format. The asynchronous serial communications format used is identical to the one used by PC's with the exception of the aforementioned voltage shift. Each character is transmitted one bit at a time with the least significant bit transferred first. This complies with the RS-232 standard for personal computers. Figure 4.2 shows how the above sample character would be transmitted. One can see that following the start bit, the data transferred bit by bit is 10101101. Comparing this to the sample byte reveals that the data is reversed because the least significant digit was indeed transferred first. This is no problem because the receiving node simply shifts the data in from left to right, which shifts the first character to the rightmost bit and each successive character one bit to the left.

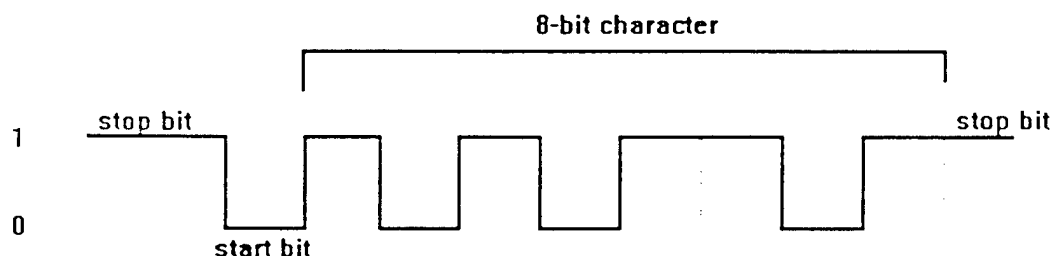


Figure 4.2. Transmission of a Character

The method by which each node generates serial communications also depends on the specific hardware. Two of the three PIC processors serving as nodes in the network are powerful PIC16C74's with serial communication built into the hardware. This means that all that is required to transmit data is simply to load a register called TXREG with the 8-bit character that is to be sent. Then, by setting a few control registers to govern the baud rate and enabling transmission, the data will be sent out at the appropriate speed, and the processor can be signaled with an interrupt when one byte is finished transmitting. The program can then proceed to the next byte in the same fashion.

The node that controls the LCD display and communicates with the PC is the less powerful PIC16C84. This processor does not possess on-board serial communications hardware. In order for it to transmit the information, it must set and clear the transmit pin itself, according to the data desired to be sent. The processor must also consider the timing involved. Depending on the speed of the crystal oscillator being used to drive the PIC (in this case 11.0592 MHz), the program must delay for the appropriate number of instruction cycles to assure that the time each bit's value is placed on the transmit line is 1/9600 second, or 0.104 millisecond. If this proper timing is not observed, nodes with standard communications interfaces will interpret the messages as garbage and ignore them. To accomplish this the PIC16C84 uses a timing loop, where a register is decremented every three instructions. When this register reaches zero, the processor sends the next bit of data.

The same timing loop must be used when the PIC16C84 is to receive a message. In this case, the processor interrupts when its receive line goes low (when it obtains the start bit). The processor then delays for half of the bit time, when it checks the start bit again just to make sure the first check was not a glitch. If it was the actual start bit, the processor then delays for one full bit time, taking it to the center of the next bit. Here, it reads the value on the receive pin and shifts this into a register. When all eight bits are received, it checks to make sure that the high stop bit was also included in the transfer and then returns to its main program until another character is reached.

The messages that are passed use a standardized format and can be divided into three sections. The first section includes the first and second bytes of the message. The first character sent is the address of the node for whom the message is sent. This allows a node to quickly tell whether a message it has just received should be looked at more thoroughly or passed on to the next node. The second byte of the message string is the address of the node that sent the message. This might be useful information in case the receiving node needs to send a return message.

The second section of the message includes the third byte and a variable number of following bytes. The third byte of the message is the command that the receiving node is to execute. For example, if one node wanted another to execute its fifth command, the third byte of the message would be 00000101_2 (a decimal 5). The reason for sending commands will be explained below. After the command portion of the message comes any data that is necessary for the receiving node to execute its given command. The sending node will know what command it is telling the receiving node to execute and will therefore be able to send the appropriate data. It may seem inefficient to transmit commands in this way, and perhaps it is. One may wonder why the actual commands cannot be sent as part of the message instead of just numbers telling the receiving node which predefined command to execute. The reason that this is not possible is due to an inherent limitation in the PIC processor. After its initial programming, the PIC is not allowed to alter any of its program memory, which is the location where the actual commands reside. The PIC may only change its data memory, the area in memory where data values are stored. When a command is received, it merely causes the program to jump down to a predetermined subroutine. The specifics of this process are explained below.

The final section of the message includes the CRC character and the end-of-message character. The CRC (cyclical redundancy check) character is a simple error checking technique that helps to eliminate corrupted messages. Before any node sends a message, it calculates the value of the CRC character based on the other characters in the message. It then sends this CRC character along with its message. When a node receives this message, it performs its own calculations to determine the value of the CRC character and compares this calculated value to the received CRC character. If the two are the same, it accepts the message as valid. If not, it discards the message as noise and terminates it. Finally, the end-of-message character is the last character to be sent in any message. It is a special character known to all the nodes. The purpose of the end-of-message character is to let the receiving node know that there are no more characters coming. This allows it to begin processing the message.

In any token ring network, nodes must continue communications even if they do not have data to send. The way this is accomplished is by using a token. The token is

simply a special character that each node recognizes upon receipt. When it receives the token, it checks to see whether it has data waiting to be sent, and either sends that data or another token. The token message consists of only one character (00000000_2) and an end-of message character. It contains no addressing characters because a token may be sent to any node from the preceding node. Because of the unique way in which CRC is calculated, the token does not need a CRC character either.

In Figure 4.3, a sample message is shown as it would be received by a node. Following is a description of how the receiving node interprets the message and carries out its instructions.

0000 0010	1) TO Address
0000 0001	2) FROM Address
0000 0011	3) Command Number
1001 0101	4) Data
0110 0001	5) Data
1010 0001	6) Data
1011 0101	7) CRC
0111 1110	8) End-of-message

Figure 4.3. A Sample Message

The first operation that a node performs when it receives a message is to determine if the message is, in fact, legitimate. As was mentioned before, this is accomplished through the use of the CRC character at the end of the message. The CRC protocol used in the network is such that all of the characters received not including the end-of-message character are added. If the result is zero, the CRC is valid; if not, it fails. The receiving node, then, is not merely looking at the CRC character. It never distinguishes this particular character from any of the others. Rather, it adds up all the characters (excluding the end-of-message character) and checks for a result of zero. The CRC character is merely there to assure that the addition equals zero. For example, looking at the sample message, a receiving node would begin adding the bytes in a binary sense and throwing out any carry bit that may occur as a result of an addition overflow. (In decimal terms, this can be thought of as adding a group of numbers together one at a time and subtracting 256 whenever the sum reaches or exceeds 256, i.e., modulo 256 arithmetic.) The addition of these particular bytes would go as follows: Each byte has the following decimal value, respectively: 2, 1, 3, 149, 97, 161, 99, 126. The end-of message character, which is not used in the CRC calculation, has the value 126. A sum of the first six bytes would yield $2+1+3+149+97+161 = 413$. Since this is greater than or equal to 256, 256 must be subtracted from the sum to give 157. When the next byte (the CRC byte) is added to the running sum, the following result is obtained: $157+99 = 256$. Again, since this is greater than or equal to 256, we must subtract 256 again to give the number zero. Since the next character is the recognizable end-of-message character, it is not added in the calculations. The receiving node then stops looking at the characters and checks the CRC sum. Finding that it is indeed zero, the node accepts the message as valid and begins processing it.

The node then looks at the first byte of the message and compares this number to its own address which is stored in its data memory. Assuming they are the same, the processing continues. If they differ, the receiving node retransmits to the next node in the sequence the exact message it received. (There is an option at startup to initialize all addresses of all nodes to zero. In this case, when node receives its first message, it will set its address to the value of the first byte of the first message it receives. It can then send a message initializing the next node in the chain.) If the addresses are the same, which we will assume for our example, the node proceeds to the next byte of the address, where it stores the address of the sender. At present, this feature is not used to its maximum potential. While a node still records the sender, it never actually uses this information. Leaving this as part of the general protocol, however, allows for easy future improvements and customized messages.

With addressing complete, the receiving node looks at the third byte of the message — the command byte. It stores this command number in a data register and performs a jump to the appropriate subroutine. Appendix A4 gives the details of this operation.

When the processor jumps to the appropriate subroutine, it may require some data (supplied in the message string) to complete its task. This data is taken from the remaining bytes in the message. In the example, it can be seen that Command 3 required 3 bytes of data. This data is taken from bytes 4, 5, and 6 of the message. As mentioned, the sending node knows what command it is telling the receiving node to execute and therefore knows which data to send in the remaining bytes of the message.

At this point the receiving node is finished executing its command. This command may or may not have required it to send a certain message to another node or even to the same node that sent it. In this case, it sends this message right away in the same format as discussed earlier. In most cases, however, a command will not require that the node send data across the network. In these situations, after a node has executed the command it was instructed to execute, it checks a flag to see whether input from the user or attached sensors requires it to send a message. This may occur when the camera detects a change in the position of the ball or when the user turns the shaft encoder, indicating a change in the desired position of the ball. If it has data to send, it jumps to the subroutine which loads the data into the appropriate message registers and then goes to its Transmit subroutine, which sends data from the message registers until the end-of message character is reached. If the computer, after having received a message addressed to it or a token, discovers that it does not have data to send, it simply sends a token to keep communications going.

This seemingly simple format still has various weaknesses that must be overcome. For example, if one node, because of sensor input, grabs the token every time that it sees it and sends a message for another token down the line, no processors between this node and the node to which it sends information will ever have the chance to transmit a message. Because this problem is more application specific, it must be dealt with depending on the application. In the application presented, the node communicating with the camera is the only node whose sensor input might direct it to send information nearly every time it encounters a token. The problem is solved in this case by placing the node that receives these messages immediately following the node sending them in the ring sequence. Another solution might have been to have the camera node send a message

only every five times it sees a token, for example. This would allow the other nodes ample time to send their traffic.

This problem seems small, but it could actually effectively lock out all other nodes from communicating across the network. One of the main reasons for choosing a token ring network in the first place was its almost perfect ability to guarantee every node a chance to send a message when it needed to. The removal of this benefit by poor placement of two nodes in the network who need to engage in a great deal of one-way communication could be disastrous.

A diagram of the network as it is implemented is shown in Figure 4.4. As can be seen, the network contains a double loop, with one particular node being assigned to both loops. The reason for this is to accommodate a personal computer into the network yet still allow each the PIC processors to communicate if the PC were removed. The extra work required by the node attached to both of the loops will be explained in detail later. The important point to note is that only one token is actually being passed around between both loops. The node which acts as a conduit between the loops is able to detect which node it receives its last message from and send the next message to the other node. A less physically accurate, but more theoretically correct model of the network is shown in Figure 4.5. Here, the network is modeled as a single loop with one node appearing two times in the loop. This represents how the actual information passes when the PC is present in the loop.

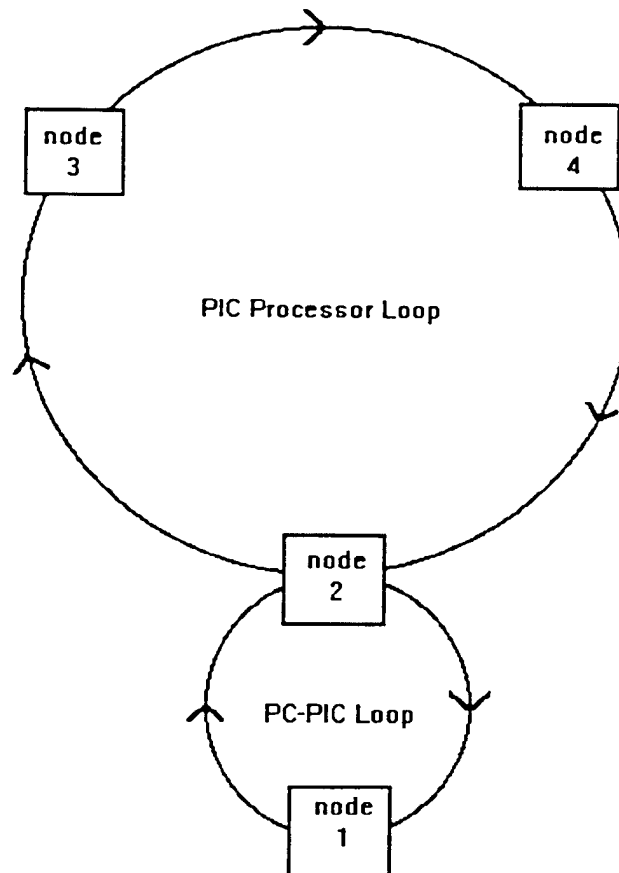


Figure 4.4. Network Connection Scheme

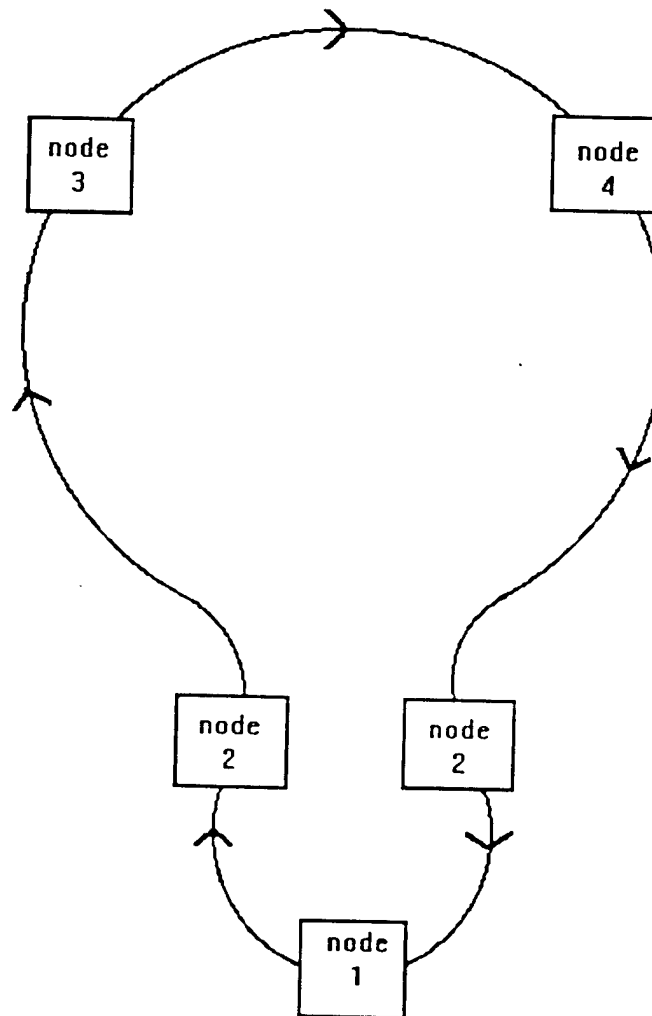


Figure 4.5. Network Token Passing Model

It is convenient to number the nodes in order to more clearly describe the specific algorithm that each follows while the network is running. The most intuitive system of labeling is simply to label the processors with the actual address that they possess upon startup. This addressing system is shown below in Table 4.1. Each node of the network is preset to follow a specific algorithm to handle message passing. That algorithm will be explained here briefly to show the broad picture of the token ring and then explained in detail in the following chapters. Node 1, the PC, starts the sequence with a keystroke from the user. This initiates a token pass to node 2, which then checks to see if it has any information to send, specifically, a new desired position that the user may have updated using the shaft encoder attached to the LCD display. If the position did change, which occurs relatively infrequently given the speed at which messages pass around the network, node 2 generates a message to node 1, informing it of the desired change of position. This message will be passed through nodes 3 and 4 and back to node 2, the sender of the

message. Node 2 will recognize that the message was relayed from node 4, and therefore transmit it to node 1, the PC. The PC will then update the change in desired position on the screen and send a message to node 2 telling it that it may now change the desired position. Now node 2 sends a message to node 4, telling the motor-controlling PIC the new desired position of the ball. No response to this message is needed. If the desired position had been changed at the computer display, node 1 would have informed node 2 to change its desired position and it would have relayed this information on to node 4.

Node	Processor	Function
1	PC	Display state variables, allow changing of desired position.
2	PIC	Handle communications with PC, Display position and desired position on LCD display.
3	PIC	Monitor camera inputs and send them to motor-controlling PIC.
4	PIC	Control motor, using digital state feedback.

Table 4.1. Nodes and their Functions

This seems rather complicated, but the entire process requires at most two loops in the ring, and this is only if the desired position changes. If the user makes no changes at the PC or LCD console, then these two nodes pass tokens to node 3, where a check is made to see if the camera detected a position or velocity change of the ball. If either of these variables have changed, then node 3 sends the new values to node 4, where the control routine uses this information. Once node 4 has received the token (or a message addressed to it), it decrements a counter. When this counter reaches zero, it transmits all of its state information to node 1, the PC, for display. The counter can be set to any number, depending on how frequently the display needs to be updated, but the process is so fast compared to what the eye can see that it does not have to pass it very often. This also cuts down on message traffic which may have precluded another node from sending data.

5. Individual Nodes

5.1 Node 2: Interfacing

As was mentioned earlier, the network is dual ring in nature to allow for the connection or removal of the PC. While the PC is connected in the network, node 2 acts as a medium through which information passes between the PC and the other nodes in the network. It also prints position and desired position information on an LCD display while accepting user inputs from a shaft encoder to change the desired position. Performing all of these functions is difficult and makes node 2 very important to the network. Adding to the difficulty is the fact that the processor at node 2 is the less powerful PIC16C84. This processor lacks the built in serial communications hardware mentioned earlier and therefore must rely on more complex software.

The actual mechanics of communicating in this way are very complicated. Instead of interrupting for a few instructions when an entire byte of data comes in like the PIC16C74, the PIC16C84 is interrupted when the start bit is detected on its receive pin. Once this occurs, the processor delays halfway into the start bit, where it checks again to verify that a start bit has arrived. This prevents errors when spurious noise causes a line to drop low. If the start bit has in fact been received, the processor delays until the midpoint of the first incoming bit and reads its value. It continues delaying to the midpoint of every bit until it reaches the stop bit. Here it stores the value of the character it obtained in a register. It then checks to see if this character is the end-of message character. If it is, it begins processing the message. If not, it continues executing its main program until another start bit causes it to interrupt again. Figure 5.1 shows how a byte of data is sampled by the node at the midpoint of every bit.

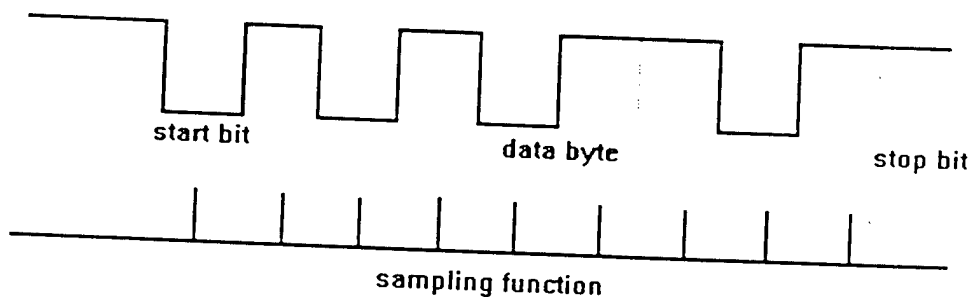


Figure 5.1. Sampling of an Incoming Transmission

Receiving data in this way has a serious drawback in that the processor's entire resources are devoted to watching this incoming signal. With the longest message approximately 10 bytes, and each byte taking approximately 1 ms to transfer, this can exhaust a lot of the processor's time. Fortunately, the actual work that the node performs besides the communications between the PC and other PIC processors does not have the need for real time data as much as some of the other nodes. It will not affect the problem

seriously if the display is not updated 10 ms from receiving the command. Other nodes might react unpredictably if one of these messages interrupted its critical timing loops. If a PIC without serial communication were used for one of these nodes, a new time-interrupt driven routine would need to be used, where the processor is able to return to the main part of its program and come back to check the data bits at their midpoints. This would not be too difficult a problem.

The processor at node 2 must also distinguish between communications coming from the PC and those coming from other nodes in the network. With only one interrupt pin, this task requires either that polling be used or that the receive signals from the two different sources actually come in on the same pin. This second method is possible using an OR gate to allow whichever of the two signals comes in to pass and two distinct transmit pins — one for each of the two connecting nodes. A diagram of this is shown in Figure 5.2. Since both sources cannot transfer at the same time, the data out of the OR gate will always be valid transmission data. The problem is detecting which of the nodes actually transmitted. If one of these two nodes generated a message then the sender can be discovered merely by reading the from address attached to all messages. If, however, the more likely situation arises where one of the two nodes is either passing along a token or a message transmitted by a different node, there is no way to tell where the message came from.

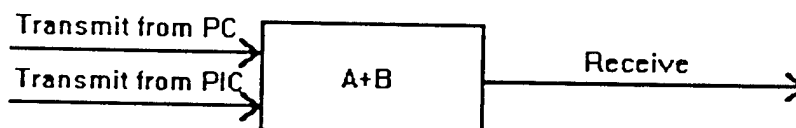


Figure 5.2. Receive Input Logic of Node 2

The solution to this problem is adding a two byte prefix onto all messages from the PC. This will allow node 2 to immediately identify which node sent it the message. It can then retransmit the message to the node that did not send it. The token character, instead of simply a zero followed by an end of message character becomes the two prefix bytes, followed by the zero, then a CRC, and finally an end-of-message character. The CRC byte is now needed because, presumably, the prefix characters are not both equal to zero, so they will skew the CRC check without a CRC character to force it to zero.

5.2 Node 3: Image Processing

An integral part to the control aspect of this project is the method of measuring the values of the states so that they can be used in a control algorithm. The measurement of the linear position and velocity of the ball is performed by a sensor suite consisting of a CCD analog camera, an LM1881 sync separator, and an LM339N quad comparator. Each of these components allows for the proper communication of useful digital data between the camera and the sensors.

The camera has only three required connections: +12 V DC, ground, and the video output signal. This output signal is analog in nature and can be viewed on a monitor. The camera sends out visual data in an interlaced format; it sends all of the odd lines followed by all of the even lines. The camera is able to transmit these signals at such high speeds that it appears that the picture is being refreshed all at once. Between each line, the camera sends a horizontal sync pulse to indicate another line is coming. This can be seen in Figure 5.3, where a typical line of data is shown. The sync pulse lasts for approximately 5 microseconds. During this time a blanking voltage of 0 V is sent to the monitor. After the sync pulse, there is approximately 5 more microseconds of useless data before the actual line data is transmitted. Each line of data lasts approximately 63.5 microseconds.

Similarly, after each half screenful (262.5 lines) is transmitted, the camera sends a vertical sync pulse to indicate that another page is coming. Shown in Figure 5.4, this pulse, which occurs much more infrequently than the horizontal sync, lasts for a much longer duration (approximately 240 microseconds). The time between vertical sync pulses is approximately 16.67 milliseconds, the time required for 262.5 horizontal sync pulses and lines of data to go by. Again, there are useless data transmitted before the first visual line.

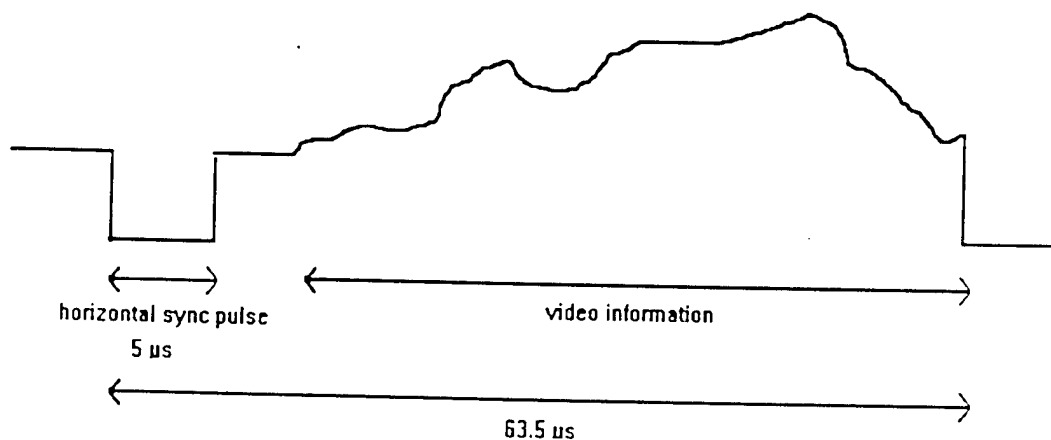


Figure 5.3. Analog Line Data

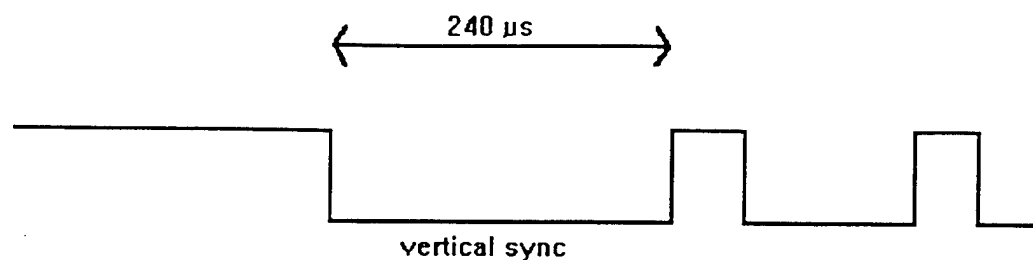


Figure 5.4. A Vertical Sync Pulse

Unfortunately, data sent in this way are not very helpful to the user of a digital computer. The solution to this problem is the LM1881 sync separator. This 8-pin chip takes as its input the analog output of the camera. It analyzes the camera signal and "strips" several useful signals from it. The vertical and horizontal sync pulses without the corresponding line data are outputs of this chip, as is a signal which changes sign each time the camera switches between transmitting even lines and transmitting odd lines. Each of these signals is extremely useful to the microprocessor because it is purely digital; the only allowable output voltages are 0 V and 5 V. One can, for instance, detect what row to monitor by watching the vertical sync pin until the pulse is received and then counting the horizontal sync pulses until the desired row is reached.

Now that the various sync pulses have been removed so that they may be used efficiently by the microprocessor, the analog data must be manipulated in a similar manner. For this particular application, the camera is searching for a white marble on a dark gray background. Assuming the computer can find the right line of data to read, its data will look something like Figure 5.5. The entire line will be dark except at the coordinates where the ball is located. What is needed is a way to tell the computer digitally when the ball is found. The solution to this part of the problem is a simple comparator whose two inputs are the analog output of the camera and a constant threshold voltage which is supplied by a voltage divider. The output of the comparator will simply be 0 V for every instance in time where the camera's output is less than the threshold voltage. When the camera's output voltage exceeds the threshold voltage (i.e., when the ball is sighted), a signal of 5 V will come from the comparator. The microprocessor can monitor this change and therefore be able to locate the ball.

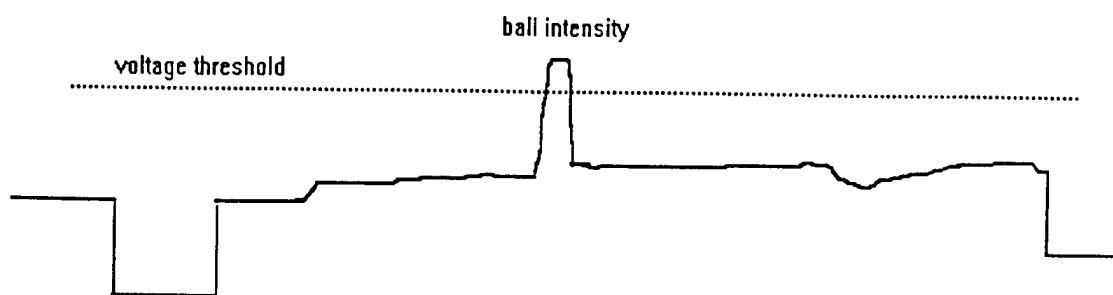


Figure 5.5. Camera Output When Ball is Found

With all the signals now conditioned to act as inputs to the PIC16C74 microprocessor, what still remains is to actually implement an algorithm to continuously monitor the position and velocity of the ball. The algorithm is interrupt-driven for precise timing and quick response. There are myriad interrupts on the PIC16C74. These include a pin that will interrupt on low-to-high or high-to-low transition, several pins which will interrupt when they change value, a timer that will interrupt when it overflows, and numerous others. Because of the many inputs to the processor, it would be difficult and time consuming to simply poll each input until one of them changes. Furthermore, a possibility exists that important data could be missed if the microprocessor was polling another input when data changed on the relevant line. To prevent this from happening, interrupts are used which allow the program to continue running until a certain condition is detected by the hardware. This causes the program to jump to a subroutine where different actions can be taken depending on the source of the interrupt.

The PIC microprocessor monitoring the camera node uses three different interrupts for the purpose of determining the velocity. The odd/even signal from the sync separator connects to one of the input pins on the PIC with an interrupt on change feature. Every time this signal transitions from low to high or high to low, an interrupt is generated. This allows the PIC to prepare for all of the even or odd lines to be transmitted. The horizontal sync pin from the sync separator is connected to the only pin on the PIC that will interrupt only on one-sided transitions (low-to-high, for example). This can be set at compile time to occur when the pin transitions from high-to-low or from low-to-high, but not both. In this particular case, since the horizontal sync is a low-going transition, the interrupt pin is set to monitor high-to-low transitions. The third interrupt used is generated by the internal timer. The PIC possesses an internal timer that increments every instruction cycle. By using an available prescaler, this timer can also increment every 2, 4, 8, 16, or 32 instructions. The timer is a one-byte register, so its maximum value is 255. When the timer increments from 255, it overflows to zero, and the timer interrupt flag is set. This interrupt is extremely useful for calculating time dependent quantities such as velocity.

The basic algorithm for the node watching the camera is outlined in the following steps:

1. **Wait for the odd/even pulse** — this tells the microprocessor that the beginning of a screen has been found, giving it a reference point for counting lines as they are transmitted. Both odd and even frames are equally useful, so no distinction need be made.
2. **Wait until desired number of horizontal sync pulses have passed** — after the odd/even pulse, a certain number of lines will be transmitted before the line on which the relevant data exists. Count and wait as these lines pass.
3. **Measure the time until the ball is found** — this action is performed without the use of an interrupt. Once the correct line is located, the output from the comparator is simply polled until a high value is obtained (this is the value where the camera's output voltage exceeds the threshold voltage). This time can easily be converted into a distance, with experimentation.
4. **Calculate velocity every 10 ms** — here the timer interrupt is used to measure an exact period of time so the velocity can be quickly calculated. Whatever the position is when this interrupt occurred last will be subtracted from the position at the current interrupt. This will tell how far the ball has moved in 10 ms, i.e., we have calculated the ball's velocity.
5. **Repeat the process** — wait for the next odd/even pulse and begin again.

The effect of the node attached to the camera in the overall token passing process is fairly simple. It must have the ability to pass any messages it receives to the next node in the sequence (as must all the nodes) as well as the ability to generate a message that sends the position and velocity data of the ball to the motor-controlling node. This node does not, however, need to worry about receiving any messages since there are no parameters that ever need changing. The algorithm for message handling is simple. If the node receives a message addressed to a node other than itself, it checks the CRC and then passes along the message to the next node in the ring. If the node receives a token, it checks to see whether the position and velocity of the ball have changed since the last update. If one or both of them have changed, the node sends a message to the motor-controlling node (which happens to be the next in sequence) updating these two states. If the node happens to receive a message addressed to it, it checks the CRC but ignores the content of the message and treats it like a token. If the CRC check ever comes up wrong, the node ignores whatever message was received and sends a token.

Part of this algorithm works so well because the node watching the camera output is positioned immediately before the motor-controlling node in the sequence. If there were nodes in between the two, a different, more complicated algorithm would need to be written so that the node assigned to the camera does not take control of the token every time it receives it. This is acceptable presently because its message is always for the node immediately following it. However, if nodes between these two wanted to send a message, they would never have the chance except in the rare case where position and velocity data did not change between passes. This placement is not an accident, nor is it

simply a convenient way to avoid a problem. In a token ring network, it makes a good deal of sense regardless of physical location to place a node that sends messages exclusively to one other node immediately preceding this other node. This is especially true if the sending node must send this data frequently, as is the case with the camera-watching node.

5.3 Node 4: Motor Control

The node which is most directly responsible for the implementation of the motor control is the PIC16C74 processor at node 4. This processor is responsible for accepting desired position inputs from nodes 1 and 2 and actual position and velocity inputs from node 3. At the same time, this processor must monitor the turn of the motor to measure the two other states—angular position and velocity of the beam. This requires a great deal of the processor's resources and therefore is best suited to the PIC16C74.

The processor measures the angular position and velocity of the motor by using an optical shaft encoder which is attached to the motor itself. This shaft encoder has two important outputs which are denoted Phase A and Phase B. These are the outputs that change as the motor is turned. A representation of the two outputs is shown below in figure 5.6.

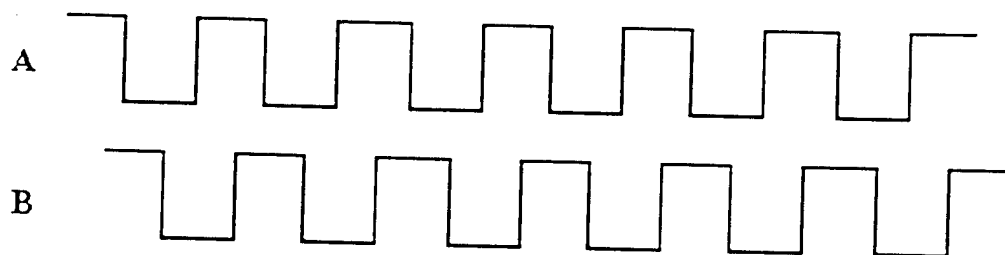


Figure 5.6. Phase Outputs of Shaft Encoder

The way these outputs work is fairly simple. As the motor turns in a clockwise direction, the two signals proceed from left to right in the figure above. If the motor instead moves counterclockwise, the signals move from right to left. The signals change at a speed proportional to the motor speed, with a resolution of 4000 phase transitions per revolution. By monitoring the changing of these signals, one can easily determine how far and in what direction the motor has moved.

The way in which the program performs this is by accepting Phase A as the input to a pin which interrupts on a low to high transition and Phase B on a noninterruptable pin. Whenever Phase A goes high, the program checks Phase B to see whether it is high or low. This determines motor direction. For example, Figures 5.7 (a) and (b) show the difference of the two signals in a clockwise and counterclockwise motion. These signals

at first look the same, but upon closer inspection, are very different. Each time Phase A is transitioning from low to high in figure 5.7 (a), Phase B is at a low point. On the other hand, each time Phase A goes from low to high in the counterclockwise direction, in Figure 5.7 (b), Phase B is high. So the processor knows after checking Phase B that the motor moved one count in either a clockwise or counterclockwise direction. It can store this in a register and update it each time an interrupt is detected.

It may be noted that this method of measuring the angular position reduces the precision by one fourth. Instead of updating the angular position register each time either phase changes value, it is updated only when phase A goes high. This reduces the resolution to 1000 counts per revolution. It is advantageous to give up this resolution because it is not really needed. At 1000 counts per revolution, the register storing the angular position increments nearly 3 times for every degree turned, easily precise enough for the current problem. If the processor were to interrupt four times as often, problems with timing and switching between functions would arise much more frequently. There would be little time to spend in the main program and excessive amounts spent in the interrupt routine.

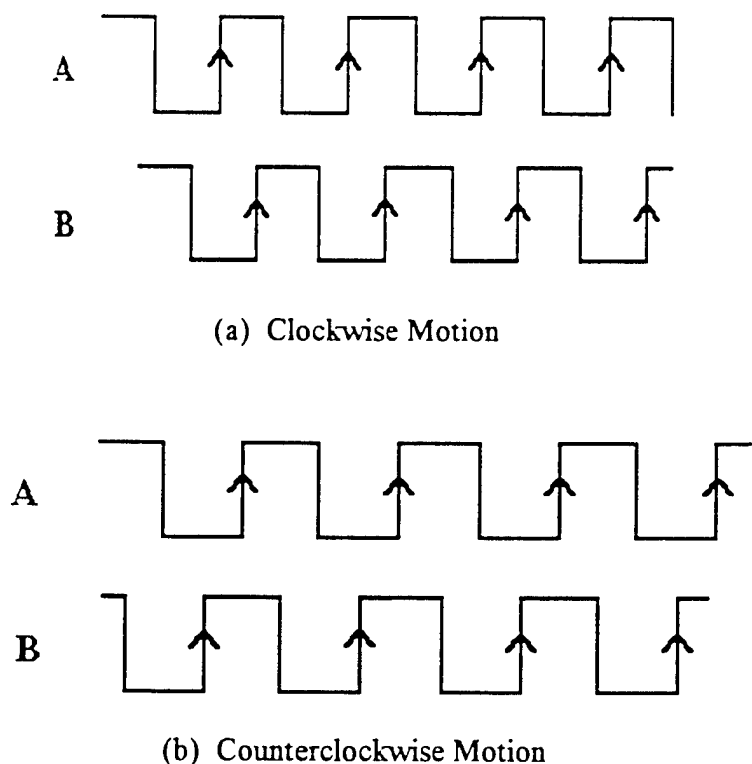


Figure 5.7. Phase Outputs of Shaft Encoder

Once the motor-controlling node has calculated the angular position and velocity of the track using the shaft encoder, and has received from the camera node the position and velocity of the ball, it is ready to begin implementing the state feedback control that

was discussed earlier. Because of the digital nature of the microprocessor, the feedback is sampled at a regular time interval. This data is used to calculate the torque that the motor must exert in order to drive the ball toward its desired position. The sampling time chosen for this particular problem was 10 milliseconds. This allows ample time for the many multiplications required to obtain the solution. Because the processor has no built in multiplication routine, repeated additions are required to multiply two numbers together. This requires time as one register is shifted while the other is added repeatedly.

As mentioned before, the proposed design is an integral state feedback design. This requires the processor to perform some operation comparable to an integral of the error data it receives. The way the processor accomplishes this is by performing a digital first order Euler approximation to the integration. A first order Euler approximation is given in (5.1) with the equivalent integral form shown in (5.2).

$$x(t + \Delta t) = x(t) + \dot{x}(t)\Delta t \quad (5.1)$$

$$\int x(t + \Delta t)dt = \int x(t)dt + x(t)\Delta t \quad (5.2)$$

The way that the processor calculates the integral is by assuming a step size of 100 milliseconds and calculating the difference between the desired position and actual position at each sample. This difference is then multiplied by the step size Δt and added to the previous value of the integral. This is an iterative process that could cause the integral to increase indefinitely if the ball were not responding properly. For this reason, a limit is placed on the maximum value of the integral. When this limit is reached, maximum torque will be applied to the motor.

After the processor calculates the required torque needed by the motor, it applies this torque by making use of the UDN-2954 motor driver. This chip, essentially an H-bridge with four transistors, allows the PIC to use pulse width modulation (PWM) to drive a constant voltage motor in both the positive and negative directions. Fortunately, this is another built in function of the PIC16C74. By setting up various control registers related to this function, the PIC can change the duty cycle of one of its output pins in a single instruction cycle. The way PWM works is simple. The periodic PWM output drives the base of a transistor which allows current from the power source to flow through the motor in the same periodic signal. When this square wave is input into the motor, it possesses a DC average value as well as components at multiples of its frequency. A DC motor being a low pass filter, these high frequency terms are cut off. So, in effect, a +6 V DC value can be created by driving a +12 V square wave at 50% duty cycle.

The other input that the motor driver requires is a Phase input. This is the input which tells the driver which way the motor is to be turned. By using these two inputs alone, the processor is able to turn the motor at any speed up to its maximum in either direction. When the program calculates the required value for the torque, it checks to see whether the number is positive or negative. This determines whether the Phase gets set or not. The program then calculates the magnitude of the required torque and places a

multiple of this value in the PWM register. In this way, the PIC is able to successfully implement state feedback control. A Phase and Enable combination is shown below in figure 5.8. The Enable pin is active low, so the extremely short duty cycle shown in the figure will actually cause the motor to turn at a high speed.

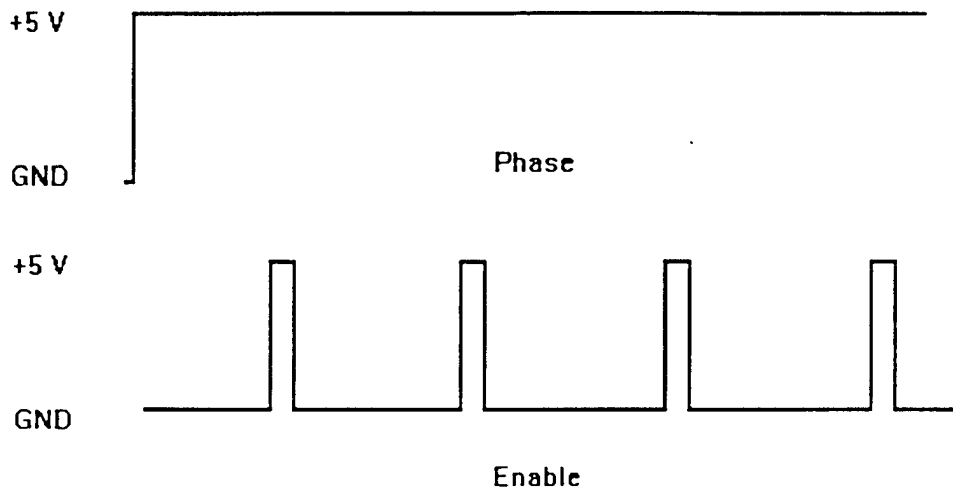


Figure 5.8. Motor Driving Signals

Most of the communications performed by node 4 are receptions. The only message that this node transmits is a periodic update of its state data every five passes through the control loop, at which point it sends to the PC the present values of position and velocity of the ball, and angular position and angular velocity of the track.

One of the abilities that was considered when designing the network was that certain values might need to be obtained at startup that were not initially hardcoded into the programming of the actual processors. It is for this reason that a serial EEPROM (electrically erasable programmable read only memory) is attached to four pins of the node governing motor control. This EEPROM can store up to 256 16-bit values in its memory, and it will retain these values when power is lost to the network. This is a useful place to store feedback gains for the processor to use when controlling the motor. It may also be useful for storing values of trigonometric functions that play a part in nonlinear problems like this one. The EEPROM can be read in a single interaction cycle. This allows the programmer to place useful data in the EEPROM ahead of time without worrying about long delays.

A series of subroutines coded into the processor at node 3 controls their operation. The connection requires only four wires: a clock pin, chip select, data input, and data output line. The process for reading from or writing to the EEPROM requires first that the chip select be turned low and that the proper command is clocked in serially across the data input line. The EEPROM interprets the command and either waits for more data over the same line in the case of a write command or places data on the data output line if a read command is invoked.

The use of the EEPROM in the control problem adds a sense of robustness. A user desiring to change the feedback gains can simply program the EEPROM at his leisure and replace the existing EEPROM in order to alter the response of the ball. This may be especially useful in a situation where no PC is present in the loop.

6. Conclusions

The research performed thus far has proven the token ring as a legitimate choice of a protocol for a control network. A network has been created and messages relating to the control problem are able to pass through it without flaw. The entire control problem has not yet been completed as there are still problems with the digital state feedback routine. Still, enough is working to make several conclusions about the performance of the token ring protocol for real-time control networks.

The response time noticed as messages are being passed among the nodes is fast enough for the sample control application. One of the worries when choosing a token ring network is that as the number of nodes is increased, the time for a message to pass around the loop increases as well. In the specific application, the messages clearly pass fast enough to allow the motor-controlling node enough time to perform its intense mathematical subroutines on updated information, instead of having to use the same sample on several times.

Even more importantly, each message was able to send its data packet in less than two passes of the token around the entire loop. The most important advantage of using a token ring network is that all messages that need to send data will be able to send that data in under a specified period of time. Under many of the other possible protocols, the average message time was much quicker, but this gain in speed was at the cost of determinacy — one could never know exactly when a node would communicate its data. If two nodes attempted to communicate at the same time, they could temporarily stall the network. This is not a problem in the token ring network.

The cyclical redundancy checking proved an effective way of stopping errors in the network. When a faulty message was intentionally sent, the first node that received it absorbed the faulty message and sent a token instead. The error detection rate approached 100 % using random flawed data. The only situations in which the CRC check failed is when the same bit in an even number of characters was altered, but this was expected and accepted as a limitation of the CRC error-checking protocol. Very little time passes while a node calculates the CRC it is required to send along with its message. The benefit of using more advanced error-checking routines that find nearly all errors is offset by the loss of network speed.

This network of single-chip microcomputers was able to successfully communicate with a modern personal computer. This allows a user to update data directly using a medium with which he is familiar. Further, myriad applications are possible now that this network can interact with a PC using the RS-232 standard.

The token ring protocol used in control networks of tiny microcomputers has proven very effective in the sample control application. In spite of its failure to complete the problem in its entirety, the scheme of message passing and error checking performed flawlessly and efficiently. Several applications now exist in which the use of this protocol may prove beneficial, and the future will continue to see the rise of such applications. One vision of the future is shared below:

“Picture your office building early one morning, some years hence. You get in by sliding your badge through an access system’s card reader. Instantly, the lights leading to your office flicker on. As you push to open the office door, lights and temperature adjust

to the settings you picked yesterday. Meanwhile, the process control network in the manufacturing area sends your computer the latest statistics.

"Abruptly, a fire alarm system warns you of smoke and fire in a section of the factory floor. It also alerts the local fire department by phone, and your heating and air conditioning die. The process control network shuts down the manufacturing line, the access control system lists who is where in the building for the fire department, and the lighting control network turns on all lights along the routes to the emergency exits. All this happens painlessly and flawlessly" [3].

This example, while not yet possible, demonstrates the power of the control network. An effective protocol for single-chip computers will only increase this power.

7. References

- [1] Reiss, Leszek. *Introduction to Local Area Networks with Microcomputer Experiments*. Prentice Hall, Inc: Englewood Cliffs, NJ, 1987.
- [2] Black, Uyless D. *Data Communications and Distributed Networks*. Prentice-Hall, Inc.: Englewood Cliffs, NJ, 1987.
- [3] Raji, Reza S. "Smart Networks for Control." *IEEE Spectrum*. June 1994.
- [4] Upender, B.P. and Koopman, P.J. "Embedded Communication Protocol Options." *Proceedings of the Embedded Systems Conference*. San Jose, CA, October 3-5, 1993.
- [5] Hummel, Robert L. *Programmer's Technical Reference: Data and Fax Communications*. Ziff Davis Press: Emeryville, CA, 1993.
- [6] Bachiochi, Jeff. "Creating the SMART-MD DC Motor Control for the I²C Bus." *Circuit Cellar Ink*, Vol. 62, September 1995.
- [7] "PIC16C71: 8-Bit CMOS EPROM Microcontroller with A/D Converter." Chandler, AZ: Microchip Technology, Inc., 1994.
- [8] Gonzalez, Rafael C. and Wintz, Paul. *Digital Image Processing*. Addison-Wesley Publishing: Reading, MA, 1987.
- [9] Vernon, David. *Machine Vision: Automated Visual Inspection and Robot Vision*. Prentice Hall: New York, 1991.
- [10] Held, Gilbert. *Token Ring Networks*. John Wiley and Sons: New York, 1994.

Appendix A1

Calculating System Equations

The following MATLAB file calculates the nonlinear equations governing the ball/beam system using Lagrange's method. It returns three different sets of equations which may be considered by the user: 1) the full dynamics, allowing slippage and orthogonal velocity effect, 2) a simplified model, ignoring the effect of the orthogonal velocity component, and 3) an even simpler model ignoring slipping as well.

```

disp('+++++')
disp('----- FULL MODEL -----')
disp('+++++')
KE1='1/2*M1*(dx^2+(x*dth)^2) + (1/2)*(2/5*M1*R^2)*(dx/R+dth)^2';
KE2='1/2*(M2*L^2/12)*dth^2';
KE=symop(KE1,'+',KE2)
PE='-M1*G*x*sin(th)'
LAG=symop(KE,'-',PE)
Fx='-Bx*dx';
T1=diff(LAG,'dx');
T2=symop(diff(T1,'x'),'*','dx');
T3=symop(diff(T1,'dx'),'*','ddx');
T4=symop(diff(T1,'th'),'*','dth');
T5=symop(diff(T1,'dth'),'*','ddth');
T6=diff(LAG,'x');
Tx=symop(T2,'+',T3,'+',T4,'+',T5,'-',T6,'-',Fx)
% collect(Tx,'ddx')
% collect(Tx,'ddth')
Fth='Tm-Bth*dth';
T1=diff(LAG,'dth');
T2=symop(diff(T1,'x'),'*','dx');
T3=symop(diff(T1,'dx'),'*','ddx');
T4=symop(diff(T1,'th'),'*','dth');
T5=symop(diff(T1,'dth'),'*','ddth');
T6=diff(LAG,'th');
Tth=symop(T2,'+',T3,'+',T4,'+',T5,'-',T6,'-',Fth)
% collect(Tth,'ddx')
% collect(Tth,'ddth')
[ddx,ddth]=solve(Tx,Tth,'ddx,ddth');

disp('ddx')
for i=1:70:length(ddx)
    disp(ddx(i:min(i+69,length(ddx))))
end

```

```

disp('ddth')
for i=1:70:length(ddth)
    disp(ddth(i:min(i+69,length(ddth))))
end

% return

% Now substitute in some numbers
ddx1=subs(ddx,'9.80621','G');
ddx1=subs(ddx1,'0.2','M1');
ddx1=subs(ddx1,'0.45','M2');
ddx1=subs(ddx1,'1.3','L');
ddx1=subs(ddx1,'0.01','R');
ddx1=subs(ddx1,'0.001','Bx');
ddx1=subs(ddx1,'0.01','Bth');

ddth1=subs(ddth,'9.80621','G');
ddth1=subs(ddth1,'0.2','M1');
ddth1=subs(ddth1,'0.45','M2');
ddth1=subs(ddth1,'1.3','L');
ddth1=subs(ddth1,'0.01','R');
ddth1=subs(ddth1,'0.001','Bx');
ddth1=subs(ddth1,'0.01','Bth');

disp('ddx1')
for i=1:70:length(ddx1)
    disp(ddx1(i:min(i+69,length(ddx1))))
end
disp('ddth1')
for i=1:70:length(ddth1)
    disp(ddth1(i:min(i+69,length(ddth1))))
end
disp('+++++')
disp('+++++      END FULL MODEL      +++++')
disp('+++++')
disp(' ')
disp('+++++')
disp('+++++      SIMPLIFIED MODEL      +++++')
disp('+++++')
KE1='1/2*M1*(dx^2+(x*dth)^2) + (1/2)*(2/5*M1*R^2)*(dx/R)^2';
KE2='1/2*(M2*L^2/12)*dth^2';
KE=symop(KE1,'+',KE2)
PE='-M1*G*x*sin(th)'
LAG=symop(KE,'-',PE)
Fx='-Bx*dx';
T1=diff(LAG,'dx');
T2=symop(diff(T1,'x'),'*','dx');
T3=symop(diff(T1,'dx'),'*','ddx');
T4=symop(diff(T1,'th'),'*','dth');
T5=symop(diff(T1,'dth'),'*','ddth');
T6=diff(LAG,'x');
Tx=symop(T2,'+',T3,'+',T4,'+',T5,'-',T6,'-',Fx)
% collect(Tx,'ddx')
% collect(Tx,'ddth')

```

```

Fth='Tm-Bth*dth';
T1=diff(LAG,'dth');
T2=symop(diff(T1,'x'),'*', 'dx');
T3=symop(diff(T1,'dx'),'*', 'ddx');
T4=symop(diff(T1,'th'),'*', 'dth');
T5=symop(diff(T1,'dth'),'*', 'ddth');
T6=diff(LAG,'th');
Tth=symop(T2,'+',T3,'+',T4,'+',T5,'-',T6,'-',Fth)
% collect(Tth,'ddx')
% collect(Tth,'ddth')
[ddx,ddth]=solve(Tx,Tth,'ddx,ddth');

disp('ddx')
for i=1:70:length(ddx)
    disp(ddx(i:min(i+69,length(ddx))))
end
disp('ddth')
for i=1:70:length(ddth)
    disp(ddth(i:min(i+69,length(ddth))))
end

% Now substitute in some numbers
ddx1=subs(ddx,'9.80621','G');
ddx1=subs(ddx1,'0.2','M1');
ddx1=subs(ddx1,'0.45','M2');
ddx1=subs(ddx1,'1.3','L');
ddx1=subs(ddx1,'0.01','R');
ddx1=subs(ddx1,'0.001','Bx');
ddx1=subs(ddx1,'0.01','Bth');

ddth1=subs(ddth,'9.80621','G');
ddth1=subs(ddth1,'0.2','M1');
ddth1=subs(ddth1,'0.45','M2');
ddth1=subs(ddth1,'1.3','L');
ddth1=subs(ddth1,'0.01','R');
ddth1=subs(ddth1,'0.001','Bx');
ddth1=subs(ddth1,'0.01','Bth');

disp('ddx1')
for i=1:70:length(ddx1)
    disp(ddx1(i:min(i+69,length(ddx1))))
end
disp('ddth1')
for i=1:70:length(ddth1)
    disp(ddth1(i:min(i+69,length(ddth1))))
end

disp('+++++')
disp('+++++  END SIMPLIFIED MODEL  +++++')
disp('+++++')

```

Appendix A2

Integrator State Feedback Design

The following MATLAB code acts on the linearized state-space matrix and calculates five gains (for the fifth order system) to place the poles in the user specified location.

```
% Linearization and Simulation of ball/beam dynamics

M1=0.113;
M2=0.455;
Bx=0.01;
Bth=0.01;
G=9.80621;
L=1.0;

x01=0; x02=.5;           %Initial and final ball position

%Use final position in forming [A,B,C,D]
x0=0;
DEN=M1*x0^2+M2*L^2/2;
Az=[0 1 0 0; 0 -5/7*Bx/M1 5/7*G 0; 0 0 0 1; M1*G/DEN 0 0 -Bth/DEN];
Bz=[0 0 0 1/DEN]';
Cz=[1 0 0 0];
Dz=0;

p=[-1+.7*j -1-.7*j -1.1+.7*j -1.1-.7*j -2]      ;desired poles

ce1=conv([1 -p(1)], [1 -p(2)]);
ce2=conv([1 -p(3)], [1 -p(4)]);
ce1=conv(ce1, ce2);
ce=conv(ce1, [1 -p(5)]);
ce=ce(2:6);

Kbar=willt(Az, Bz, Cz, ce);
K=-Kbar(5)
%Kz=place(Az, Bz, p);
Kz=Kbar(1:4);
K1=Kz(1)
K2=Kz(2)
K3=Kz(3)
K4=Kz(4)
Acl=Az-Bz*Kz;

t=0:.01:10;
[num, den]=ss2tf(Acl, Bz, Cz, Dz);
num=K*num;
den=conv(den, [1 0]);
[numcl, dencl]=cloop(num, den);
```

```

step(numcl,denc1,t);
[Ai,Bi,Ci,Di]=tf2ss(numcl,denc1);

%U=ones(size(t));
%U=x02*U;
XX0=[x01 0 0 0]';
%return;

%[YY,XX]=lsim(Acl,Bz,Cz,Dz,U,t,XX0);
%[YY,XX]=lsim(Ai,Bi,Ci,Di,U,t,XX0);

%plot(t,YY); grid; figure(gcf);

%Tm=-XX*Kz'-M1*G*x02;    %Motor torque to apply
%tt=t';

```

Appendix A3

Additive Clock Operations

The PIC processor has a program counter that increments after each instruction. The new value of the counter will be the address in program memory of the next command that it executes. On a GOTO or CALL statement (the only two jumping instructions), the processor loads the value of the address that is jumped to into the program counter before executing the jump instruction. This is how it knows where to go to execute the next instruction. It is also possible to simulate a jump by adding a number directly to the program counter. The program counter increments after it executes the instruction on its address line, so if the instruction adds a number to it, it increments this by one more. Looking at the following, the first and second lines save the third character of the message (the command byte) into a register called Command. The program counter then increments by one to go to the next step. In this next step, the PIC adds the value just stored in the command register the present value of its program counter and then increments the value of the program counter by one. For simplicity, assume that the message was the same as the example message earlier and that the program counter has the value of the line numbers in the code below. The program counter currently has the value 2. When the value of the Command register (a decimal 3) is added to it, it then contains the value 5. After it executes this addition instruction, it increments itself by one expecting to go to the next line. Actually, it contains the value 6, so it jumps to that line. When it reaches line 6, it executes the instruction there and jumps to the section of code labeled Cmd3. In this way, it is able to use the value of the third byte of the message to execute a certain command.

```

1      movf      Msg+3,W      ;move third byte of message into W
2      movwf     Command      ;move W reg into Command reg
3      addwf     PCL,F        ;add value of Command to counter
4      goto      Send_Token
5      goto      Cmd1
6      goto      Cmd2
7      goto      Cmd3
```

Appendix A4

Node 2 Source Code

The following code is the actual machine language code that is programmed into node 2 of the network, the node responsible for communicating with the PC as well as the other PIC processors.

```
;      "NODE2.SRC"
;
;
      DEVICE  PIC16C84,HS_OSC,WDT_OFF,PWRT_OFF,PROTECT_OFF

;Registers
      ORG     0Ch
TmpW      DS    1
TmpSTAT   DS    1
TmpINT    DS    1
Tmp       DS    1
Tmp1      DS    1
Tmp2      DS    2
CNT       DS    2
i         DS    1
CRC1      DS    1
ComStat   DS    1
ComReg    DS    1
ComCnt    DS    1
DelCnt    DS    1
ADDR      DS    2
PosD      DS    1
Pos       DS    1
Flg       DS    1      ; various flag bits
Disp      DS    3      ; digits to display (up to 6, BCD)
DCnt      DS    1
Msg       DS    1

;Numeric constants
F         =      1
LSB       =      0
MSB       =      7
RCF       =      ComStat.0
TXF       =      ComStat.1
CRCErr    =      ComStat.2
FERR      =      ComStat.3
OERR      =      ComStat.4
DTS       =      ComStat.5
toPC      =      ComStat.6
EOMchr    equ    '~'          ; End-of-message character
IntVEC    equ    10110000b     ; GIE=1, RTIE=1, INTE=1
RS1       equ    Flg.7        ; temporary storage for RS status
tmout     equ    Flg.6
```

```

PHAlast    equ    Flg.5    ; last value of 'PHA' of input shaft encoder
msec100    equ    Flg.4    ; 1=100 msec elapsed
ch_pos     equ    Flg.3    ; ball position changed
ch_posd    equ    Flg.2    ; desired position changed
first      equ    Flg.1    ; 1=first character received

```

```

;Pin assignments

```

```

RX          equ    PORTB.0
TX1         equ    PORTA.3
TX          equ    PORTA.2

```

```

; Signals to the 40x2 LCD display (data is sent on PORTB.6 - PORTB.3)

```

```

RS          equ    PORTB.1
RW          equ    PORTB.2
CLK         equ    PORTA.4

```

```

; Signals from input shaft encoder (inputs)

```

```

PHA         equ    PORTA.0
PHB         equ    PORTA.1

```

```

        ORG        00h                ;Code Space
        goto       START

```

```

        ORG        04h                ;ISR Space
        goto       ISR

```

```

LUT1  addwf        2, F
      retw         'POS(d) ', EOMchr

```

```

LUT2  addwf        2, F
      retw         'POS ', EOMchr

```

```

LUT3  addwf        2, F
      retw         'OK', EOMchr

```

```

ISR   movwf        TmpW                ;Save contents of W register
      swapf        STATUS,W
      movwf        TmpSTAT            ;Save contents of STATUS register

```

```

      btfsc        INTF
      call         Receive
      btfsc        RTIF
      call         Timer

```

```

ENDISR

```

```

      swapf        TmpSTAT,W
      movwf        STATUS            ;Put contents of STATUS register back
      swapf        TmpW,F
      swapf        TmpW,W            ;Put contents of W register back
      retfie

```

```

; "Timer" handles the 100 microsecond timer interrupts

```

```

Timer bcf      RTIF
      clrf     RTCC

Shaft btfscc   PHAlast
      goto    :L0
      btfscc   PHA
      goto    :L0
      bsf      ch_posd
      bsf      DTS
      btfscc   PHB
      goto    :inc
:dec  decf     PosD, F
      movlw    156
      subwf    PosD, W
      btfscc   Z
      goto    :L1
      movlw    157
      movwf    PosD
      goto    :L1
:inc  incf     PosD, F
      movlw    100
      subwf    PosD, W
      btfscc   Z
      goto    :L1
      movlw    99
      movwf    PosD
:L1   bsf      tmout
:L0   bsf      PHAlast
      btfscc   PHA
      bcf      PHAlast
;     bsf      tmout
      ;call    PosD_Chg
      return

START bsf      RP0
      clrf     INTCON      ;disable interrupts
      movlw    03h
      movwf    05h        ;TRISA
      movlw    01h        ; 81h
      movwf    06h        ;TRISB
      movlw    10001000b
      movwf    OPTION     ;Interrupt on falling edge of B0
      bcf      RP0
      movlw    2
      movwf    ADDR
      clrf     PosD
      clrf     Pos
      clrf     Flg

      bsf      ch_pos
      bsf      ch_posd
      bcf      RCF
      bsf      TX1

```

```

        bsf          TX
        ;movlw       5
        ;movwf       testcnt

LCDprep
        call         LCD_init
        call         LCD_clear

Prep0   movlw        Msg                ;move address of Msg register to FSR
        movwf        FSR
Prep1   movlw        INTVEC             ;Enable interrupts (GIE RTIE and INTE)
        movwf        INTCON
        ;goto        print_ok

MAIN    btfss        tmout
        goto         :M1
        bcf          tmout
        call         message
:M1     btfss        RCF                ;wait for receipt of byte
        goto         MAIN              ;Main part of program here

        ;goto        loop
        bcf          RCF
        ;goto        print_OK
        movf         ComReg,W
        movwf        0                ;move byte into current Msg register
        ;goto        print_OK
        sublw        EOMchr
        btfsc        Z
        goto         Process
        incf         FSR,F             ;prepare for next byte
        movf         FSR,W             ;see if addressing beyond memory
        sublw        2Fh
        btfsc        C
        goto         Prep1             ;reenable interrupts
        ;goto        print_ok
        ;goto        Prep0             ;end of buffer, start over

;Begin processing message
Process
        ;goto        print_ok
        bsf          toPC
        movlw        Msg
        movwf        FSR                ;move address of first byte (address)
to FSR
        movf         0,W               ;read address
        btfsc        Z                 ;token received
        goto         New_Info          ;check to see if data needs to be sent
        sublw        'W'
        btfsc        Z
        goto         CPU
        call         CRC
        btfss        Z

```

```

        nop
        ;goto      print_ok
        movlw      Msg
        movwf      FSR
        movf        ADDR,W
        btfsc      Z                ;check to see if node is addressable
yet
        movf        0,W              ;first byte is defining address
        movwf      ADDR
        movf        0,W
        subwf      ADDR,W
        btfss      Z
        goto       RetranstoPC      ;message not for this node
        ;call      CRC
        ;goto      CRC              ;check CRC
P0      movlw      Msg+1             ;"From" address
        movwf      FSR
        movf        0,W
        movwf      ADDR+1           ;move address of sending byte into
                                     ;ADDR+1 register

```

;Execute instruction specified by message

Command

```

        incf        FSR,F
        movf        0,W
        addwf      PCL, F
Cmd0    goto       New_Info
Cmd1    goto       Chg_PosD
Cmd2    goto       Chg_Pos
Cmd3    goto       New_Info         ;reserved for future use
Cmd4    goto       New_Info
Cmd5    goto       New_Info
Cmd6    goto       New_Info
Cmd7    goto       New_Info
Cmd8    goto       New_Info
Cmd9    goto       New_Info
CmdA    goto       New_Info
CmdB    goto       New_Info
CmdC    goto       New_Info
CmdD    goto       New_Info
CmdE    goto       New_Info
CmdF    goto       New_Info

```

```

CPU     bcf        toPC
        call       CRC
        btfss      Z
        goto       Send_Token
        ;goto      print_ok
        movlw      Msg+1
        movwf      FSR
        movf        0,W
        sublw      'P'
        btfss      Z

```

```

        goto      New_Info      ;message not valid; check for new
info/pass token
P1      incf      FSR,F
        movf      0,W          ;read address
        btfsc     Z            ;token received
        goto      New_Info      ;check to see if data needs to be sent
        movf      ADDR,W
        btfsc     Z            ;check to see if node is addressable
        movf      0,W          ;first byte is defining address
        movwf     ADDR
        movf      0,W
        subwf     ADDR,W
        btfss     Z
        ;goto     print_OK
        goto      Retrans      ;message not for this node; retransmit
        ;call     CRC          ;check CRC
        movlw     Msg+3        ;"From" address
        movwf     FSR
        movf      0,W
        movwf     ADDR+1      ;move address of sending byte into
ADDR+1 register
        goto      Command

```

;Cmd1 - update the desired position of the ball

```

Chg_PosD
        movf      Msg+5,W
        movwf     PosD
        bsf       ch_posd
        ;bsf      DTS
        ;goto     New_Info
        call      message
        goto      PosD_Chg
        ;goto     Send-Token

```

;Cmd2 - update the position of the ball

```

Chg_Pos
        movf      Msg+3,W
        movwf     Pos
        bsf       ch_pos
        goto      New_Info

```

;Check to see if node has data to be sent; if so, send that data. If not, pass token

```

New_Info
        movf      Msg,W
        btfsc     DTS          ;does new data need to be sent?
        goto      PosD_Chgr

```

;Transmit token to next node in sequence

```

Send-Token
        clrf      Msg
        movlw     EOMchr
        movwf     Msg+1
        btfss     toPC

```

```

        goto      Send
        goto      SendtoPC

;Change desired position of ball
PosD_Chgr                                ;request from PC to change PosD
        bcf       DTS
        movlw     Msg
        movwf     FSR
        movlw     01h
        ;movlw    04h                ;message goes to node 4
        movwf     0
        incf      FSR,F
        movlw     02h                ;message is from mode 2
        movwf     0
        incf      FSR,F
        movlw     02h                ;execute command 2
        movwf     0
        incf      FSR,F
        movf      PosD,W
        movwf     0
        incf      FSR,F
        movlw     EOMchr
        movwf     0
        call      CRCCalc
        btfss     toPC
        goto      Send
        goto      SendtoPC

PosD_Chg                                ;request from PC to change PosD
        bcf       DTS
        movlw     Msg
        movwf     FSR
        movlw     04h                ;message goes to node 4
        movwf     0
        incf      FSR,F
        movlw     02h                ;message is from mode 2
        movwf     0
        incf      FSR,F
        movlw     02h                ;execute command 2
        movwf     0
        incf      FSR,F
        movf      PosD,W
        movwf     0
        incf      FSR,F
        movlw     EOMchr
        movwf     0
        call      CRCCalc
        btfss     toPC
        goto      Send
        goto      SendtoPC

;Transmit data in Msg+n to Node 3 until End of Message byte is reached
;Strip WP from message
Retrans

```

```

        movlw      Msg
        movwf      FSR
:R0      incf       FSR,F
        incf       FSR,F
        movf       0,W
        decf       FSR,F
        decf       FSR,F
        movwf      0
        sublw      EOMchr
        btfsc      Z
        goto       :R1
        incf       FSR,F
        goto       :R0
:R1      call       CRCCalc
        goto       Send

```

; Transmit data in Msg+n to Node 1 until End of Message byte is reached

RetranstoPC

```

        goto       SendtoPC

```

; "Send" retransmits back command string for checking or passing on to next machine

```

Send      movlw      Msg                ; Address of "Msg" into W
        movwf      FSR                ; Load FSR with Command Byte
Send1     movf       0, W              ; Read command
        movwf      ComReg
        call       XMIT
        movf       0, W              ; Read command
        sublw      EOMchr            ; check for end-of-message byte
        btfsc      Z
        goto       Prep0
        incf       FSR, F            ; increment FSR register
        goto       Send1

```

; "SendtoPC" retransmits back command string for checking or passing on to next machine

```

SendtoPC
        movlw      Msg                ; Address of "Msg" into W
        movwf      FSR                ; Load FSR with Command Byte
StP1      movf       0, W              ; Read command
        movwf      ComReg
        call       XMITtoPC
        movf       0, W              ; Read command
        sublw      EOMchr            ; check for end-of-message byte
        btfsc      Z
        goto       Prep0
        incf       FSR, F            ; increment FSR register
        goto       StP1

```

; ----- LCD Routines -----

LCD_init

```

        bcf        RS1                ; Send instructions

```

```

        call    D41                ; Delay at least 15 ms (4x4.1 is OK)
        call    D41
        call    D41
        call    D41
        movlw   00000011b
        call    Send4              ; send lower 4 bits of [W]
        call    D41
        movlw   00000011b
        call    Send4

        movlw   00000010b          ; Function Set: Sets to 4-bit
        call    Send4
        movlw   00101000b          ; Function Set: 2-line display and
5x7 dot character font
        call    Send8
;        movlw   00001110b          ; Display ON/OFF Control:
        movlw   00001100b          ; Display ON/OFF Control:
        call    Send8
        movlw   00000110b          ; Entry Mode Set: Increment address
by 1 and shift cursor to right
        call    Send8
        movlw   00000001b          ; Clear display and place cursor on
left
        call    Send8
        bsf     RS1                ; Prepare to Send data
        return

LCD_clear
        bcf     RS1                ; Send instructions
        movlw   01h                ; Clear display
        call    Send8
        bsf     RS1                ; Send data
        return

LCD_home
        bcf     RS1                ; Send instructions
        movlw   02h                ; Return Cursor to Home Position
        call    Send8
        bsf     RS1                ; Send data
        return

LCD_white_space                ; write ' ' characters (total number
                                ; is [W])
        movwf   Tmp
        bsf     RS1                ; Send data
:L0    movlw   20h                ; ASCII for 'space'
        call    Send8
        decfsz  Tmp, F
        goto    :L0
        return

; "GotoXY" places the cursor at character specified in [W] register
GotoXY
        iorlw   80h                ; command to set DD RAM address

```

```

        bcf      RS1                ; Send instructions
        call     Send8
        bsf      RS1                ; Send data
        return

; "Send8" sends 8-bit data (in 2 4-bit nibbles out PORTB bits 6-3)
Send8 movwf     Tmp2+1              ; Temporary storage
      movwf     Tmp2                ; Temporary storage

      bcf      CLK
      rrf      Tmp2, W              ; place upper 4 bits of data in [W]
      movwf    PORTB               ; Send upper 4 bits of data
      bcf      RW
      bcf      RS
      btfsc    RS1
      bsf      RS
      bsf      CLK
      nop
      nop
      bcf      CLK                ; clock data into LCD

      movf     Tmp2+1, W
Send4: movwf    Tmp2                ; Temporary storage

      bcf      CLK
      swapf    Tmp2, F
      rrf      Tmp2, W              ; place lower 4 bits of data in [W]
      movwf    PORTB               ; Send lower 4 bits of data
      bcf      RW
      bcf      RS
      btfsc    RS1
      bsf      RS
      bsf      CLK
      nop
      nop
      bcf      CLK                ; clock data into LCD

; Make PORTB.6 input to watch BUSY flag
      bsf      RP0
      bsf      06h,6
      bcf      RP0

; Set/Reset bits for checking BUSY condition
      bcf      RS
      bsf      RW

:L3   bsf      CLK                ; clock in upper 4 bits
      nop
      nop
      btfss    PORTB, 6           ; check for busy
      goto     :L4                ; not busy
      bcf      CLK
      bsf      CLK                ; still busy, so clock in lower 4 bits
      bcf      CLK                ; and discard

```

```

        goto        :L3
:L4      bcf         CLK
        bsf         CLK          ; clock in lower 4 bits and discard
        bcf         CLK

        bcf         RW
; Return PORTB.6 to output function
        bsf         RP0
        bcf         06h,6
        bcf         RP0
        return

; Delay 4.1 milliseconds (w/ 11.0592 MHz crystal)
D41      movlw       15
        movwf       CNT
        clrf        CNT+1
:L0      decfsz      CNT+1, F
        goto        :L0
        decfsz      CNT, F
        goto        :L0
        return

; Delay long time
D410
        clrf        CNT
        clrf        CNT+1
:L0      decfsz      CNT+1, F
        goto        :L0
        decfsz      CNT, F
        goto        :L0
        return

; "Digit" sends one digit out to the display.
Digit    bsf         RS1          ; Send data
        andlw       0Fh
        icrlw       30h
        call        Send8
        return

; "message" sends the standard data to LCD display
message

        ;print      "POS(d)= (+/-).xx"
:L1a     btfss       ch_posd
        goto        :L2a
        bcf         ch_posd
        movlw       15
        call        GotoXY
        clrf        i
:L1      movf        i, W
        call        LUT1
        movwf       Tmpl          ; temporary storage
        sublw       EOMchr

```

```

        btfsc      Z
        goto       :N1
        movf       Tmp1, W
        call       Send8
        incf       i, F
        goto       :L1
:N1     btfsc      PosD,MSB
        goto       :NEG
        movlw      ' '
        movf       PosD,F
        btfss      Z
        movlw      '+'
        call       Send8
        movlw      '.'
        call       Send8
        movf       PosD,W
        movwf      CNT
        clrf       CNT+1
        call       to_dec8
        goto       :N1a
:NEG    movlw      '-'
        call       Send8
        movlw      '.'
        call       Send8
        movf       PosD,W
        sublw      0
        movwf      CNT
        clrf       CNT+1
        call       to_dec8
:N1a    swapf      Disp, W
        call       Digit
        movf       Disp, W
        call       Digit

        ;print     "POS= (+/-) .xx"
:L2a    btfss      ch_pos
        return
        bcf        ch_pos
        movlw      40h + 15
        call       GotoXY
        clrf       i
:L2     movf       i, W
        call       LUT2
        movwf      Tmp1                ; temporary storage
        sublw      EOMchr
        btfsc      Z
        goto       :N2
        movf       Tmp1, W
        call       Send8
        incf       i, F
        goto       :L2
:N2     btfsc      Pos,MSB
        goto       :NEG2

```

```

        movlw      ' '
        movf       Pos,F
        btfss      Z
        movlw      '+'
        call       Send8
        movlw      '.'
        call       Send8
        movf       Pos,W
        movwf      CNT
        clrf       CNT+1
        call       to_dec8
        goto       :N2a
:N2a    movlw      '-'
        call       Send8
        movlw      '.'
        call       Send8
        movf       Pos,W
        sublw      0
        movwf      CNT
        clrf       CNT+1
        call       to_dec8
:N2a    swapf      Disp, W
        call       Digit
        movf       Disp, W
        call       Digit
        return

```

; ----- END LCD Routines -----

; "to_dec8" is 100's digits and below for
; single-byte counts in [CNT] only (but ensure [CNT+1]=0.
; Registers: [CNT+1:CNT], [Tmp2+1:Tmp2]

```

to_dec8                                ; 100's digits and lower
        movlw      0F0h                ; clear proper display location
        andwf      Disp+1
        clrf       Disp
:E2a    movf       CNT, W                ; save copy for later restoration
        movwf      Tmp2
        movf       CNT+1, W
        movwf      Tmp2+1
        movlw      64h                ; 100's digit
        subwf      CNT, F
        btfsc      C
        goto       :E2b
        movlw      1
        subwf      CNT+1, F
        btfss      C
        goto       :E1                ; 10's digit
:E2b    incf       Disp+1, F
        goto       :E2a
:E1     movf       Tmp2, W              ; restore last value
        movwf      CNT

```

```

:E1a  movf      CNT, W           ; save copy for later restoration
      movwf    Tmp2
      movlw    10               ; 10's digit
      subwf    CNT, F
      btfss    C
      goto     :E0             ; 1's digit
      movlw    10h
      addwf    Disp, F
      goto     :E1a

:E0    movf      Tmp2, W         ; restore last value
      addwf    Disp, F
      return

```

;Serial Communications routines

;-----XMIT Subroutine-----

;This subroutine transmits one byte of data in ComReg register at 9600 bps.

;The transmit line is PORTB, bit 1

;

```

XMIT  bcf      TXF              ;byte not transmitted yet
      movf     INTCON,W         ;store entering value of INTCON
      movwf    TmpInt
      clrf     INTCON           ;temporarily disable all interrupts
      movlw    9               ;start bit + 8 data bits coming
      movwf    ComCnt
      bcf      C               ;start bit in carry
:X0    btfss    C               ;test carry data
      bcf      TX
      btfsc    C
      bsf      TX
      call     Delay           ;delay for 1/9600 s
      rrf      ComReg,F         ;shift in next bit of data
      decfsz   ComCnt
      goto     :X0
      bsf      TX              ;send stop bit
      ;call    D41
      call     Delay           ;delay for 1/9600 s
      movf     TmpInt,w         ;restore interrupt values
      movwf    INTCON
      bsf      TXF             ;indicate valid transmission of byte
      return

```

;-----XMITtoPC Subroutine-----

;This subroutine transmits one byte of data in ComReg register at 9600 bps.

;The transmit line is PORTB, bit 1

;

XMITtoPC

```

      bcf      TXF              ;byte not transmitted yet
      movf     INTCON,W         ;store entering value of INTCON
      movwf    TmpInt

```

```

        clrf      INTCON          ;temporarily disable all interrupts
        movlw     9               ;start bit + 8 data bits coming
        movwf     ComCnt
        bcf       C               ;start bit in carry
:X0     btfss     C               ;test carry data
        bcf       TX1
        btfsc     C
        bsf       TX1
        call      Delay           ;delay for 1/9600 s
        rrf       ComReg,F       ;shift in next bit of data
        decfsz    ComCnt
        goto      :X0
        bsf       TX1            ;send stop bit
        call      Delay           ;delay for 1/9600 s
        movf      TmpInt,w
        movwf     INTCON         ;restore interrupt values
        bsf       TXF            ;indicate valid transmission of byte
        return

```

; "CRCCalc" calculates CRC and adds it to message
CRCCalc

```

        movlw     Msg             ; Address of "Msg" into W
        movwf     FSR            ; Load FSR with Command Byte
        clrf      CRC1           ; CRC check holder
:L0     movf      0, W            ; Read command
        sublw     EOMchr         ; check for end-of-message byte
        btfsc     Z
        goto      :L1           ; all bytes read, check if CRC is zero
        movf      0, W            ; Read command
        addwf     CRC1, F
        incf      FSR, F
        goto      :L0           ; next command byte
:L1     movf      CRC1,W
        sublw     0
        movwf     CRC1
        movwf     0
        incf      FSR,F
        movlw     EOMchr
        movwf     0
        return

```

; "CRC" checks CRC at end of message

```

CRC     movlw     Msg             ; Address of "Msg" into W
        movwf     FSR            ; Load FSR with Command Byte
        clrf      CRC1           ; CRC check holder
:L0     movf      0, W            ; Read command
        sublw     EOMchr         ; check for end-of-message byte
        btfsc     Z
        goto      :L1           ; all bytes read, check if CRC is zero
        movf      0, W            ; Read command
        addwf     CRC1, F
        incf      FSR, F

```

```

        goto      :L0          ; next command byte
:L1      movf      CRC1, F
        ;btfss    Z
        ;goto     print_ok
        ;goto     Send_Token    ; CRC failed, ignore message
        ;goto     P0           ; CRC OK, continue processing
        return

;-----Receive subroutine-----
;Receive one byte of data at 9600 bps and store it in ComReg register.
;This routine is called by the ISR when the INTF flag is set.
;Uses PORT B, bit 0 (interrupt pin)

Receive
        bcf       RCF          ;byte not received yet
        movlw     8
        movwf     ComCnt
        call      HDelay       ;delay for half of time
        btfsc     RX          ;check again for start bit
        goto      RDone       ;exit if not proper start bit
:R0      call      Delay       ;delay until midpoint of next bit
        bsf       portb, 7
        btfss     RX
        bcf       C
        btfsc     RX
        bsf       C
        rrf       ComReg
        bcf       portb, 7
        decfsz    ComCnt, F
        goto      :R0
        call      HDelay       ;delay until start of stop bit
        bsf       RCF          ;indicate valid reception of byte
RDone    bcf       INTF
        ;decfsz    testcnt, F
        return
        bsf       ch_pos
        movf      ComReg, W
        movwf     Pos
        call      message
        goto      loop

;-----DELAY subroutine-----
;delays are set for 9600 baud using 11.05 MHz crystal

Delay    movlw     45
        movwf     DelCnt
:D0      decfsz    DelCnt
        goto      :D0
Hdelay
        movlw     45
        movwf     DelCnt
:D1      decfsz    DelCnt, F
        goto      :D1
        return

```

```

print_OK
    nop
    call    LCD_clear
    clrf    i
:L2    movf    i, W
    call    LUT3
    movwf   Tmp1          ; temporary storage
    sublw   EOMchr
    btfsc   Z
    goto    loop
    movf    Tmp1, W
    call    Send8
    incf    i, F
    goto    :L2

disp_message
    nop
    btfss   first
    return
    call    LCD_clear
    movlw   Msg
    movwf   FSR
:L2    movf    0, W
    sublw   EOMchr
    btfsc   Z
    goto    loop
    movf    0, W
    movwf   CNT
    clrf    CNT+1
    call    to_dec8
    swapf   Disp, W
    call    Digit
    movf    Disp, W
    call    Digit
    movlw   ' '
    call    Send8
    incf    FSR, F
    goto    :L2

```

Appendix A5

Node 3 Source Code

The following code is the actual machine language code that is programmed into node 3 of the network, the node responsible for monitoring camera output and communicating this information with the motor-controlling node.

```
;      "NODE3.SRC"
;
;

DEVICE  PIC16C74,HS_OSC,WDT_OFF,PWRT_OFF,PROTECT_OFF


;Registers
      ORG      20h
TmpW      DS      1
TmpSTAT   DS      1
TmpInt     DS      1
CRC1      DS      1
ComStat    DS      1
ComReg     DS      1
ComCnt     DS      1
DelCnt     DS      1
CNT        DS      2
ADDR       DS      2
Pos        DS      1
Vel        DS      1
RowCnt     DS      1
lcnt       DS      1
Msg        DS      1


;Numeric constants
F          =      1
LSB        =      0
MSB        =      7
Row        =     140
;RCF       =     ComStat.0
TXF        =     ComStat.1
CRCErr     =     ComStat.2
DTS        =     ComStat.3
SIP        =     ComStat.4
EOMchr     =     '~'           ; End-of-message character
INTVEC     =     11001000b
RTCC       =     TMR0
RTIF       =     T0IF


;Pin assignments
```

```

Horiz      equ      PORTB.0
Vert       equ      PORTB.4
CompOut    equ      PORTB.1
RCF        equ      PORTB.2

```

```

ORG        00h
goto      START

```

```

ISR        ORG        04h
movwf     TmpW
swapf     STATUS,W
movwf     TmpSTAT

```

```

btfsc     RBIF
call      VertInt
btfsc     INTF
call      HorInt
btfsc     RCIF
call      RCInt

```

```

ENDISR
swapf     TmpSTAT,W
movwf     STATUS
swapf     TmpW,F
swapf     TmpW,W
retfie

```

```

RCInt      ;btfsc     FERR                ;Test for framing error
           ;goto      Send_Token
           ;goto      loop
           ;btfsc     OERR                ;Test for overrun error
           ;goto      Send_Token
           ;goto      loop
           movf        RCREG,W            ;Recover data byte and store it in a
register
           movwf       0
           sublw       EOMchr
           btfsc       Z
           bsf         RCF
           incf        FSR,F
           movf        0,W
           ;movlw      10101010b
           ;movwf      PORTD
           return

```

```

HorInt
bcf        INTF                ;Clear interrupt flag.
decfsz    RowCnt              ;If not desired interrupt, go back
return     ;and wait for next row.
bsf        RP0
movlw     010000001b
movwf     OPTION              ;Change prescaler to WDT ;1:4 on RTCC.
bcf        RP0
movf      Pos,W              ;Age position variable

```

```

        movwf    Pos+1          ;to use to find velocity.
        clrf     RTCC
:H1     btfsc    CompOut        ;Test the comparator output
        goto     :H2
        movf     RTCC,W
        sublw    70             ;ball not found
        btfsc    C
        goto     :H1
        movf     Pos+1,W        ;use last value of position for ball
        movwf    Pos
        goto     :H3
:H2     movf     RTCC,W        ;Move the value of the timer into a
        bsf      PORTB.7        ;register for storing the position.
        movwf    Pos
        movlw    35
        subwf    Pos,F
        comf     Pos,F
        incf     Pos,F
:H3     movlw    01000100b
        bsf      RP0
        movwf    OPTION        ;Change prescaler back to 1:32
        bcf      RP0
        bcf      INTE          ;Disable horizontal interrupt
        return

```

```

VertInt
        movlw    96             ;Wait for 5089-5092 clock pulses
        movwf    RTCC
        bcf      RTIF
        bcf      PORTB.7
:V1     btfsc    RTIF
        goto     :V1
        bsf      INTE          ;Enable horizontal interrupt.
        bsf      RP0
        movlw    01000001b
        movwf    OPTION
        bcf      RP0
        bcf      RBIF          ;Clear interrupt flag.
        bcf      INTF          ;Clear flag from unwanted hor. syncs
        return                ;Return and wait for the hor. syncs

```

```

START
        clrf     INTCON
        bsf      RP0
        movlw    01000100b      ;Interrupt on rising edge of RB.0
        movwf    OPTION        ;and set prescaler to 1:32 on RTCC.
        movlw    01110011b      ;Use pins RB.0,1,4 as input
        movwf    TRISB
        movlw    00h
        clrf     TRISA
        movlw    10000000b
        movwf    TRISC
        clrf     TRISD
        clrf     TRISE

```

```

        movlw      00100000b
        movwf      PIE1
        bcf        RP0
        call       RS232
        movlw      10101010b
        movwf      PORTD
        clrf       Pos           ;Set initial position to zero
        clrf       Vel           ;Set initial velocity to zero
        movlw      Row
        movwf      RowCnt
        movlw      3
        movwf      ADDR
        bsf        CREN

Prep0
        movlw      Msg
        movwf      FSR
        movlw      INTVEC
        movwf      INTCON
        bcf        RCF
        ;goto Send_Token

MAIN    movf       RowCnt, F     ;If position was just found (i.e.--the
        nop                          ;right row was just completed
        btfsc      Z             ;and therefore RowCnt is now zero),
        nop                          ;update the velocity of the
        call       VELCALC       ;ball.
        btfss      RCF
        goto       MAIN

;Begin processing message
Process
        ;goto      loop
        movlw      Msg
        movwf      FSR           ;move address of first byte to FSR
        movf       0, W          ;read address
        btfsc      Z             ;token received
        goto       New_Info      ;check to see if data needs to be sent
        call       CRC
        btfss      Z
        goto       loop
        movlw      Msg
        movwf      FSR
        movf       ADDR, W
        btfsc      Z             ;check to see if node is addressable
        movf       0, W          ;first byte is defining address
        movwf      ADDR
        movf       0, W
        subwf      ADDR, W
        btfss      Z
        ;goto      loop
        goto       Retrans       ;message not for this node
        ;goto      CRC           ;check CRC
P0      incf       FSR, F        ;move next byte into FSR register

```

```

        movf      0,W
        movwf     ADDR+1          ;move address of sending byte into
                                   ;ADDR+1 register

;Execute instruction specified by message (none should come)
Command
        incf      FSR,F
        movf      0,W
        addwf     PCL
Cmd0 goto New_Info
Cmd1 goto New_Info
Cmd2 goto New_Info
Cmd3 goto New_Info          ;reserved for future use
Cmd4 goto New_Info
Cmd5 goto New_Info
Cmd6 goto New_Info
Cmd7 goto New_Info
Cmd8 goto New_Info
Cmd9 goto New_Info
CmdA goto New_Info
CmdB goto New_Info
CmdC goto New_Info
CmdD goto New_Info
CmdE goto New_Info
CmdF goto New_Info

;Check to see if node has data to be sent; if so, send that data.  If
not, pass token
New_Info
        movf      Msg,W
        btfsc     DTS          ;does new data need to be sent?
        goto      Update
        goto      Send_Token

;Update the linear position and velocity of the ball
Update
        bcf       DTS
        movlw     Msg
        movwf     FSR
        movlw     4
        movwf     0
        incf      FSR,F
        movlw     3
        movwf     0
        incf      FSR,F
        movlw     1
        movwf     0
        incf      FSR,F
        movf      Pos,W
        movwf     0
        incf      FSR,F
        movf      Vel,W
        movwf     0
        incf      FSR,F

```

```

        movlw      EOMchr
        movwf      0
        call       CRCCalc
        goto       Retrans

;Transmit token to next node in sequence
Send_Token
        clrf       Msg
        movlw      EOMchr
        movwf      Msg+1
        goto       Retrans

;Transmit data in Msg+n to Node 3 until End of Message byte is reached
Retrans
        goto       Send
;goto       Prep0

; "CRCCalc" calculates CRC and adds it to message
CRCCalc

        movlw      Msg                ; Address of "Msg" into W
        movwf      FSR                ; Load FSR with Command Byte
        clrf       CRC1               ; CRC check holder
:L0      movf       0, W               ; Read command
        sublw      EOMchr             ; check for end-of-message byte
        btfsc      Z
        goto       :L1               ; all bytes read, check if CRC is zero
        movf       0, W               ; Read command
        addwf      CRC1, F
        incf       FSR, F
        goto       :L0               ; next command byte
:L1      movf       CRC1, W
        sublw      0
        movwf      CRC1
        movwf      0
        incf       FSR, F
        movlw      EOMchr
        movwf      0
        return

; "CRC" checks CRC at end of message
CRC      movlw      Msg                ; Address of "Msg" into W
        movwf      FSR                ; Load FSR with Command Byte
        clrf       CRC1               ; CRC check holder
:L0      movf       0, W               ; Read command
        sublw      EOMchr             ; check for end-of-message byte
        btfsc      Z
        goto       :L1               ; all bytes read, check if CRC is zero
        movf       0, W               ; Read command
        addwf      CRC1, F
        incf       FSR, F
        goto       :L0               ; next command byte
:L1      movf       CRC1, F
        return

```

; "Send" retransmits back command string for checking or passing on to next machine

```

Send  movf      INTCON,W           ;store entering value of INTCON
      movwf     TmpInt
      clrf      INTCON
      bsf       SIP
      movlw     Msg                ; Address of "Msg" into W
      movwf     FSR                ; Load FSR with Command Byte
S0    movf      0, W               ; Read command
      movwf     ComReg
      bcf       CREN               ;Disable reception
      bsf       RP0
      bsf       TXEN
      bcf       RP0
:X0   btfss     TXIF               ;Wait until transmit buffer is clear
      goto      :X0
XMIT  movf      ComReg,W
      movwf     TXREG
      movf      0, W               ; Read command
      sublw     EOMchr             ; check for end-of-message byte
      btfsc     Z
      goto      :S1
:S0a  incf      FSR, F             ; increment FSR register
      goto      S0
:S1   bsf       RP0
:S1a  btfss     TRMT
      goto      :S1a
      bcf       TXEN
      bcf       RP0
      bsf       CREN
      movf      TmpInt,W
      movwf     INTCON            ;restore interrupt values
      goto      Prep0

RS232 bsf       RP0
      movlw     00100000b
      movwf     TXSTA             ;Setup transmit
      movlw     32
      movwf     SPBRG             ;9600 baud using 20 MHz crystal
      bcf       RP0
      movlw     10000000b
      movwf     RCSTA             ;Enable asynchronous serial port
      return

```

-----DELAY subroutine-----
;delays are set for 9600 baud using 20 MHz crystal

```

Delay movlw     82
      movwf     DelCnt

```

```

:D0  decfsz    DelCnt
      goto     :D0
Hdelay
      movlw    82
      movwf    DelCnt
:D1  decfsz    DelCnt, F
      goto     :D1
      return

```

```

loop  clrf     INTCON
      movf     Msg+1, W
      addlw    32
      movwf    PORTD
      movlw    51
      movwf    lcnt
:11   call     D410
      decfsz    lcnt
      goto     :11
      ;bsf     CREN
      goto     loop

```

; Delay 4.1 milliseconds (w/ 11.0592 MHz crystal)

```

D41  movlw     15
      movwf    CNT
      clrf     CNT+1
:L0  decfsz    CNT+1, F
      goto     :L0
      decfsz    CNT, F
      goto     :L0
      return

```

; Delay long time

```

D410
      clrf     CNT
      clrf     CNT+1
:L0  decfsz    CNT+1, F
      goto     :L0
      decfsz    CNT, F
      goto     :L0
      return

```

VELCALC

```

      movf     Pos+1, W      ;Subtract the old position from new
      subwf    Pos, W        ;and store the result in
      movwf    Vel          ;the Vel register.
      movlw    Row          ;Get the correct row back in the
      nop      ;RowCnt register in preparation for
      movwf    RowCnt       ;next vertical sync.
      movf     Pos, W
      movwf    PORTD
      bsf      DTS
      return

```

Appendix A6

Node 4 Source Code

The following code is the actual machine language code that is programmed into node 4 of the network, the node responsible for controlling the motor which turns the beam.

```
;      "NODE4.SRC"
;
;
```

```
DEVICE PIC16C74,HS_OSC,WDT_OFF,PWRT_OFF,PROTECT_OFF
```

```
      ORG    20h
EECONTROL DS    1
EEADDR    DS    1
EEDATAH   DS    1
EEDATAL   DS    1
EEOP      DS    1
EECNT     DS    1
TmpW      DS    1
TmpSTAT   DS    1
K         DS    5
Tmp       DS    3
Tmp1      DS    1
DelCnt    DS    1
ACB       DS    3
ACC       DS    1
ComStat   DS    1
ComReg    DS    1
ADDR      DS    2
PosE      DS    2
PosD      DS    1
Pos       DS    1
PosOld    DS    1
Vel       DS    1
Ang       DS    1
AngVel    DS    1
KX1       DS    2
KX2       DS    2
KX3       DS    2
KX4       DS    2
KX        DS    2
SIGN      DS    1
CRC1      DS    1
CNT       DS    2
i         DS    1
Eint      DS    2
TmpInt    DS    1
Msg       DS    1
```

```
;Numeric constants
```

```
F          =      1
LSB        =      0
MSB        =      7
RCF        =      ComStat.0
TXF        =      ComStat.1
CRCErr     =      ComStat.2
DTS        =      ComStat.3
SIP        =      ComStat.4
EOMchr     =      '~'
```

```
;Pin assignments
```

```
;Shaft encoder related pins
```

```
Aout      equ      PORTB.0      ; (input)
Bout      equ      PORTB.5      ; (input)
```

```
;Motor controller-related pins
```

```
Phase     equ      PORTB.1      ; (output)
```

```
;EEPROM-related pins
```

```
CS        equ      PORTA.0      ; (output)
CLK       equ      PORTA.1      ; (output)
DI        equ      PORTA.2      ; (output)
DO        equ      PORTA.3      ; (input)
```

```
ORG        00h
goto       START
```

```
ISR        ORG        04h
movwf     TmpW          ;Save contents of W register
swapf     STATUS,W
movwf     TmpSTAT       ;Save contents of STATUS register

btfss     RCIF          ;Test receive interrupt flag
call      RCINT
btfsc     INTF          ;Check for source of interrupt and
call      PosUpdate     ;go to the appropriate routine
btfsc     T0IF
call      VelUpdate
ENDISR

swapf     TmpSTAT,W
movwf     STATUS        ;Put contents of STATUS register
swapf     TmpW,F
swapf     TmpW,W        ;Put contents of W register back
retfie
```

```
RCINT
;btfsc     FERR          ;Test for framing error
;goto      Send_Token
;btfsc     OERR          ;Test for overrun error
;goto      Send_Token
```

```

movf      RCREG,W           ;Recover data byte and store it
movwf     0
sublw     EOMchr
btfsc     Z
bsf       RCF
incf      FSR,F
movf      0,W
return

```

```

;-----"PosUpdate"-----

```

PosUpdate

```

;This part of the interrupt service routine is called when pin Aout
;alone changes value in a positive direction. It determines which way
;the track moved and updates the position vectors associated with the
;track, Ang and Ang+1, accordingly.

```

```

bcf       INTF              ;Clear the interrupt flag
btfss     Bout              ;Determine the direction that the
goto      ADDPos            ;motor moved by polling pin B
goto      SUBPos

```

ADDPos

```

incf      Ang,F
;btfsc     Z
;incf      Ang+1,F
movf      Ang,W
movwf     PORTD
return

```

SUBPos

```

movlw     1
subwf     Ang,F
;btfss     C
;decf      Ang+1,F
movf      Ang,W
movwf     PORTD
return

```

```

;-----"VelUpdate"-----

```

VelUpdate

```

;When 50,000 instructions have passed (approximately 10 ms),
;this part of the interrupt service is called. It determines
;the velocity of the motor by subtracting from the
;most recent position of the motor the last saved position.

```

```

bcf       T0IF              ;Clear the interrupt flag
movf      PosOld,W
subwf     Ang,W             ;Subtract old position from new
movwf     AngVel
;movf      Ang+1,W
btfss     C
;decf      Ang+1,W
;movwf     AngVel+1

```

```

;movf      PosOld+1,W
;subwf     AngVel+1,F
movf       Ang,W           ;Put contents of position registers
movwf     PosOld          ;into old position registers
;movf      Ang+1,W
;movwf     PosOld+1
movf       AngVel,W
movwf     PORTD
return

```

```

;-----
;
;           Start of Main Code Space
;-----

```

```

START movlw      Msg
      movwf      FSR
      bsf        RP0           ;Bank1
      movlw      01000011b     ;Interrupt on low to high transitions
      movwf      OPTION
      clrf       INTCON
      movlw      00001000b
      movwf      TRISA
      movlw      00100001b
      movwf      TRISB
      movlw      10000000b
      movwf      TRISC
      movlw      0
      movwf      TRISD
      movlw      0
      movwf      TRISE
      movlw      00100000b
      movwf      PIE1
      bcf        RP0
      movlw      01010101b
      movwf      PORTB
      bcf        RP0
      movlw      00000100b
      movwf      T2CON        ;Prescaler = 1, TMR2 is on
      bsf        RP0
      movlw      0FFh
      movwf      PR2          ;PWM Frequency = 19.53 kHz
      bcf        RP0
      ;movlw     5
      ;movwf     DrvCnt
      movlw      00001100b
      movwf      CCP1CON      ;PWM mode, 8-bit resolution
      movlw      200
      movwf      CCPR1L
      call       RS232        ;Set up PIC16C74 for serial comm
      clrf       PosD

```

```

        clrfs      Pos
        clrfs      Vel
        clrfs      Ang
        clrfs      AngVel
        clrfs      Eint
        clrfs      Eint+1
        movlw      4
        movwf      ADDR
        bsf        CREN

Prep0    movlw      Msg
        movwf      FSR
Prep1    movlw      11110000b
        movwf      INTCON      ; (GIE, PIE, RBIE, TOIE)
        bcf        RCF
        movlw      5
        movwf      i

MAIN     movf       PosD, W
        movwf      PosE
        movf       Pos, W
        subwf      PosE, W      ; PosE = PosD - Pos
        btfss      PosE, MSB
        goto       :Inc
:Dec     movlw      1
        subwf      Eint, F
        btfss      C
        decf       Eint+1, F
        movf       Eint+1, W
        sublw      128
        btfss      Z
        goto       :Cont
        movlw      129
        movwf      Eint+1
        goto       :Cont

:Inc     incf       Eint, F
        btfsc      Z
        incf       Eint+1, F
        movf       Eint+1, W
        sublw      128
        btfss      Z
        goto       :Cont
        movlw      255
        movwf      Eint
        movlw      127
        movwf      Eint+1
        ;goto      Send_Token
        ;goto      MAIN

:Cont
;1/64*(163.84*K1)*(100*Pos)
        movf       K+1, W
        movwf      ACC
        movf       Pos, W

```

```

movwf    ACB
call     Mult
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
movf     ACB,W
movwf    KX1
movf     ACB+1,W
movwf    KX1+1

```

;16*(128*K2)*Vel

```

movf     K+2,W
movwf    ACC
movf     Vel,W
movwf    ACB
call     Mult
bcf      C
rlf      ACB,F
rlf      ACB+1,F
movf     ACB,W
movwf    KX2
movf     ACB+1,W
movwf    KX2+1

```

;1/32*(51.473*K3)*(159.15*Ang)

```

movf     K+3,W
movwf    ACC
movf     Ang,W
movwf    ACB
call     Mult
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F

```

```

bcf      C
rrf      ACB+1,F
rrf      ACB,F
bcf      C
rrf      ACB+1,F
rrf      ACB,F
movf     ACB,W
movwf    KX3
movf     ACB+1
movwf    KX3+1

```

```
;1*(160.854*K4)*(1.5915*AngVel)
```

```

movf     K+4,W
movwf    ACC
movf     AngVel,W
movwf    ACB
call     Mult
movf     ACB,W
movwf    KX4
movf     ACB+1,W
movwf    KX4+1

```

```
;Add KX1 and KX2 (result in KX2)
```

```

movf     KX1,W
addwf    KX2,F
btfsc    C
incf     KX2+1,F
movf     KX1+1,W
addwf    KX2+1,F

```

```
;Add KX3 and KX4 (result in KX4)
```

```

movf     KX3,W
addwf    KX4,F
btfsc    C
incf     KX4+1,F
movf     KX3+1,W
addwf    KX4+1,F

```

```
;Add KX1, KX2, KX3, and KX4 (result in KX4)
```

```

movf     KX2,W
addwf    KX4,F
btfsc    C
incf     KX4+1,F
movf     KX2+1,W
addwf    KX4+1,F

```

```
;Take negative of answer
```

```

comf     KX4,F
comf     KX4+1,F
incf     KX4,F
btfss    Z
incf     KX4+1,F

```

;Add answer to Eint (result in KX4)

```

movf      Eint,W
addwf     KX4
btfsc     C
incf      Eint+1,F
movf      Eint+1,W
addwf     KX4+1
;CHECK FOR BAD ADDITION

```

```

movlw     KX4+1
movwf     CCPR1L

```

```

btfss     RCF
goto      MAIN

```

;Begin processing message
Process

```

movlw     Msg
movwf     FSR                                ;move address of first byte to FSR
movf      0,W                                ;read address
btfsc     Z                                  ;token received
goto      New_Info                           ;check to see if data needs to be sent
call      CRC
btfss     Z
goto      loop
movlw     Msg
movwf     FSR
movf      ADDR,W
btfsc     Z                                  ;check to see if node is addressable
movf      0,W                                ;first byte is defining address
movwf     ADDR
movf      0,W
subwf     ADDR,W
btfss     Z
goto      Retrans                            ;message not for this node
;call     CRC                                ;check CRC
incf      FSR,F                              ;move next byte into FSR register
movf      0,W
movwf     ADDR+1                            ;move address of sending byte into
;ADDR+1 register

```

;Execute instruction specified by message
Command

```

incf      FSR,F
movf      0,W
addwf     PCL
Cmd0      return
Cmd1      goto      Chg_Pos
Cmd2      goto      Chg_PosD
Cmd3      goto      Chg_Gains
Cmd4      goto      New_Info
Cmd5      goto      New_Info

```

```

Cmd6  goto      New_Info
Cmd7  goto      New_Info
Cmd8  goto      New_Info
Cmd9  goto      New_Info
CmdA  goto      New_Info
CmdB  goto      New_Info
CmdC  goto      New_Info
CmdD  goto      New_Info
CmdE  goto      New_Info
CmdF  goto      New_Info

```

```

Chg_Pos
    movf      Msg+3,W
    movwf     Pos
    movf      Msg+4
    movwf     Vel
    goto      New_Info

```

```

Chg_PosD
    movf      Msg+3,W
    movwf     PosD
    goto      New_Info

```

```

Chg_Gains
    movf      Msg+3,W
    movwf     K+1
    movf      Msg+4
    movwf     K+2
    movf      Msg+5,W
    movwf     K+3
    movf      Msg+6
    movwf     K+4
    movf      Msg+7,W
    movwf     K
    goto      New_Info

```

;Check to see if node has data to be sent; if so, send that data. If not, pass token

```

New_Info
    movf      Msg,W
    btfsc     DTS                ;does new data need to be sent?
    goto      Update
    goto      Send_Token

```

```

Update
    nop
    goto      Send_Token

```

;Transmit token to next node in sequence

```

Send_Token
    clrf      Msg
    movlw     EOMchr
    movwf     Msg+1
    goto      Retrans

```

; Transmit data in Msg+n to Node 3 until End of Message byte is reached
Retrans

goto Send

; "CRCCalc" calculates CRC and adds it to message
CRCCalc

```

    movlw    Msg                ; Address of "Msg" into W
    movwf    FSR                ; Load FSR with Command Byte
    clrf     CRC1               ; CRC check holder
:L0  movf     0, W              ; Read command
     sublw    EOMchr            ; check for end-of-message byte
     btfsc    Z
     goto     :L1               ; all bytes read, check if CRC is zero
     movf     0, W              ; Read command
     addwf    CRC1, F
     incf     FSR, F
     goto     :L0               ; next command byte
:L1  movf     CRC1, W
     sublw    0
     movwf    CRC1
     movwf    0
     incf     FSR, F
     movlw    EOMchr
     movwf    0
     return

```

; "CRC" checks CRC at end of message

```

CRC  movlw    Msg                ; Address of "Msg" into W
     movwf    FSR                ; Load FSR with Command Byte
     clrf     CRC1               ; CRC check holder
:L0  movf     0, W              ; Read command
     sublw    EOMchr            ; check for end-of-message byte
     btfsc    Z
     goto     :L1               ; all bytes read, check if CRC is zero
     movf     0, W              ; Read command
     addwf    CRC1, F
     incf     FSR, F
     goto     :L0               ; next command byte
:L1  movf     CRC1, F
     return

```

; CRC OK, continue processing

; "Send" retransmits back command string for checking or passing on to next machine

```

Send  movf     INTCON, W          ; store entering value of INTCON
     movwf    TmpInt
     clrf     INTCON

```

```

        bsf      SIP
        movlw    Msg                ; Address of "Msg" into W
        movwf    FSR                ; Load FSR with Command Byte
S0      movf     0, W                ; Read command
        movwf    ComReg
        bcf      CREN                ; Disable reception
        bsf      RP0
        bsf      TXEN
        bcf      RP0
:X0     btfs     TXIF
        goto     :X0
XMIT    movf     ComReg, W
        movwf    TXREG
        movf     0, W                ; Read command
        sublw    EOMchr             ; check for end-of-message byte
        btfs     Z
        goto     :S1
:S0a    incf     FSR, F              ; increment FSR register
        goto     S0
:S1     bsf      RP0
:S1a    btfs     TRMT
        goto     :S1a
        ;bsf     RP0
        bcf      TXEN
        bcf      RP0
        bsf      CREN
        movf     TmpInt, W
        movwf    INTCON              ; restore interrupt values
        goto     Prep0

RS232   bsf      RP0
        movlw    00100000b
        movwf    TXSTA                ; Setup transmit
        movlw    32
        movwf    SPBRG                ; 9600 baud using 20 MHz crystal
        bcf      RP0
        movlw    10000000b
        movwf    RCSTA                ; Enable asynchronous serial port
        return

```

```

;-----DELAY subroutine-----
;delays are set for 9600 baud using 20 MHz crystal

```

```

Delay   movlw    82
        movwf    DelCnt
:D0     decfsz    DelCnt
        goto     :D0
Hdelay
        movlw    82
        movwf    DelCnt
:D1     decfsz    DelCnt, F
        goto     :D1

```

return

; Signed Multiplication routine: [ACB+1 ACB] = ACB * ACC

```

Mult  clrf      SIGN
      clrf      TMP
      clrf      TMP+1
      clrf      ACB+1
      movlw     0FFh
      btfsc     ACB,MSB
      movwf     ACB+1
      movf      ACC,W
      movwf     TMP1
      call      :NEG
:M1   clrw
      btfsc     ACC,0
      call      :M2
      bcf       C
      rrf       ACC,F
      bcf       C
      rlf       ACB,F
      rlf       ACB+1,F
      movf      ACC,F
      btfss     Z
      goto      :M1
      movf      TMP1,W
      movwf     ACC
      movf      TMP+1,W
      movwf     ACB+1
      movf      TMP,W
      movwf     ACB
      btfsc     SIGN,0
      call      :NEG0
      return
:M2   movf      ACB,W
      addwf     TMP,F
      btfss     C
      incf      TMP+1,F
      movf      ACB+1,W
      addwf     TMP+1,F
      return
:NEG  btfsc     ACC,7
      goto      :NEG1
:NEGa btfss     ACB+1,7
      return
:NEG0 incf      SIGN,F
      comf      ACB,1
      comf      ACB+1,F
      incf      ACB,F
      btfsc     Z
      incf      ACB+1,F
      return
:NEG1 incf      SIGN,F
      comf      ACC,F

```

```
        incf      ACC,F
        goto      :NEGa

; Delay 4.1 milliseconds (w/ 11.0592 MHz crystal)
D41     movlw      15
        movwf     CNT
        clrf      CNT+1
:L0     decfsz     CNT+1, F
        goto      :L0
        decfsz     CNT, F
        goto      :L0
        return

; Delay long time
D410    clrf      CNT
        clrf      CNT+1
:L0     decfsz     CNT+1, F
        goto      :L0
        decfsz     CNT, F
        goto      :L0
        return
```