

DEVELOPMENT OF TELE-SURGICAL ROBOT



PROJECT ADVISOR:

DR. OSMAN HASAN

PROJECT CO ADVISOR:

MR. SHAMYL BIN MANSOOR

GROUP MEMBERS:

SARMAD MUNIR 2007-NUST-BEE-292

SHEHRYAR TARIQ 2007-NUST-BEE-293

SYED KAMRAN HAIDER 2007-NUST-BEE-297



NATIONAL UNIVERSITY of
SCIENCES & Technology
Pakistan



CERTIFICATION

The following project **“Development of Tele-Surgical Robot”** for Final year project, by the students of Bachelors of Electronics Engineering (BEE-4), Of School of Electrical Engineering and Computer Sciences (SEECS), NUST, namely

- **Sarmad Munir**
- **Shehryar Tariq**
- **Syed Kamran Haider**

has been approved for the undertaking & communication.

.....
HoD Department of Electrical Engineering

Mr. Habeel Ahmad

.....
Advisor

Dr. Osman Hasan

.....
Co-Advisor

Mr. Shamyl Bin Mansoor



Dedication

This Project is dedicated to our Parents, teachers and our colleagues who really provided us their full support.

Acknowledgement

Our principal debt must be to Dr. Osman Hasan and Mr. Shamyf Bin Mansoor who let us do this project and helped us throughout the project.. They are really an authority on this subject. Without their esteemed guidance regarding issues in question, Tele-Surgery was all Greek to us. Our thanks are also due to Mr. Abdul Afram, whose guidance was like blessing for us. Dr. Zawar Hussain also spared some precious time for us and guided us in important issues. Finally we thank the Smart Lab members especially Mr. Mohsin who provided us ministerial support.

Abstract

This report throws some light on the first ever prototype of Tele-Surgical Robot developed at SEECS NUST. Tele-Surgical Robotics is a new concept and a lot of research is being done around the world in this field. Main domains of Tele-surgical robotics in which work can be done are: Embedded Systems, Control Systems, Network Communications, and Programming etc.

Tele-Surgery can be used to operate patients who are present in remote area. A robotic arm is fixed at patient site. Surgeon can control that robot sitting in his room with the help of joystick. Surgeon can also see the views of Patient side with the help of camera. Control Signals are sent to Patient PC via internet connection. Micro-controllers connected to patient PC control the movement of the Robot.

Peg Transfer Exercise is the basic exercise for the training of surgeons. This exercise was performed with the developed system. Initially there were few hindrances in the peg transfer exercise. It was not easy at all to perform but with certain improvements, peg transfer exercise was successfully executed.

Table of Contents

| | | |
|------------------|---|----------|
| | ABSTRACT |6 |
| Chapter 1 | INTRODUCTION | 9 |
| Chapter 2 | REVIEW OF LITERATURE | 13 |
| | • UCB/UCSF Tele-surgical Workstation |14 |
| | • Raven Tele-Surgical Robot |17 |
| | • ZEUS Tele-Surgical Robot |19 |
| | • Da Vinci robot | 21 |
| | • Network Communication in Tele-Surgery |22 |
| | • Communication Between Patient Side Computer And Robotic Arm |32 |
| | • Graphical User Interface |36 |
| | • Input to Surgeon Computer |37 |
| Chapter 3 | METHODOLOGY |40 |
| | • Network Communication |41 |
| | • Data Rate and RTT |45 |
| | • Serial Communication |46 |
| | • GLUI Implementation |49 |
| | • Joystick Interfacing |50 |
| | • Threads |52 |
| | • Video Streaming |53 |
| | • Embedded and Control Systems |56 |
| | • PID Basics |56 |
| | • PID implementation for DC motor speed control |63 |
| | • Tuning of the Controller |68 |
| | • Multi Processor Communication |69 |
| | • Implementation on Tele-Surgical Robot |75 |
| | • Implementation on Position Feedback |78 |
| | • Layouts/Circuits |79 |
| Chapter 4 | RESULTS |88 |
| Chapter 5 | DISCUSSON |99 |
| | • Peg Transfer Exercise |102 |
| Chapter 6 | CONCLUSION |104 |
| Chapter 7 | RECOMMENDATIONS |107 |

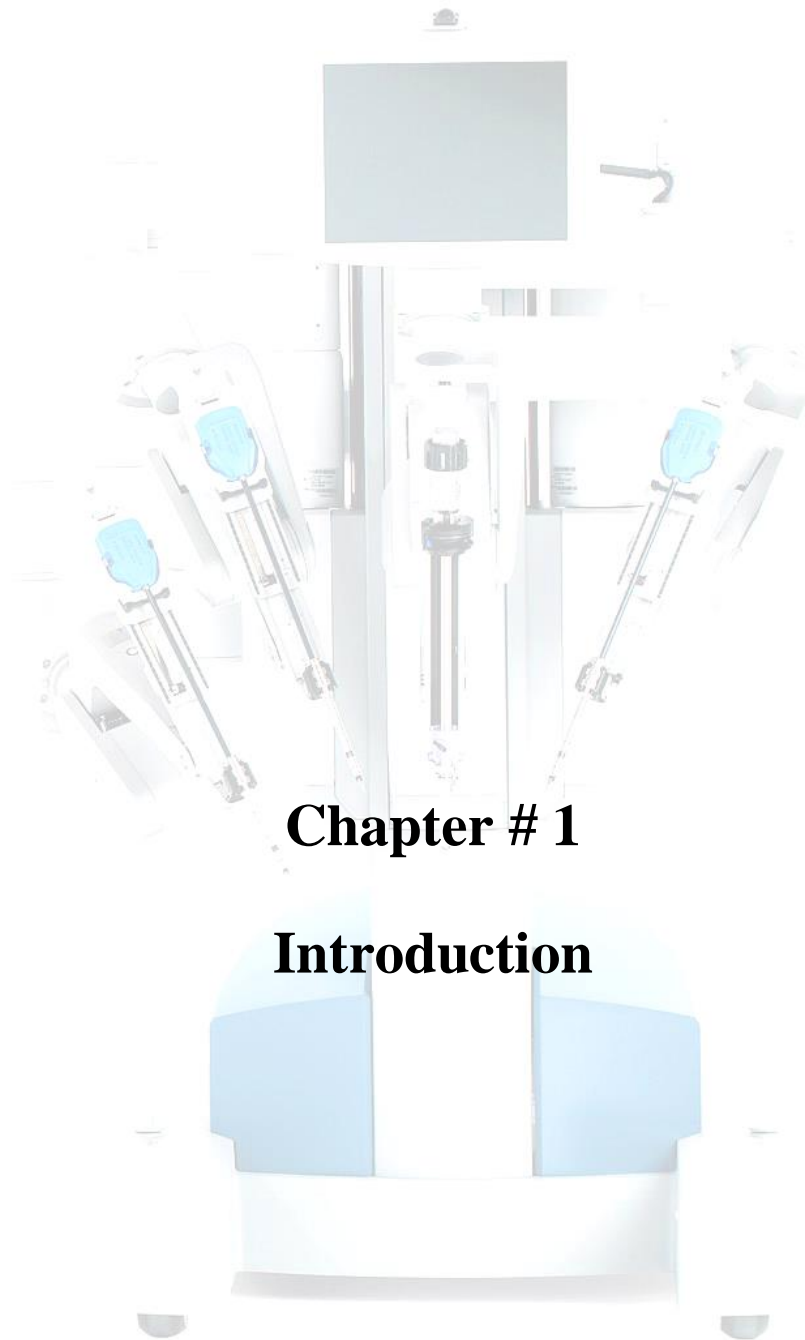
| | | |
|------------------|-------------------|----------|
| Chapter 8 | APPENDICES |109 |
| | REFERENCES |148 |

List of Tables

| |
|---|
| Table 1 Comparison between UDP and TCP |
| Table 2: Effects of increasing the parameters |
| Table 3: Sample PID Data |

List of Figures

| |
|--|
| Figure 1 Summary of Tele-Surgical Experiments |
| Figure 2 Mean Network Latency and Significance of Each |
| Figure 3 Zeus Tele-Surgical System |
| Figure 4 Graphic of Operation Lindbergh |
| Figure 5-a Illustration of NeuRobot for telesurgery |
| Figure 5: PID Block Diagram |
| Figure 6: Effect of varying K_p |
| Figure 7: Effect of varying K_i |
| Figure 8: Effect of varying K_d |
| Figure 9: H-Bridge Circuit Diagram |
| Figure 10: DC Motor PID Block Diagram |
| Figure 11: Timing Diagram |
| Figure 12: Mechanical Design |
| Figure 13: Response of motor with $K_d=0$; $K_i=0.02$ |
| Figure 14: Response of motor with $K_d=0$; $K_i=0.03$ |
| Figure 15: Response of motor with $K_d=0$; $K_i=0.04$ |
| Figure 16: Response of motor with $K_d=2$; $K_i=0$ |
| Figure 17: Response of motor with $K_d=3$; $K_i=0$ |
| Figure 18: Response of motor with $K_d=8$; $K_i=0$ |
| Figure 19: Response of motor with $K_d=8$; $K_i=0.01$ |
| Figure 20: Response of motor with $K_d=9$; $K_i=0.01$ |
| Figure 21: Best Response |



Chapter # 1

Introduction

Tele is a Greek word which means “Far OFF”. Tele-Surgery is the Surgery in which Patient and Surgeon are not present physically in the same location. Therefore, tele-surgery is also called remote surgery, is performed by a surgeon at a site removed from the patient. Surgical tasks are directly performed by a robotic system controlled by the surgeon at the remote site.

Tele-surgery became a possibility with the advent of laparoscopic surgery in the late 1980s. Laparoscopy (also called minimally invasive surgery) is a surgical procedure in which a laparoscope (a thin lighted tube) and other instruments are inserted into the abdomen through small incisions. The internal operating field may then be visualized on a video monitor connected to the scope. In certain cases, the technique may be used in place of more invasive surgical procedures that require more extensive incisions and longer recovery times.



Why Tele Surgical Robots are required

Question arises what is the need of tele surgical robots when Surgeons are there to operate. Answer to this question is given by one of the surveys about Robotic carried out recently. The survey gives an idea what this robot can do and what are its benefits.

Benefits:

Based on the latest medical studies, Benefits of Tele-Surgical robots are:

- With the help of tele-surgical robot blood loss is significantly decreased. It also reduces a number of other complications as well which are encountered in Surgery.
- It reduces the operating time. Many patients are afraid of long anesthesia, therefore by reducing operating time it lessens the anesthesia time as well.
- It does not tear off every tissue which comes in its way; it is more precise in operations, so it reduces post-operative hospitalization stay.
- Patient undergoing robotic surgery can resume his/her normal life in 24 to 48 hours.
- Risk of infections is less than normal surgeries.
- It can perform complicated surgeries with ease and with better results. It can has more DOFs than a human hand thus yielding better results.
- It can perform many operations which were considered impossible with surgeon's hands.
- Such robots can be used to train new surgeons.
- If handled properly it can also be used to operate injured soldiers near battlefield.



Domains of Tele-Surgical Robot

The major domains of this project are as follows:

a) Embedded Systems:

Processing all the inputs and generating corresponding output control signals at the patient end.

b) Control Systems:

Control Systems will be used in controlling the movement of robotic arm according to the control signals generated by the operator.

c) Communication:

Communication part consists of communication between the two sides, the patient and the surgeon, via internet and interfacing of the input device and robot controllers with the two PCs.

Formation of Tele-Surgical Robot

There are three fundamental parts of Tele-Surgical Robot in which it can be divided.

a) Patient Side

b) Network

c) Surgeon Side

Patient Side

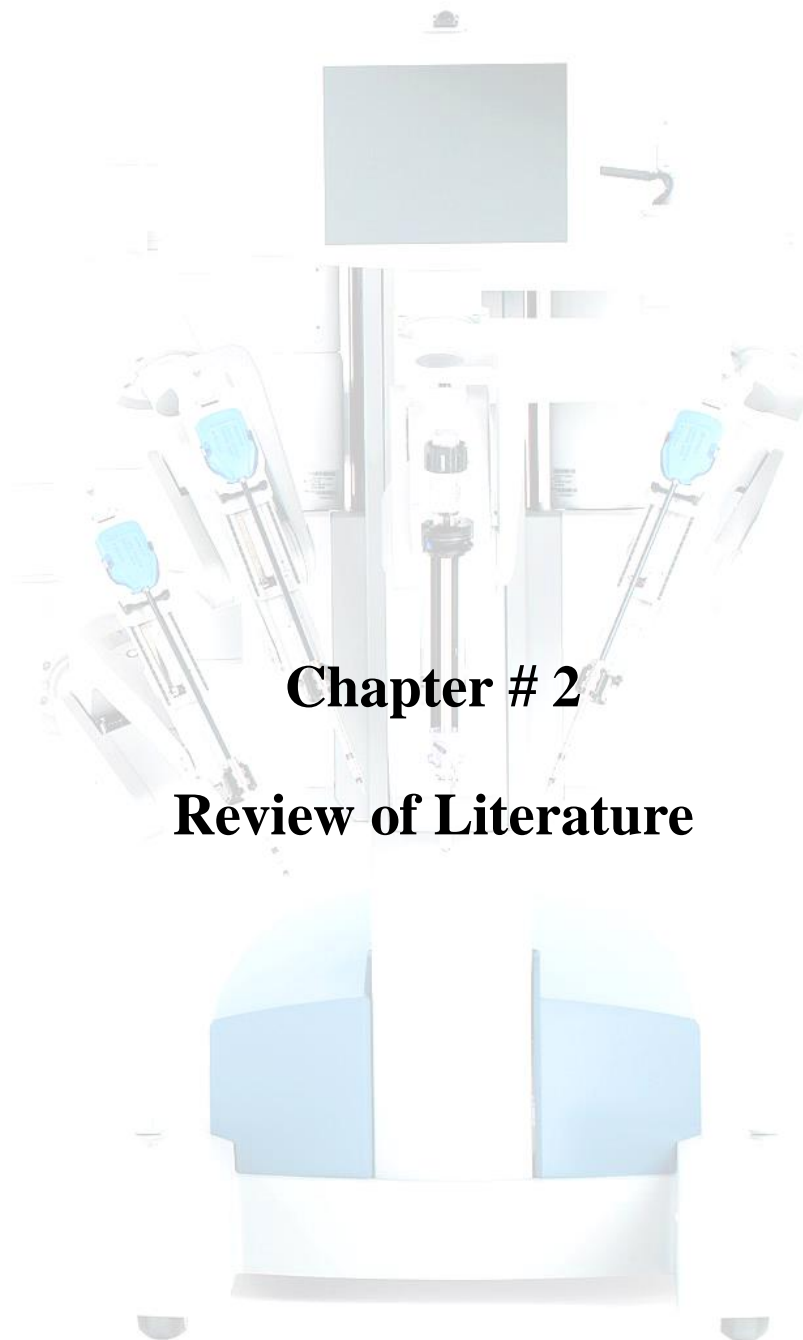
On this side we have robotic arm whose motion is controlled by motors present on the side on the basis of signals coming from surgeon side.

Network

Network is used to transmit signals for one side to other as quick as possible and reliability.

Surgeon Side

This side sends signals to patient side. It controls movement of the robot.



Chapter # 2

Review of Literature

TELE-SURGICAL ROBOTS: WHOLE SYSTEM

Many different types of tele-surgical has been developed. We went through many of them. Summaries from Mechanical Perspective of the following systems are mentioned in the chapter.

- a) UCBUCSF Laparoscopic Telesurgical Workstation
- b) Raven Tele-Surgical Robot
- c) Zeus Tele-Surgical Robot
- d) Da Vinci

Second Generation UCB/UCSF Laparoscopic Tele-surgical Workstation

Issues in Robotic Surgery:

- Dexterity or one's hand-skills are significantly reduced due to less number of DOF (i.e. 4)
- There is no tactile sense while using a robot, at which the surgeons highly depend to locate arteries etc.
- Reduced precision and difficulty in operation due to a totally different interface.

Research Areas in development of Tele-Surgical systems:

- Design of small sized yet significantly powerful manipulators. The size should be around 10 mm where as the force capability should be several Newtons.

- Achieving highly accurate Tele-operation.
- Providing a 6 DOF (preferred) force feedback at surgeon end.
- Providing tactile sensing and display.

Target Tasks in design of UCB/UCSF Robotic Tele-Surgical Workstation:

- Suturing or stitching of wounds
- Knot tying

These tasks are difficult to perform with existing laparoscopic tools mainly due to the lack of ability of orienting the tool in right position due to less DOF. Moreover the difficulties in operating the robot due to no feedback are also considerable.

System requirements:

- 2 DOF wrist with smallest possible diameter and an appropriate length.
- Fulfill the force & torque requirements for the targeted tasks.
- Provide sufficient ranges of rotation at different joints.

Current System:

Slave manipulator

Gross Stage:

- 4 DOF arm, same as conventional laparoscopic tools.
- No constraint of space & size
- Rigid enough

- Actuated by 4 DC Servo motors using lead screws connected to them.

Milli-Robot

- Small sized, 15mm in diameter
- 2 DOF wrist, yaw & roll axis rotation
- Wrist to gripper length 5 cm
- Actuated with tendons jointly by 3 DC servo motors positioned outside.

Master Workstation

- 6 DOF haptic interface which provides force feedback
- Force reflecting interfaces modified to be similar to the wrist configuration in slave manipulator.

Experimental Results

- Additional 2 DOF (i.e. the 2 DOF wrist) greatly improves the performance at different suturing surface orientations and incision directions.
- Limited ranges of roll & yaw axis of wrist need to be increased.
- The handles at Master end should be comfortable to the surgeons. Interfaces should be of the form with which they are already used to.
- There should be ability to change the end tools of the wrist so that different tasks may be easily performed during the phases of an operation.

- Force feedback is highly important because without it, the surgeon cannot exactly estimate that how much force is being applied or what is happening at patient's end.

Future Work

- The size of the instrument need to be reduced further to perform more delicate operations
- Further small sized instruments will require new mechanical technologies for actuation.
- Develop a system for beating heart surgery

Raven Tele-Surgical Robot

Statistics

- Analysis of the data indicated that 95% of the time the surgical tools were located within a conical range of motion with a vertex angle 60° (termed the dexterous workspace, DWS).
- A measurement taken on a human patient showed that in order to reach the full extent of the abdomen, the tool needed to move 90° in the mediolateral (left to right) and 60° in the superior/inferior direction (head to foot).
- The extended dexterous workspace (EDWS) was defined as a conical range of motion with a vertex angle of 90° and the workspace required to reach the full extent of the human abdomen without reorientation of the base of the robot.

Basic Design

The 7-DOF cable-actuated surgical manipulator is broken into three main pieces

- The static base that holds all the motors
- The spherical mechanism that positions the tool

- The tool interface.

Joints:

The motion axes of the surgical robot are:

- 1) Shoulder Joint (rotational)
- 2) Elbow Joint (rotational)
- 3) Tool Insertion / Retraction (translational)
- 4) Tool Rotation (rotational)
- 5) Tool Grasping (rotational)
- 6) Tool Wrist-1 Actuation (rotational)
- 7) Tool Wrist-2 Actuation (rotational)

Actuators

- The RAVEN utilizes DC brushless motors located on the stationary base, which actuate all motion axes.
- Maxon EC-40 motors with 12:1 planetary gearboxes are used for the first three axes, which see the highest forces.
- The first two axes, those under the greatest gravity load, have power-off brakes to prevent tool motion in the event of a power failure.
- The fourth axis uses an EC-40 without a gearbox, and Maxon EC-32 motors are used for the remaining axes.
- Maxon DES70/10 series amplifiers drive these brushless motors. The motors are mounted onto the base via quick-change plates that allow motors to be replaced without the need to disassemble the cable system.

Zeus Tele-Surgical System

The Zeus robot is particularly well suited to telesurgery for two main reasons:

1) It is already in a master–slave configuration that naturally and cleanly separates into two distinct parts. There is a master console from which the surgeon controls the surgery and a set of positioners and camera-control equipment that is mounted on the operating room table. Though designed for use where the surgeon is in the same operating room with the patient, the equipment comprising Zeus includes console-related electronics and positioner-related servo controllers with cabling interconnecting the two.

2) It is a fail-safe machine that has been used extensively in standard minimally-invasive surgeries.

A. Surgeon's Console

Figure given shows the surgeon's console. In addition to providing a stable platform upon which the control handles are mounted, the console includes a high-quality video monitor for displaying the view from the endoscope and a touch-screen panel for setting various options and interacting with the central control computer. There is remarkably little processing performed on the control handle signals within the master console equipment, especially as compared with the video signals. In essence, the handles' positions and orientations [five degrees of freedom (DOF)] are acquired by a fully redundant set of sensors with the outputs of those sensors appearing directly at the cable socket for attachment to the servo control electronics



residing near the slave robots. Outputs from the surgeon console sensors include analog voltages, quadrature readings (from optical encoders) and digital inputs/outputs (I/Os). Every sensor channel is represented by two implementations so that errors can be detected and appropriately dealt with.

B. Patient-Side Robots

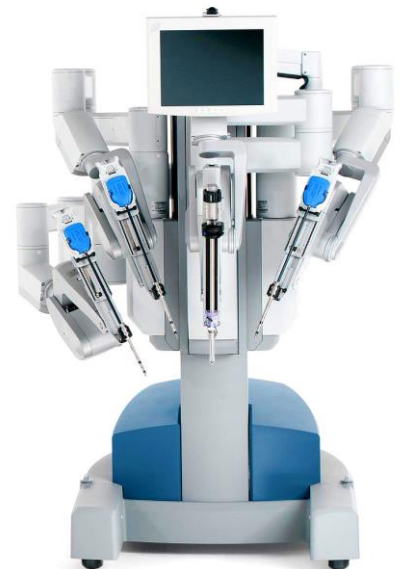
On the patient-side there are several subsystems: three servo control units for three robotic arms (two positioning the instruments and one positioning the endoscopic camera), and one instrument driver/controller to manage the control of graspers, scissors, or other instruments. The signals feeding these various control units come directly from the sensors in the surgeon's master console. There are also digital commands expressed as packet-oriented messages issued by a central processor housed in the surgeon's console and exchanged via RS232 serial connections.



Each controller associated with the patient-side equipment has its own power supply and case. Each unit acts quite autonomously, controlling and driving its resources in accordance with the sensor readings coming from the master console. Coordination between the various positioners and controllers occurs through the serial message exchanges with the central computer that is housed in the master console.

Da Vinci

- i) It Makes laparoscopic surgery even easier for surgeons and less staff is required for the procedure. Surgery is performed without any direct contact between surgeon and patient.
- ii) Surgeon sits a few feet away from operating table at a computer console viewing 3D visual of operative region.
- iii) Surgeon uses 2 masters that each control a mechanical arm of the robot.
- iv) One of the three arms is an endoscope arm which Provides over a thousand frames per second of the instrument position to get rid of possible background noise.
- v) Endoscope is also programmed to regulate the temperature of the tip to prevent any fogging during the procedure. Surgeon can switch to different views easily by the touch of a foot pedal.
- vi) Other two arms are equipped with specialized tools called EndoWristdetachable instruments.
- vii) Through the incisions, the video camera and robotic arms eliminate hand tremors from the surgeon.
- viii) Robot arms also have wider range of motion than the human hand- rotating in 7 different planes and can rotate around completely.
- ix) Each instrument has its own function and can easily be switched using the quick-release lever on each arm.
- x) The device memorizes the position of the arm before being replaced so the next instrument can be set to the same position.
- xi) Surgeon can also choose how much force to be applied- can go from an ounce to several pounds



NETWORK COMMUNICATION

Real Time and Reliable communication is desired in tele-Surgical robots. But it is more idealistic than reality. Although Real time communication is not possible with reliable communication yet communication as close to real time as possible is desired.

There are two basic protocols which are used for communication on transport layer.

- a) Transmission Control Protocol (TCP)
- b) User Datagram Protocol (UDP)

Transmission Control Protocol (TCP)

It is used for connection-oriented transmissions. It undergoes a three-way handshake between server and client to set up connection.

TCP provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer.

a) Reliability

- TCP uses a sequence number to identify each byte of data.
- TCP identifies the sequence number of Packet sent just in case it is lost or received out of order. Sequence number is incremented for every packet sent.
- TCP uses cumulative “ACK”. It means when a receiver sends ACK, it shows all the preceding data has been received.
- Essentially, the first byte in a segment's data field is assigned a sequence number, which is inserted in the sequence number field, and the receiver sends an acknowledgment specifying the sequence number of the next byte they expect to receive.

- For example, if Client sends 4 bytes with a sequence number of 100 then the receiver would send back an acknowledgment of 104 since that is the next byte it expects to receive in the next packet.

b) Ordered Delivery

- Ordered data transfer - the destination host rearranges according to sequence number.
- Retransmission of lost packets - any cumulative stream not acknowledged is retransmitted.
- Error-free data transfer (The checksum in UDP is optional)
- Flow control - limits the rate a sender transfers data to guarantee reliable delivery. The receiver continually hints the sender on how much data can be received (controlled by the sliding window). When the receiving host's buffer fills, the next acknowledgment contains a 0 in the window size, to stop transfer and allow the data in the buffer to be processed.
- Congestion control.^[1]

User Datagram Protocol (UDP)

- UDP uses a simple transmission model. It does not undergo handshaking.
- It does not ensure reliability i.e. Packets may be lost, they may arrive out of order.
- UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level.
- Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.^[1]

Comparison of UDP and TCP:

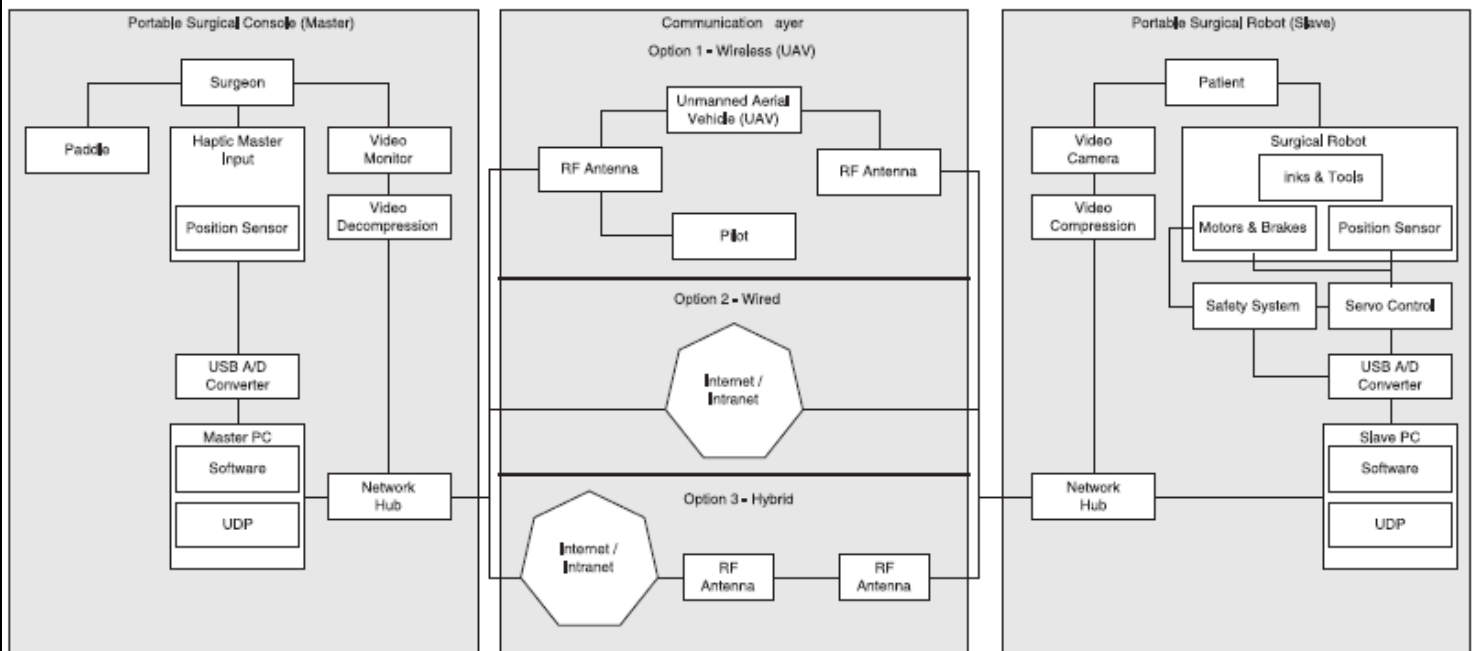
Table 1 Comparison between UDP and TCP

| Key Features | TCP | UDP |
|--------------|-----|-----|
| Reliability | YES | NO |
| Ordered | YES | NO |
| Lightweight | NO | YES |

Raven Tele-Surgical Robot

System Make-Up

- The surgeon site consists of two control devices and a video feed from the operative site.
- The communication layer can be any TCP/IP network including a local private network, the Internet or even a wireless network.
- Patient Side Robot.



Experiments under different Conditions

The RAVEN has been tested in a variety of environments using a multiple communication layer topologies and has demonstrated its portability and robustness.

Following are the results of experiments conducted in different environments:

High Altitude Platforms/Mobile Robotic Telesurgery (HAPs/MRT)

- The datalink was provided by AeroVironment which utilized Internet-style communication at a rate of 1MB per second between the two sites.
- HaiVision Inc. (Montreal, Canada) provided a hardware codec that used MPEG-2 and transmitted the video signal at 800kbps.

Imperial College, London, England (ICL) to University of Washington, 1 Seattle, WA, USA.

- On July 20, 2006, in the lab in Imperial College London (ICL), iChat (Apple Computer Inc) was used for video feedback.
- Time delay between the patient and surgeon sites was about 140 ms for Internet latency (measured by ping) and about 1 second for video encoding/decoding.

NASA Extreme Environment Mission Operations (NEEMO) XII

- Communication between the patient and surgeon sites travelled between UW and NURC via commercial Internet.
- The UDP packet reflector program receives the UDP data packets and routes them to back to the sender, in this case, back to our workstation at the UW.

- Each UDP data packet was time stamped at the workstation in UW and sent to the servers at NURC and Aquarius and the reflected packets were used to measure the elapsed round-trip time between the two locations.
- UDP packet sequence number was also used to measure the number of lost and out-of-sequence packets during the tests.

| Experiment | Date(s) | Patient site | Surgeon site | Communication layer | |
|----------------|-----------------|--|-----------------------------------|----------------------|--|
| | | | | Video | Network architecture |
| HAPs/MRT | June 5–9, 2006 | Field, Simi Valley, CA | Field, Simi Valley, CA | HaiVision Hai560 | Wireless via UAV |
| ICL | July 20, 2006 | BioRobotics Lab, Seattle, WA | Imperial College, London, England | iChat or Skype | Commercial Internet |
| Animal Lab | March 8, 2007 | CVES, Seattle, WA | CVES, Seattle, WA | Direct S-video | LAN |
| NEEMO Aquarius | May 8–9, 2007 | Aquarius Undersea Habitat, 3.5 miles off Florida Keys, 60 ft depth | University of Washington, Seattle | HaiVision Hai1000 | Commercial Internet between Seattle, WA and Key Largo, FL; microwave communication link across 10 miles, Key Largo to Aquarius |
| NEEMO NURC | May 12–13, 2007 | National Undersea Research Center, Key Largo, FL | University of Washington, Seattle | HaiVision Hai200 | Commercial internet |

Figure 1 Summary of Tele-Surgical Experiments

| Experiment | Mean Network Latency (ms) | Significance |
|----------------|---------------------------|---|
| HAPs/MRT | 16 | Operated in a field environment to test ruggedness and portability. Communicated via wireless through a UAV. |
| ICL | 172 | Adaptability of surgeon site to other Sensable devices. Teleoperation over long distance. |
| Animal lab | 1 | Demonstrated ability to operate on a real patient through MIS ports. |
| NEEMO Aquarius | 76 | TeleRobotic FLS for performance measurement. Operating in a unique environment. Communicating across both commercial Internet and long-distance wireless. |
| NEEMO NURC | 75 | Additional opportunity to collect TeleRobotic FLS data over long communication network. |

Figure 2 Mean Network Latency and Significance of Each

ZEUS Tele-Surgical Robot

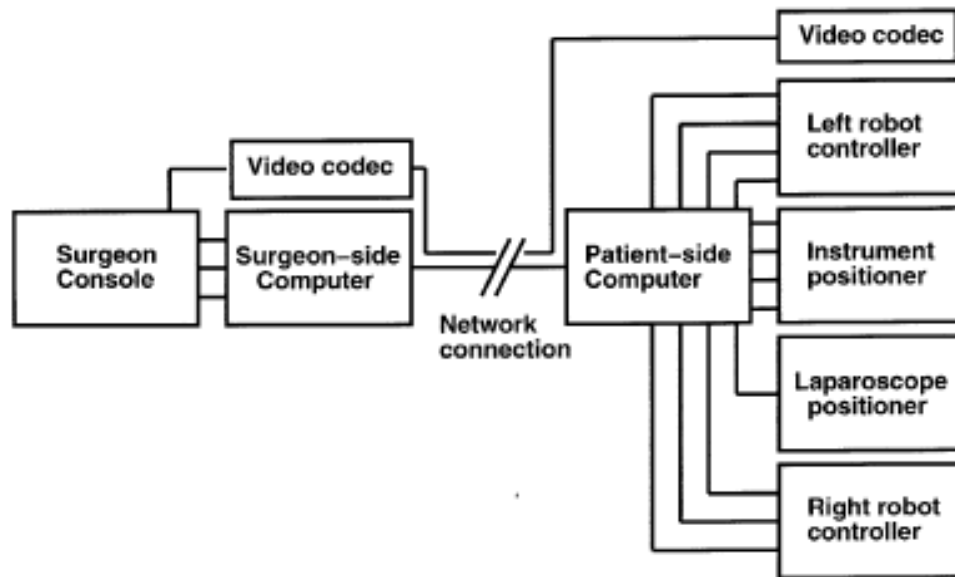


Figure 3 Zeus Tele-Surgical System

Operation Lindbergh

The Operation Lindbergh tele-surgery procedure was done using the Zeus™ robotic surgical system. It quickly became apparent that the system could be efficiently employed for a broad variety of surgical disciplines, including general surgery, thoracic surgery, gynecology, urology, etc.

Graphic of Operation Lindbergh

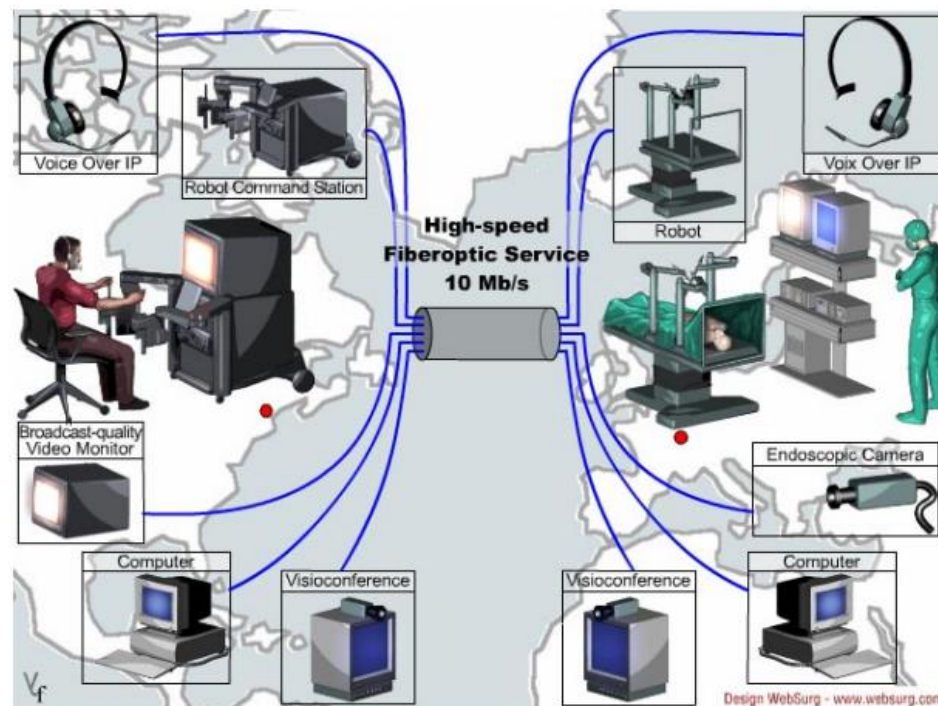


Figure 4 Graphic of Operation Lindbergh

Based on experience with transcontinental and transatlantic telecommunication circuits, the minimum round-trip latency for Operation Lindbergh was anticipated to be in the range of 110–140 ms. Number of experiments were conducted as part of the project to evaluate the effect that various amounts of latency had on tele-surgery. It was found that latencies up to **330 ms** were manageable and safe for the type of surgery that was to be performed, given the excellent quality-of-service conditions of our networking infrastructure.

Zeus Network

- All message exchanges are based on a simple custom packet-based protocol.
- Packets have sequence numbers, and every packet that is sent is acknowledged by the receiver.
- In the standard Zeus system, the timeouts for message exchanges have a short fixed setting well below the communication delays expected and experienced during transatlantic cases.
- The communicating computers use 100 base-T Ethernet operating with internet user datagram (IP/UDP) protocol.
- Ethernet was the communications interface of choice because of its wide acceptance, availability, and reliability.
- Because of this choice, the system can easily be staged or tested almost anywhere without any special communications circuits or equipment.
- For Operation Lindbergh, the New York site was connected to the Strasbourg, France site via a single private virtual channel provided by a commercial telecommunications carrier. The provider, France Telecom, chose to use a 10 Mb/s constant bit rate (CBR).
- Robot control used far less bandwidth than the video.
- The payload of the UDP packet sent from the surgeon side to the patient side was 152 bytes in size with 128 packets sent each second.
- The payload of the UDP packet sent from patient side back to surgeon side was 88 bytes in size with 128 packets sent per second.
- The total communications bandwidth required for the UDP payload corresponding to the remote controls for the robot was only 30 720 b/s.
- With IP/UDP overheads and ATM/SONET framing added by the telecom gear, the bandwidth used by the robot was well under 60 Kb/s, including all embedded serial communications streams as well.

No. of Packets per second = 128

Size of Each Packet from Surgeon to patient side = 152 bytes = 152×8 bits = 1216 bits

Data rate = 128×1216 = 155 kbps

Size of Each Packet from Patient to Surgeon Side = 88 bytes = 88×8 = 704 bits.

Data rate = 128×704 = 90 kbps

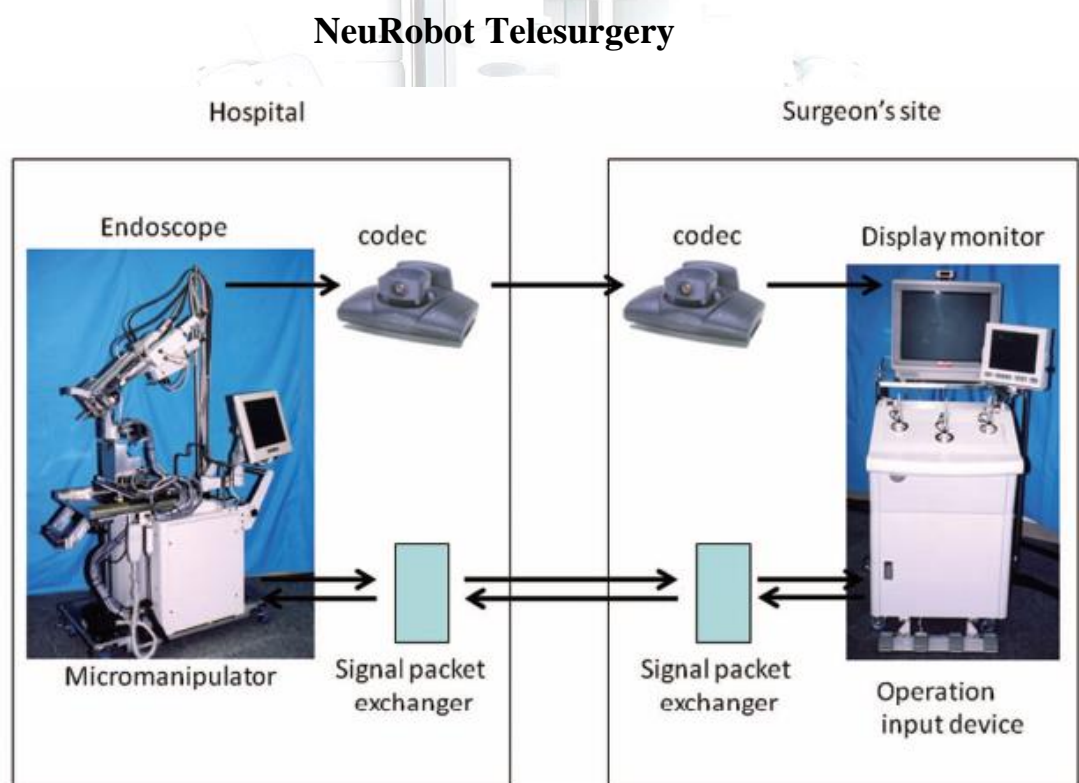


Figure 5- An Illustration of NeuRobot for tele-surgery

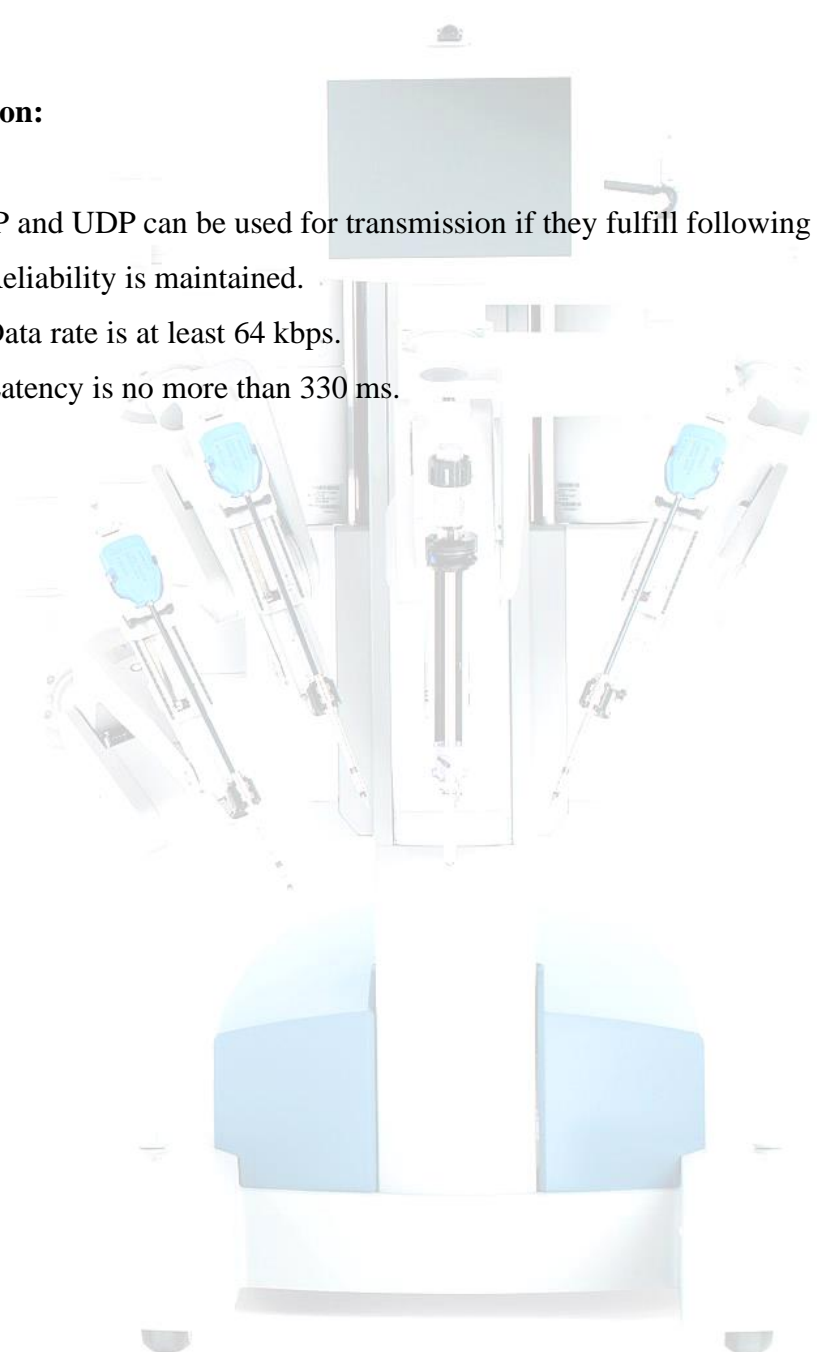
The signal packet exchanger changed the signal to 400 bytes of User Data Protocol (UDP) packet and transmitted in 20 cycles per second, which gave 64 kilobit per second (kbps) connection rate. NeuRobot was re-programmed to stop the movements of micromanipulator automatically for safety reason when the manipulator did not

receive a consecutive 5 packets (250 ms) between the micromanipulator and operation input device.

Conclusion:

Both TCP and UDP can be used for transmission if they fulfill following conditions:

- i) Reliability is maintained.
- ii) Data rate is at least 64 kbps.
- iii) Latency is no more than 330 ms.



COMMUNICATION BETWEEN PATIENT SIDE COMPUTER AND ROBOTIC ARM

The most important requirement of telesurgical robot is to create a communication link between computer and motor drivers. Each system use either USB or Serial to achieve this task. Both of these links have some advantages and disadvantages.

USB vs. RS232

RS232 is use for serial communication. It defines the interface layer. To use Rs232, application specific software must be written on both ends o f the connecting RS232 cable. The developer has freedom to write his own protocol. RS232 ports can be either accessed directly by an application or via a device driver.

USB on the other hand is a bus system which allows more than one peripheral to be connected to a host computer via one **USB** port. Hubs can be use to increase the number of devices which can be connected at a single time. The standard describes the physical properties of the interference as well as the protocol of communication. Because of the complex **USB** protocol requirements, communication with **USB** ports on a computer is always performed via a device driver.

The RS232 interface protocol provide facility of changes setting like baud rate, data bits, hardware software flow control can often be changed within the application. The **USB** interface does not give this flexibility. Many applications expect certain timing with **RS232** communications. With ports directly fitted in a computer this is most of the time no problem. Communication congestion may be the result of this, and the timeframe in which specific **RS232** actions are performed might not be so well defined as in the direct port approach. Also, the double device driver layer with an **RS232** driver working on top of the complex **USB** driver might add extra overhead to the communications, resulting in delays.

Hardware specific problems

RS232 ports require three power sources: +5 Volts for the UART logic, and -12 Volts and +12 Volts for the output drivers. **USB** however only provides a +5 Volt power source. Some **USB** to **RS232** converters use integrated **DC/DC** converters to create the appropriate voltage levels for the **RS232** signals, but in very cheap implementations, the +5 Volt voltages is directly used to drive the output.

RS232 to USB converter is not compatible to all devices it is possible that converter work with one device but it may not work with some other devices. This can particularly become a problem with industrial applications.

Another hardware specific problem arises from handshaking to prevent buffer overflows at the receiver's side. **RS232** supplications can use two types of handshaking, either with control commands in the data stream, called software flow control, or with physical lines, called hardware flow control.

Comparison

Here is how USB stack up against other protocols available for other host adaptors:

| | USB | Serial | Parallel |
|--------------------------|-------------|-----------------------|------------------------------|
| Industry Standard | Yes | Yes | No |
| Bandwidth | 12 Mbps | 115 Kbps | 115 KBps EPP/ECP - 3 MBps |
| Number of | 127 devices | Limited to the number | Limited to the number |

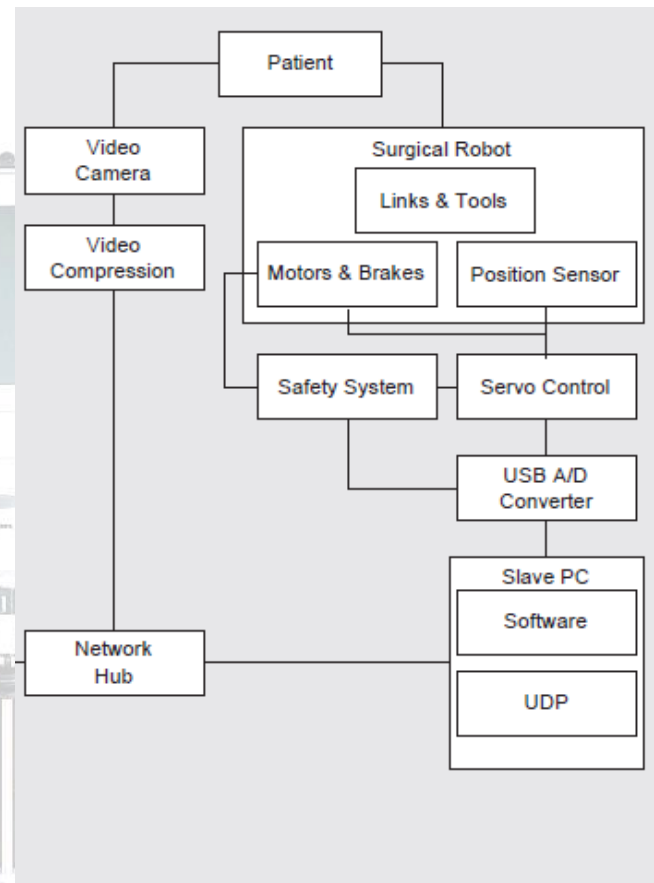
| | | | |
|---------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| Devices | on a single USB bus | of ports available on the computer. | of ports available on the computer. |
| Bus Power | Yes, can provide up to 500 mA at 5V | No | No |
| Cable Length Limit | 5 m / 16.4 ft | 3 m / 10 ft | 1.8 m / 6 ft |
| Plug'n'Play | Yes | No | No |
| Hot Swappable | Yes | No | No |

USB to RS232 Converter

It is a device which connects USB port to serial port. By using USB to Serial Converter USB port act as serial port. We can communicate with serial port directly. When however an **RS232** port is used via an **USB** to **RS232** converter, this flexibility should be present in some way. Therefore to use an **RS232** port via an **USB** port, a second device driver is necessary which emulates a **RS232** UART, but communicates via **USB**.

RAVEN

The patient side of Raven system consists of a computer which runs control software in the kernel space of a RTAI Linux computer at rate of 1 kHz. USB 2.0 interface board is used to provide a communication link between control software and motor controllers. USB board provides eight channels of high-resolution 16 bit D/A for control signal output to each controller and eight 24bit quadrature encoder readers.



ZEUS

The patient side of Zeus system consists of computer which receive signal from surgeon's master console. There are also digital commands expressed as packet-oriented messages issued by a central processor housed in the surgeon's console and exchanged via RS232 serial connections. Each controller associated with patient side equipment work in accordance with the sensor reading coming from master console. Coordination between the various positioners and controllers occurs through the serial message exchanges with the central computer that is housed in the master console.

Graphical User Interface

It is a user interface that allows users to interact with electronic devices with images rather than text commands.

It is requirement of all the system to provide such a interface to it user which is easy to understand and convenient to be use.

Raven:

The SGUI allows the user to execute high-level commands during operation. It was written using Qt 4.1.2 from Trolltech, Inc. It has password verification feature for security purpose.

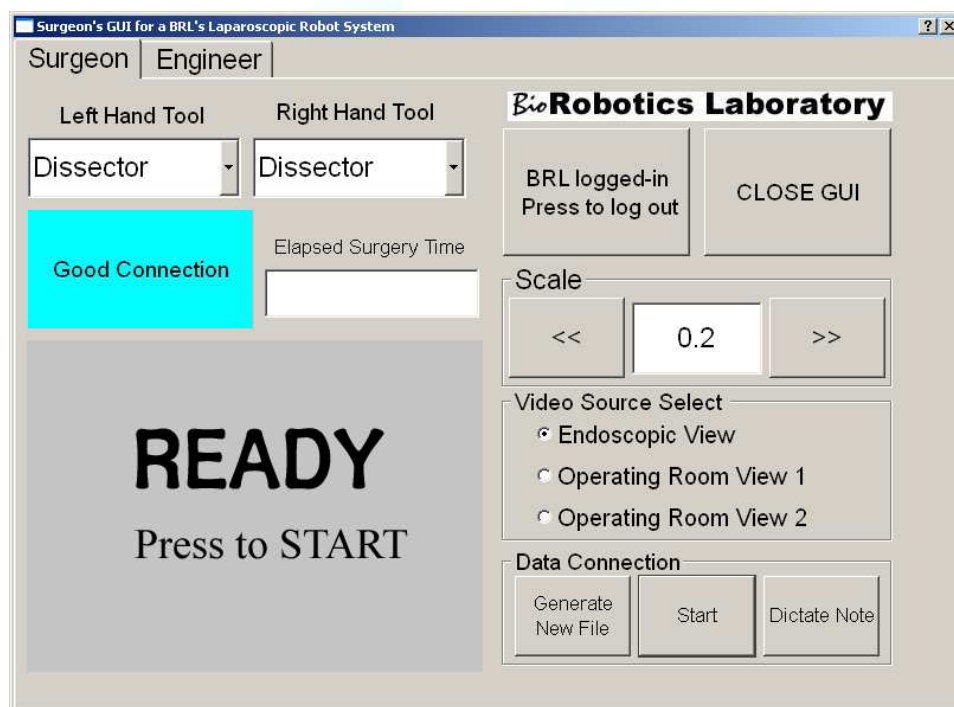
A SGUI have two windows which can be selected from SGUI tabs on top.

Surgeon Window:

It provides information and options which is required by surgeon. For example, a variable scale factor which control movement of the surgeon. In addition to that it displays the status of the system.

Engineer Window:

In this technical parameter of the system are set. The IP of the computer at the patient side can be selected through this window. It also allow variable transmission rate for optimal performance.



INPUT TO SURGEON COMPUTER

There are many ways to take input from human being but most commonly used devices are as followed:

- 1) Joystick
- 2) Omni devices
- 3) CyberGlove

Omni devices

It is a haptic device which simulates a pen in 3D free space. It uses software called VTX Designer. It is developed by Sensable Technologies. It is a company which was created on the basis of research done at MIT at undergraduate level in 1990s by industry pioneers Thomas Massie and Dr. Kenneth Salisbury, and is headquartered in Woburn, MA.

The Phantom Omni devices and Phantom Desktop devices offer a affordable desktop solutions.

Phantom Desktop device

It provides higher fidelity, stronger forces, and lower friction as compared to phantom Omni device.



Phantom Omni device:

PHANTOM Omni is a lowest-priced haptic device.

The PHANTOM® Omni Developer Kit includes the PHANTOM Omni device & the OpenHaptics toolkit.



openHapticsToolkit

The open Haptics toolkit provide developer a way to use haptic and true 3D navigation to a board range of applications. This toolkit handles complex calculation, provides control for developer. It support a wide range of Phantom devices.

Cyberglove:

The cyberGlove uses virtual technologies' proprietary resistive bend-sensing technology to accurately transform hand and finger motions into real-time digital joint-angle data. The virtualHand suite 2000 software convert the data into a graphical hand. There are total 22 sensors

i.e. three flexion sensors per finger, four abduction sensors, a palm-arch sensor, and sensors to measure wrist flexion and abduction.



CyberTouch:

It provide s CyberTouch vibro-tactile feedback option for the CyberGlove,. Virtual Technologies, Inc. has added touch feedback to its industry leading hand-sensing product which enables CyberGlove users to manually experience virtual worlds.

Raven:

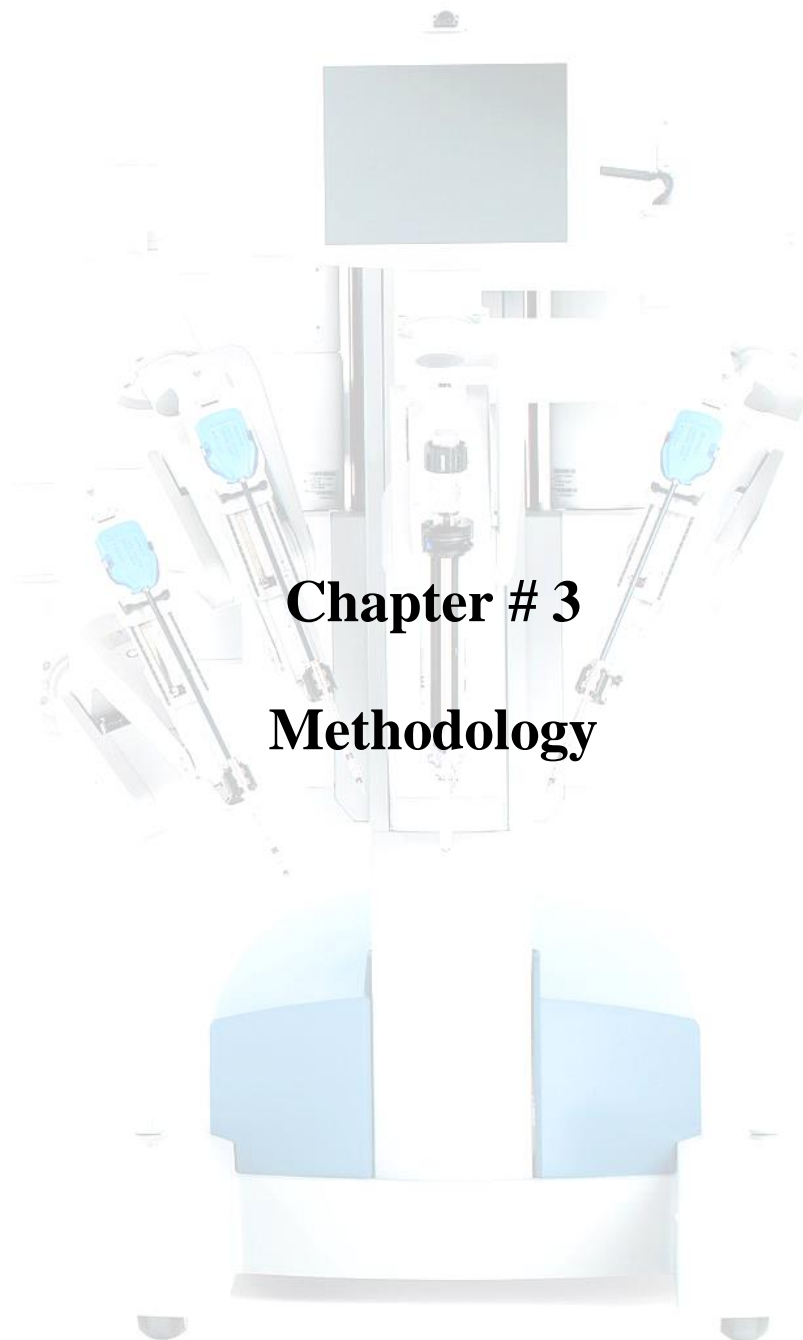
The surgeon site was developed to be low cost and portable, choice that allows for easier telesurgical collaboration. It consists of two Phantom Omni devices (SensAble

Technologies, Woburn, MA), a USB foot-pedal. SensAble's Phantom haptic devices are well established amongst haptics researchers with a development environment that is straight forward to use. The Omni is a cost effective solution that allowed us to quickly implement a surgeon interface device for our master/slave system.



UC berkley:

- **Joystick:** Pencil-sized; 1 for each hand



Chapter # 3

Methodology

NETWORK COMMUNICATION

Our first requirement was to make a reliable connection between surgeon and patient side.

TCP has inbuilt reliability mechanism process. Therefore, we started to work on establishing connection between client and server using TCP.

First of all we should be clear that how TCP connection is established.

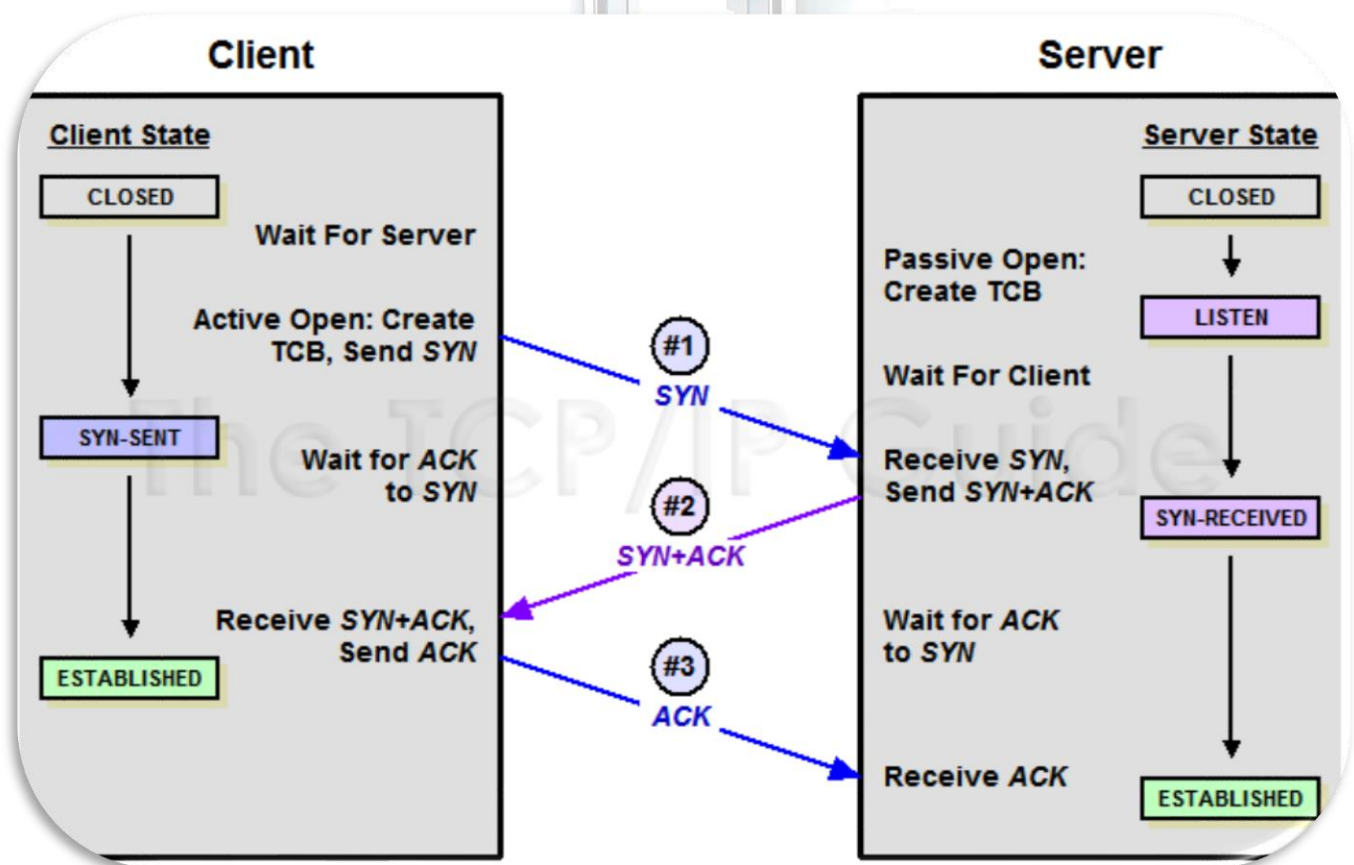


Figure shown below demonstrate the making of connection between client and server.

- Client sends a TCP SYNchronize packet to Server
- Server receives Clients's SYN
- Server sends a SYNchronize-ACKnowledgement
- Client receives Server's SYN-ACK

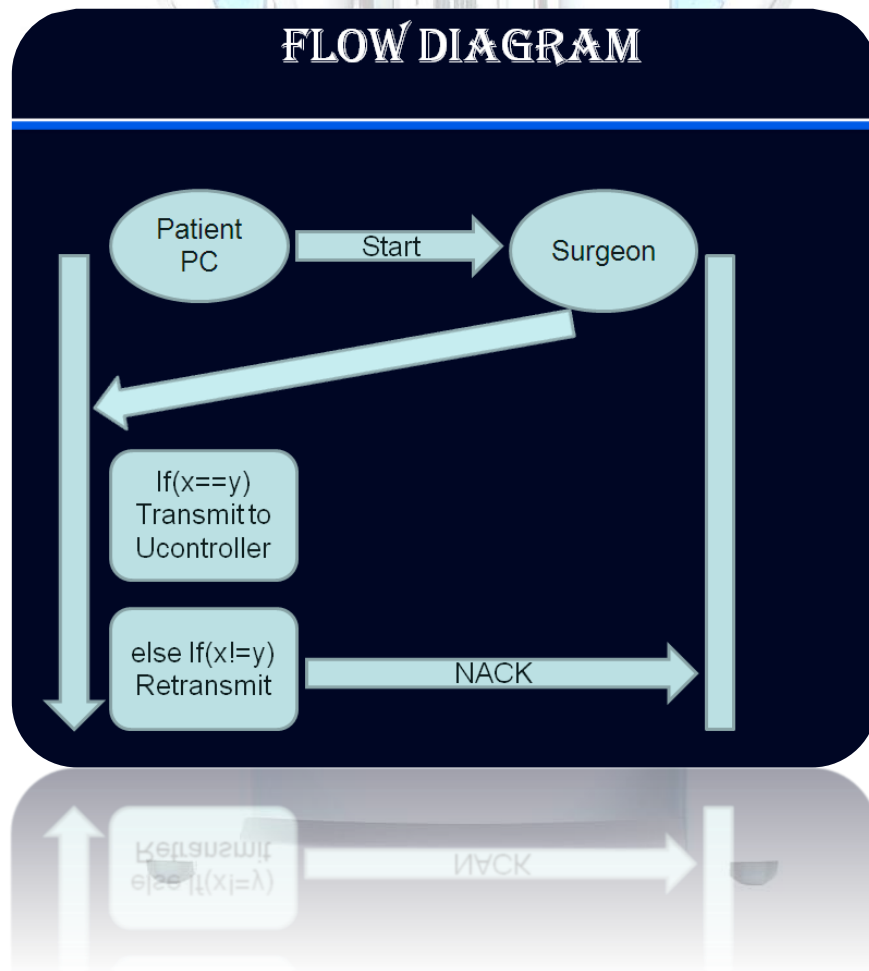
- e) Client sends ACKnowledge
- f) Server receives ACK.

TCP socket connection is ESTABLISHED.

We worked on this basic principle and were able to make a TCP connection. Its code in C++ is mentioned in appendix A. There were some shortcomings in the code so it was modified.

For further improvement in delays we moved to UDP instead of TCP. Reliability was also introduced in UDP. Sequence number was attached to the packet and we used NACK to check whether packets have been lost or not.

Figure below shows the Flow Diagram of Hybrid UDP.



It operates in following steps:

- Client sends a SYNchronize packet to Server
- Server gets IP of client and start sending data to it.
- Client receives data.
- If data is out of order or packet loss has occurred it asks server to resend the packet which is not received.

Some of the functions which are used in the code are explained below:

socket()

It returns a new socket descriptor that can be used to do sockety things with. This is generally the first call in the whopping process of writing a socket program, and its results can be used for subsequent calls to `listen()`, `bind()`, `accept()`, or a variety of other functions.

```
int socket(int domain, int type, int protocol);
```

bind()

When a remote machine wants to connect to server program, it needs two pieces of information: the IP address and the port number. The `bind()` call allows to do just that.

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Return Value

Returns zero on success, or -1 on error (and **errno** will be set accordingly.)

listen()

Socket descriptor (made with the `socket()` system call) can be told to listen for incoming connections. This is what differentiates the servers from the clients. The `backlog` parameter can mean a couple different things depending on the system, but loosely it is how many pending connections tolerated before the kernel starts rejecting

new ones. So as the new connections come in, quickly **accept()** them so that the backlog doesn't fill.

Try setting it to 10 or so, and if clients start getting "Connection refused" under heavy load, set it higher. Before calling **listen()**, server should call **bind()** to attach itself to a specific port number.

That port number (on the server's IP address) will be the one that clients connect to.

Return Value

Returns zero on success, or -1 on error (and **errno** will be set accordingly.)

```
int listen(int s, int backlog);
```

accept()

After listening **accept()** is called to actually get a new socket descriptor to use for subsequent communication with the newly connected client.

The old socket that was being used for listening is still there, and will be used for further **accept()** calls as they come in.

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

send(), sendto()

These functions send data to a socket. **send()** is used for TCP **SOCK_STREAM** connected sockets, and **sendto()** is used for UDP **SOCK_DGRAM** unconnected datagram sockets. With the unconnected sockets, destination of a packet must be specified each time packet is sent, and that's why the last parameters of **sendto()** define where the packet is going.

```
send(int s, const void *buf, size_t len, int flags);
```

```
sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```

With both **send()** and **sendto()**, the parameter *s* is the socket, *buf* is a pointer to the data you want to send, *len* is the number of bytes to be sent, and *flags* allows to specify more information about how the data is to be sent.

recv(), recvfrom()

Once a socket is up and connected, incoming data can be received from the remote side using the **recv()** (for TCP SOCK_STREAM sockets) and **recvfrom()** (for UDP SOCK_DGRAM sockets).

```
recv(int s, void *buf, size_t len, int flags);  
recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

Data Rate and Latency

Although we have made a reliable connection which can send any number of control signals from one side to another yet we have not calculated its data rate and latency. Our next step is to calculate Latency and data rate.

Data Rate

To find data rate we sent packets from server to client and then checked how many packets have been sent in certain time. Code given below is capable of fulfilling our requirements.

Number of Experiments was performed.

We sent packets to another PC through commercial internet and checked the number of received packet in a specified time interval. Dividing the number of packets with that time interval gave us the required data rate.

Delay

Round trip Time (RTT) was calculated. Packet was sent from surgeon to patient side and then it was echoed to find out the RTT.

SERIAL COMMUNICATION

As in our project computer is involved which major purpose is to receive signal from surgeon and transfer this signal into commands which is then transmitted to microcontroller. We need some way to communicate with microcontroller.

There are three way to communicate with microcontroller

- 1) USB Port
- 2) Serial Port
- 3) Parallel Port

Most commonly USB and serial port is use for communicating with microcontroller.

We selected Serial port over USB because

- a) We have to write driver for USB device which is difficult in C++.
- b) Almost all microcontrollers which are available in markets of Pakistan do not directly support USB interface.
- c) USB have its own fixed protocol so we cannot change protocol whereas in serial communication we have freedom of setting your own standards.
- d) Resources for serial communication are widely available on internet.

There are some issues which was needed to be addressed before using serial communications

- a) Serial Communication is slow as compared to USB
- b) Most of the laptops don't have serial port.

These problems can be solved by using serial port at higher baud rate and using USB to serial port converter.

Serial Communication

It is the process of sending data one bit at a time, sequentially, over a communications channel.

Parameters

These are some parameter on which both side must agree to communicate with each other:

- 1) Baud Rate
- 2) Data bits
- 3) Parity
- 4) Stop bit

Baud Rate

Baud rate is a measure of how fast data (symbols) are moving between two systems.

Data bits

It is the number of bits in a single frame.

Parity bit

It is used for error checking.

Stop bit

Stop bits are used to signal the end of communication for a single frame.

USB to Serial Converter

It is a device which connects USB port to serial port. By using USB to Serial Converter USB port act as serial port. We can communicate with serial port directly.

Explanation

This code can transmit and receive data from the serial port simultaneously. It makes two thread one for transmit data to the serial port and second to receive data from serial port.

Initial() Function

This function purpose is to initialize the Serial port. This initializes serial port to the required Baud rate, parity and other parameters. It also Specify time-out between characters for receiving. So this function decides the protocol of serial communication.

Main() Function

In this first we create two threads by using createthread function and give these threads a starting address. If for reason if thread can be created then this will tell error.

write(char*) Function

This function is used to transmit a single character through serial port.

DWORD WINAPI StartThre(LPVOID iValue) function

This is the starting address of one of the thread. It takes a single character input and transmits it through serial port.

DWORD WINAPI StartThread(LPVOID iValue) function

This is the starting address of a thread which function is to receive data from the serial port.

GUI IMPLEMENTATION

GLUT

The OpenGL Utility Toolkit (GLUT) is a programming interface for C programs for writing window system independent OpenGL programs.

It provide a platform which allows developer to write code for graphics and interfacing external devices like joysticks which requires continuous polling to take data from the joystick.

GLUI

GLUI is a GLUT-based C++ user interface library which provides controls such as buttons, checkboxes, radio buttons, spinners, and list boxes to OpenGL applications. It is window-system independent, relying on GLUT to handle all system-dependent issues, such as window and mouse management.

This is the GUI of our system at the surgeon side.

JOYSTICK INTERFACING

A joystick is an input device consisting of a stick that pivots on a base and reports its angle or direction to the device it is controlling.

Code Explanation:

glutInit Function:

glutInit is used to initialize the GLUT library.

argc : A pointer to the program's *unmodified* argc variable from main. Upon return, the value pointed by argc will be updated, because glutInit extracts any command line options intended for the GLUT library.

argv: The program's *unmodified* argv variable from main. Like argc, the data for argv will be updated because glutInit extracts any command line options understood by the GLUT library.

Description

glutInit will initialize the GLUT library and negotiate a session with the window system. During this process, glutInit may cause the termination of the GLUT program with an error message to the user if GLUT cannot be properly initialized.

glutInitDisplayMode

glutInitDisplayMode sets the initial display mode.

glutInitWindowPosition

glutInitWindowPosition set the initialwindow position.

glutInitWindowSize:

glutInitWindowSize set the initialwindow size.

glutDisplayFunc:

glutDisplayFunc sets the display callback for the current window.

glutMainLoop

glutMainLoop enters the GLUT event processing loop.

Description

glutMainLoop enters the GLUT event processing loop. This command is called at most one time. It will call all other callbacks that have been registered.

glutJoystickFun(func,10)

It is a joystick callback for the current window.

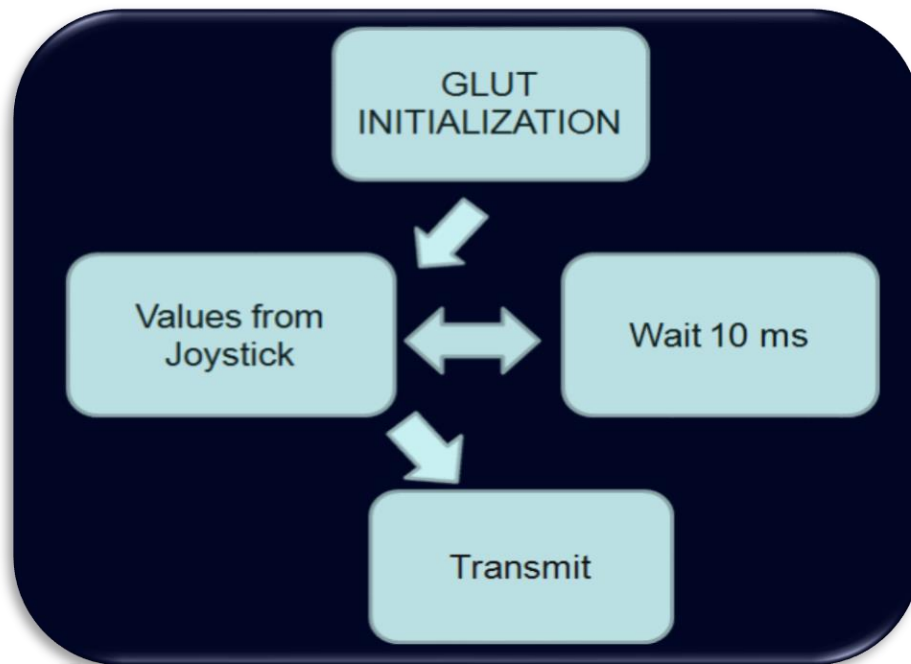
Func:

The function which is to be called.

pollInterval:

It is a polling time in millisecond.

Block Diagram:



Threads

A computer program which run two or more concurrently running tasks.

We have used threading in serial communication as well as networking.

Explanation:

We have made two threads which allow computer to transmit and receive data at the same time. Similarly we have use two threads with TCP i.e. one for transmitting data to the network and one for receiving data from the network.

VIDEO STREAMING

One of the most important requirements of Tele-surgery is video streaming. As surgeon is far away from the patient therefore video streaming is the only way which can provide surgeon the real time video of what actually happening at the patient side. According to the research video should not have delay more than 1 second else it will become really difficult for surgeon to perform surgery.

Different ways for video streaming

- 1) Skype
- 2) Media Player
- 3) Veetle
- 4) OpenCV

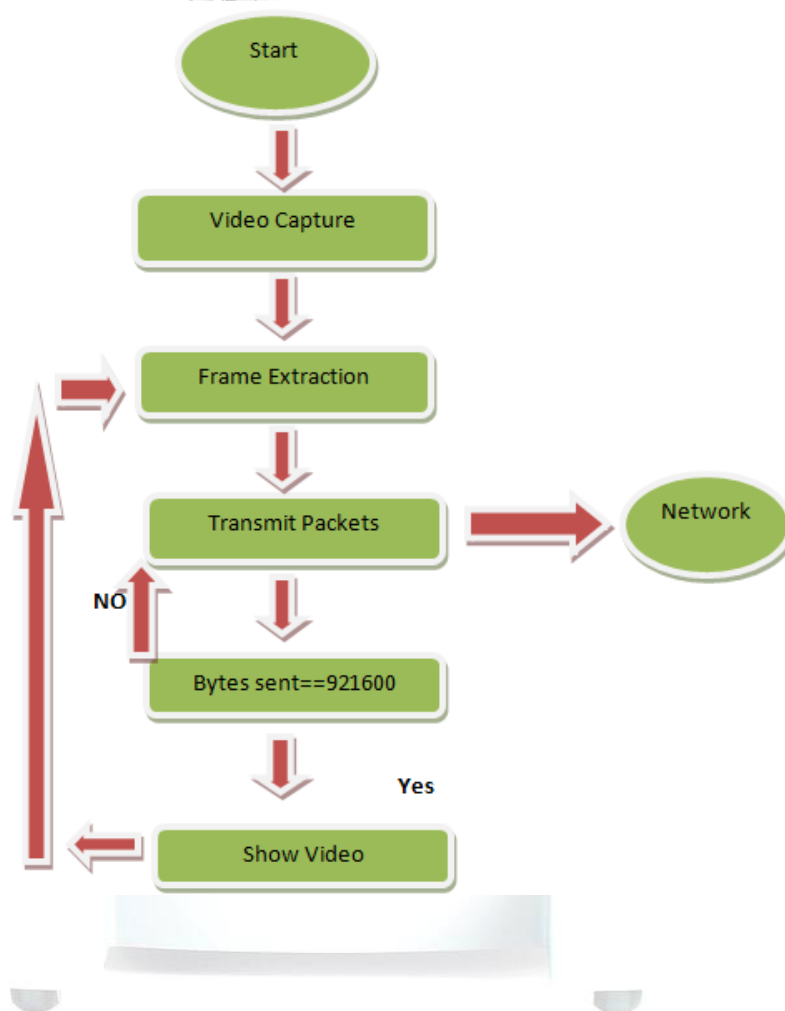
I have tried all of the above method for video streaming but I found in all of them OpenCV is the best way for video streaming because it provide us a complete control on video streaming algorithm and at the same time we can apply different compression techniques in order to reduce the network delay. It also provides us a way to add image processing in it.

OpenCV

OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real time computer vision. The library is cross-platform. It focuses mainly on real-time image processing.

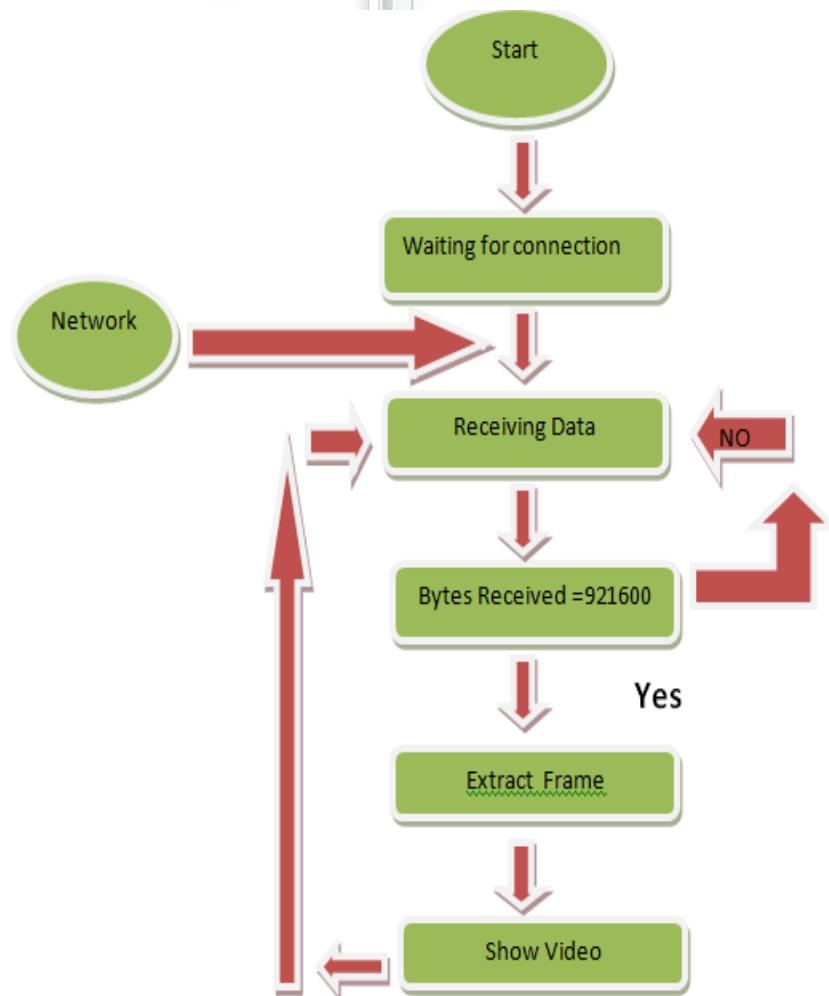
Video Transmission Flowchart

Video is first captured from camera and then frame is extracted from the video which is transmitted using udp. In each frame we have 921600 bytes which has to be transmitted. Once a single frame is completely transmitted it capture next frame.



Video Receiving Flowchart

A process on the surgeon waits for the data from the patient side. Once it starts receiving packets it waits till it received 921600 bytes then it extract frame from the packets and then start receiving data of next frame.



EMBEDDED & CONTROL SYSTEMS

In this project, there is a large section of embedded systems and control systems at the patient side. The instructions and commands given from the surgeon side need to be properly handled in order to take appropriate action.

Some of the major goals we want to achieve at patient side under this domain are;

- Precise control of motors
- Simultaneous movements of different links
- Quick response to command

In this section, we aim to target different solutions to meet the above mentioned requirements and functionalities.

PID Basics

Introduction & a Brief History

“PID” is an acronym for “proportional, integral, and derivative.” A PID controller is a controller which uses these three mathematical functions to effectively control the underlying process. The PID controller was first placed on the market in 1939 and has remained the most widely used controller in process control until today. A research carried out in 1989 in Japan indicated that more than 90% of the controllers used in process industries are PID controllers or the advanced versions of the PID controller.

Why PID Controller?

PID controllers are everywhere these days, e.g. temperature, motion and flow controllers. A PID controller is very important because it helps the output of the system (velocity, temperature, position) at the desired level

- ✓ In a minimum possible time
- ✓ With minimum overshoot
- ✓ With little error

Thus we can identify the major goals of using a PID controller as follows;

- Decreasing the Rise time and the settling time of the system
- Decreasing the percentage overshoot
- Minimize the steady state error

PID CONTROLLER ALGORITHM

As the name suggests, a PID algorithm consists of three basic coefficients: proportional, integral and derivative. These gains are varied to achieve an optimal system response. The basic algorithm of a PID controller is as follows:

- The system output (also called the process variable) is read with a sensor and compared with a reference value (also called the set-point)
- The comparison of reference and the measured output values results in an “Error” value which is used to calculate proportional, integral and derivative responses.
- These three responses are then summed to obtain the output of the controller to get closer to the set-point.
- The output of the controller is used as an input to the system which is to be controlled. For example, to control a motor, the controller would provide more or less current. To control the flow of a fluid, the controller would cause a valve to open or close.

- The system output is then measured again and the whole iterative process repeats again and again.

We can represent the manipulated variable or the output of the controller in terms of its three correcting terms, which is the basic equation of PID controller;

$$MV(t) = P_{out} + I_{out} + D_{out}$$

where P_{out} , I_{out} , and D_{out} are the contributions to the output from the PID controller from each of the three terms, Proportional, Integral and the Derivative respectively.

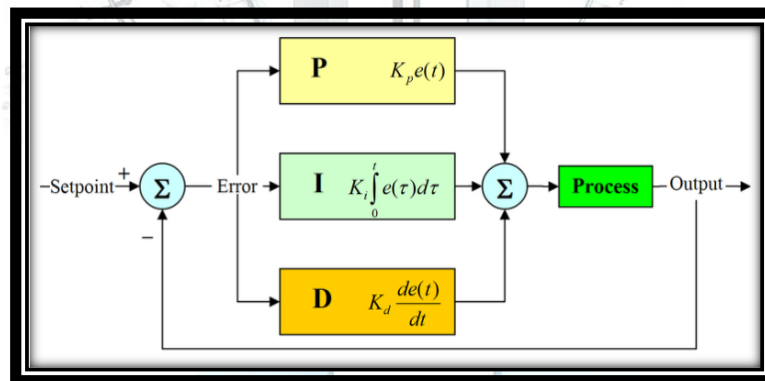


Figure 5: PID Block Diagram

The PID controller is scalable and tunable. It is scalable in the sense that one can implement a simpler controller by just using the proportional gain, or a combination of the proportional gain and either the integral or derivative gain (P, PI, PD, or PID controllers). The system is tunable because P, I, and D gains can be adjusted to tune the controller for the specific system.

The individual role of each term of a PID controller is explained below.

Proportional Term

The proportional term depends only on the difference between the set point and the measured output value. This difference is referred to as the Error term. The *proportional gain* (K_p) determines the ratio of output response to the error signal. In general, increasing the proportional gain will increase the speed of the control system response. Essentially it improves the rise time of the system. But if the proportional gain is too large, the output will begin to oscillate. If the gain is increased further, the oscillations will become larger and the system will become unstable and may even oscillate out of control.

On the other hand, too small proportional gain results in a small output in result to a large input hence a less responsive/sensitive controller.

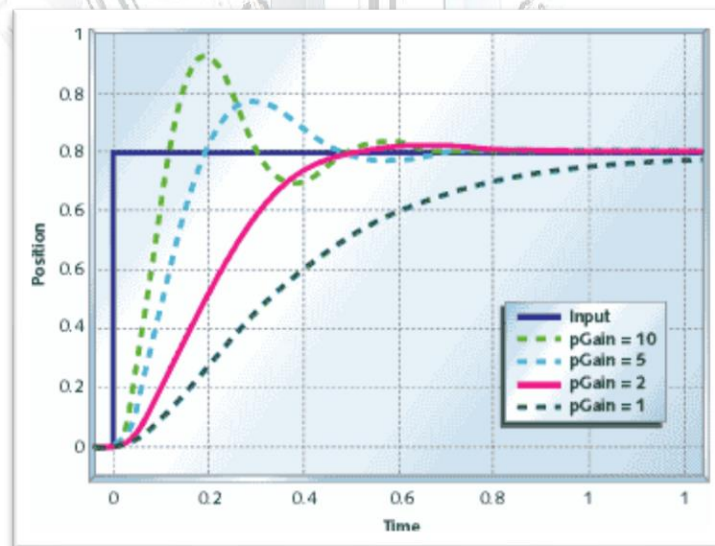


Figure 6: Effect of varying K_p

Integral Term

The integral component sums the error term over time. The result is that even a small error term will cause the integral component to increase slowly. This accumulated error is multiplied with the *integral gain* (K_I) and added to the output of the controller. The integral response will continually increase over time unless the error is zero, so the effect is to drive the Steady-State error to zero. The integral term is perfect for fixing small errors. Since the integral adds up the errors, several consecutive small errors eventually makes the integral big enough to make a difference.

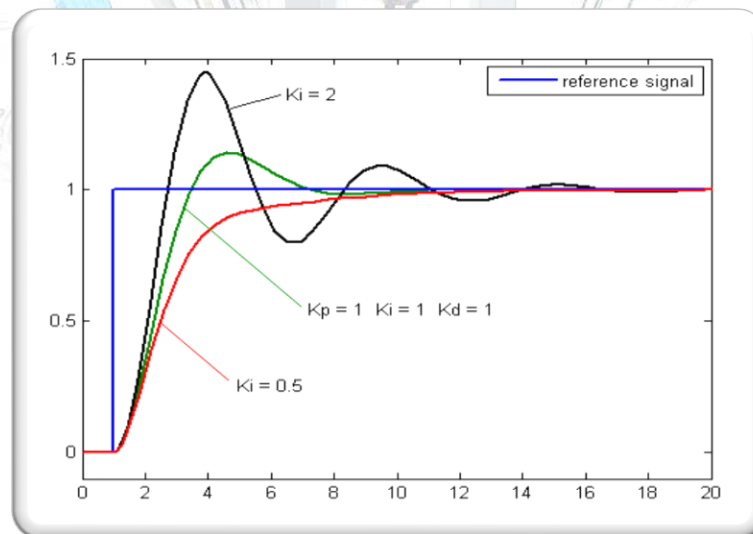


Figure 7: Effect of varying K_i

Derivative Term

The derivative response is proportional to the rate of change of the error with respect to time. This rate is multiplied with the *derivative gain* (K_D) and added to the output. If the current error is worse than the previous error then the D term tries to correct the error. If the current error is better than the previous error then the D term tries to stop the controller from correcting the error. It is the second case that is particularly useful. If the error is getting close to zero then we are approaching the point where we want to stop correcting. Hence, increasing the derivative gain will reduce the magnitude of the overshoot produced by the integral component and improve the system stability. However, a controller with large derivative gain is highly sensitive to noise in the error term, and can cause a process to become unstable if the noise and the derivative gain are sufficiently large.

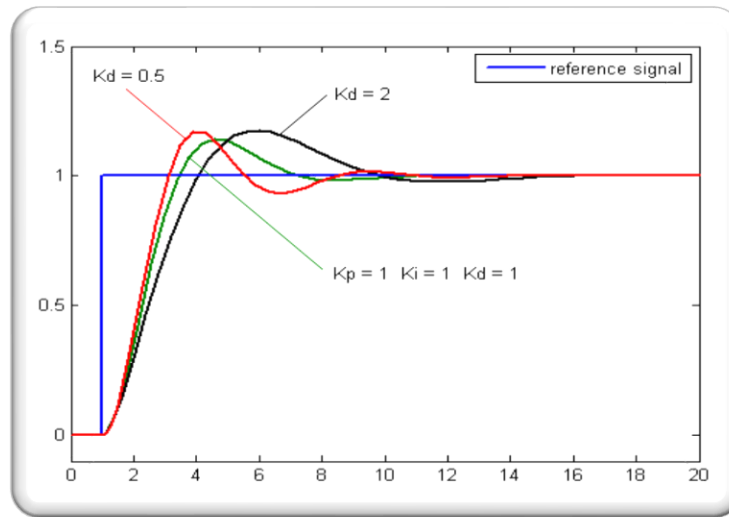


Figure 8: Effect of varying K_d

Summary

We can summarize the effects of increasing the PID parameters on the system's output response in the following tabular form;

Table 2: Effects of increasing the parameters

| Effects of <u>increasing</u> parameters | | | | |
|---|--|-----------|---------------|----------------------|
| Parameter | Rise time | Overshoot | Settling time | Error at equilibrium |
| K_p | Decrease | Increase | Small change | Decrease |
| K_i | Decrease | Increase | Increase | Eliminate |
| K_d | Indefinite (small decrease or increase) | Decrease | Decrease | None |

PID Implementation for DC motor speed control

H-Bridge Driver

H-Bridge driver is the most commonly used driver for controlling the DC motor. It provides the following functionalities;

- **Speed Control**

In order to control the speed of a DC motor, instead of applying a constant voltage across the motor, we repeat switching it ON and OFF at a very fast rate with a fixed voltage (V_{cc}) applied to it. This is done by sending a train of pulses (PWM) to a power MOSFET in order to turn it on and off. Then, the motor sees the average voltage while it depends on duty cycle of PWM pulses. The speed of rotation is proportion to this average voltage.

By PWM method, it is easier to control the DC motor than by directly controlling the voltage across it. All we have to do is to change the duty cycle of pulses according to the required speed. Also, a power MOSFET consumes negligible power in switching.

- **Direction Control**

The direction of rotation of motor is controlled by the direction of current flow through the motor. H-Bridge consists of four Power MOSFETs which are connected in an arrangement like the alphabet “H”. We give the control signals from the Microcontroller to H-Bridge to control the direction of rotation of the motors.

- **Components Used**

The transistors we used for our H-Bridge are;

- IRFZ44N (N-channel MOSFET)
- IRF9540 (P-channel MOSFET)
- 2222N (NPN BJT)



Figure: H-Bridge Driver

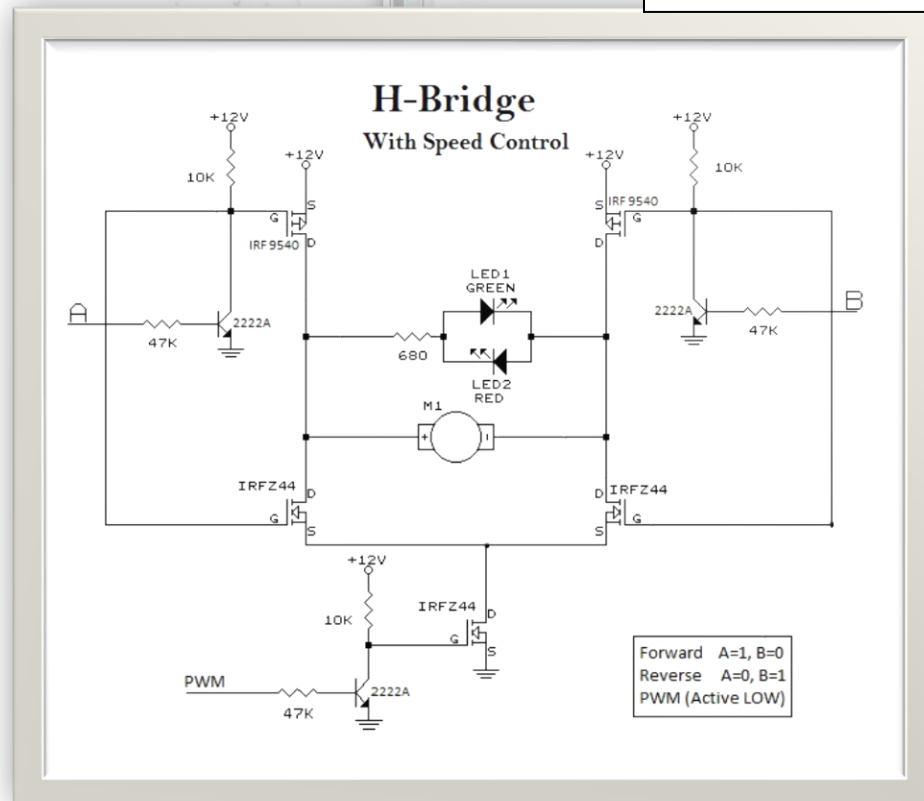


Figure 9: H-Bridge Circuit Diagram

Encoder

To implement a PID, the first and foremost requirement is to have some feedback to measure the output of the controller. In case of a DC motor, the feedback is provided by an optical encoder, also known as a “*Quadrature encoder*”. They employ two outputs called A & B which are called Quadrature outputs, as they are 90 degrees out of phase. These signals are decoded to produce a count up pulse or a

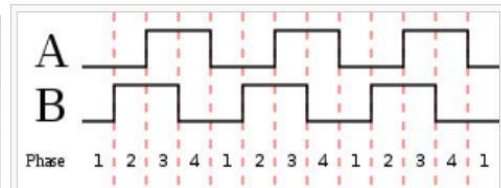


Figure: Quadrature Encoder Output

count down pulse. For decoding in software, the A & B outputs are read by software via an interrupt on any edge. This gives the speed and direction information of the motor.

Discrete Implementation

These days, the digital world has replaced the analogue technology. Normally we use microcontrollers in our control applications. The reason why it is called discrete implementation is that we take the samples of the motor speed or position after a fixed time interval and use this information for our algorithm.

The block diagram of the DC Motor PID system is;

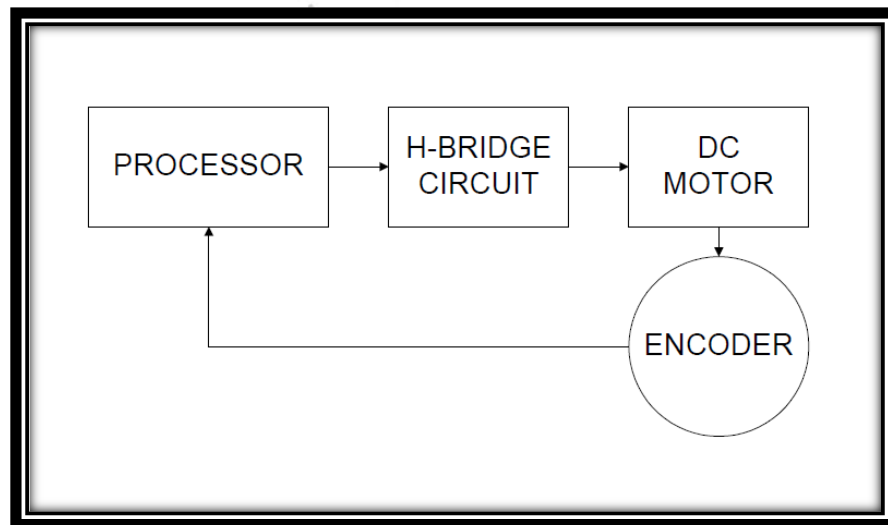


Figure 10: DC Motor PID Block Diagram

The devices used for this testing were;

- ✓ 89S52 Microcontroller
- ✓ H-Bridge using IRF9540 and IRF510
- ✓ Pittman motor with an encoder having a resolution of 500 counts per revolution (CPR)



Figure: DC Motor with Encoder

Variables Involved

Error term: The error term is found by subtracting the feedback (motor speed) from the set point (set speed). This is the error in terms of a number of encoder counts per unit time.

P –Proportional Term: The error term is multiplied by the proportional gain K_p which gives the proportional term. This term provides linear response to the system.

I –Integral Term: The error term added to the accumulated error and an integral gain K_i is multiplied with it which gives the integral term. This term provides response to accumulated errors.

D –Derivative Term: The derivative gain K_d is multiplied with the difference between the current and the previous error. This provides the response to change in error from one PID cycle to the other.

Pseudo Code

The pseudo code for this implementation is as follows;

```
previous_error = 0
integral = 0
start:
    error = setpoint - actual_position
    integral = integral + (error*dt)
    derivative = (error - previous_error)/dt
    output = output+ (Kp*error) + (Ki*integral) + (Kd*derivative)
    previous_error = error
    wait(dt)
    goto start
```

Pitfalls of PID

Integral Windup:

This problem occurs when the system faces a large change in input or the error signal. The integral term accumulates the error in every PID cycles during this transition when the error is comparatively large than steady state and grows too large that it creates oscillations in the system. Therefore the integral term must be limited to certain maximum and minimum range.

PWM Overflow Prevention:

During the time of transition when error is large, the output term may exceed the maximum limit of the PWM duty cycle (which is actually the system's maximum gain). Therefore care should be taken to limit the output within the range of PWM.

Tuning of the controller (Brute force approach)

The process of setting the optimal gains for P, I and D to get an ideal response from a control system is called *tuning*. Usually the behavior of a system is measured by system's step response. There are different methods of tuning. I used the brute force approach.

It involves the following steps;

- Add code to monitor the output of the PID algorithm (i.e. encoder speed feedback, counts per PID)
- Store the feedback speed value into an array element for the first 40 PID executions.
- Change the set speed from 0 to 50% of the motor's maximum speed. (16 counts per PID) This is equivalent to a step function.
- After 40 PID executions stop the motor and print the array data to the serial port.
- Now the results of all PID values within the test range are plotted with respect to time.
- The values which yield the best curve will be used for the PID controller.

Brute force Tuning Code

The tuning algorithm loops through a range of values for each coefficient P, I, and D. For example:

```
for (P=P_start; P<P_end;P++)  
{
```

```
For (I=I_start; I<I_end; I++)  
{  
    For (D=D_start; D<D_end; D++)  
    {  
        Set motor speed to 16 counts/PID  
        Wait  
        Print the P, I, and D values and the 40 array elements  
    }  
}  
}
```

MULTIPLE PROCESSORS COMMUNICATION

Overview

The patient side mainly consists of a robotic arm and a PC. The PC receives the commands from the surgeon side through internet and after necessary computation; it transmits this data through serial port to the controller of the robot. In the robot, this data is received by a master microcontroller. It analyzes it and then sends appropriate commands to the two slave microcontrollers which eventually run their respective motors to act upon the surgeon's command.

Why Multiple Processors??

The question arises that why do we need multi processors?

The reason is that generally a microcontroller has;

- Limited number of peripherals
 - 2 to 3 timers/counters

- 2 external interrupts
- Limited I/O pins
- Relatively less computational power

Considering these facts, one microcontroller cannot be sufficient to accurately control all the motors employed in the robot. Therefore we need divide the computational load over more than one controller. Moreover with more microcontrollers employed, we have more number of peripherals and I/O pins.

Ultimately there should be some communication protocol between these microcontrollers to perform appropriately.

Protocols

The two basic communication protocols which are most commonly used are;

Parallel Communication

- ✓ Physical layer is capable of transporting multiple bits of data at a time.
- ✓ Multiple data, control, and possibly power wires
- ✓ One bit per wire
- ✓ High data throughput with short distances
- ✓ Typically used when connecting devices on same IC or same circuit board
- ✓ Bus must be kept short, long parallel wires result in high capacitance values which requires more time to charge/discharge
- ✓ Data misalignment between wires increases as length increases

Serial Communication

- ✓ Physical layer transports one bit of data at a time.
- ✓ Single data wire, possibly also control and power wires

- ✓ Higher data throughput with long distances
- ✓ Less average capacitance, so more bits per unit of time
- ✓ Cheaper, less bulky.
- ✓ More complex interfacing logic and communication protocol.
 - Sender needs to decompose word into bits
 - Receiver needs to recompose bits into word
 - Control signals often sent on same wire as data increasing protocol complexity

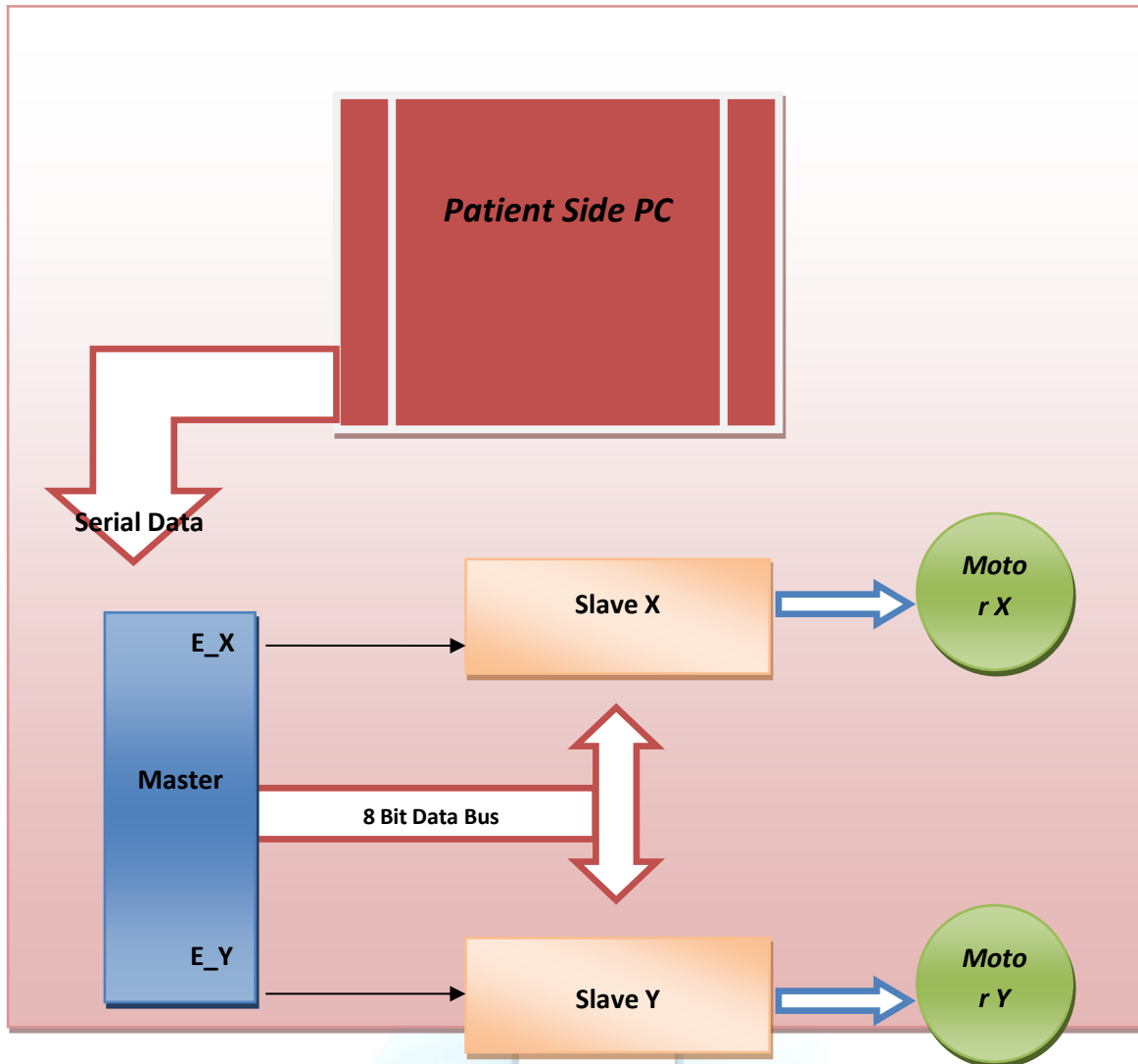
Approach Followed (Parallel Communication)

In this system, currently I am employing one Master and two slave microcontrollers. The serial data from PC will be received by the master and transmitted to slaves over a parallel data bus 8 bits wide.

The driving factors for parallel approach are;

- It minimizes the delays in communication between controllers.
- No constraint of number of wires at the same circuit board
- Provides high data throughput
- Simpler communication protocol for sender and receiver.

Block Diagram of the Master-Slave Configuration



How it works?

When the master controller wants to send the data to any of the slaves, it puts the data at the data bus and then activates the respective enable signal of the slave. The slave is configured to read the data bus at the falling edge of the enable signal which is connected to one of its external interrupt pin. After reading the data, the slave starts processing it and if the master wants to send new data, it needs to give another falling edge at the enable signal of the slave.

Similar process happens when the master wants both the slaves to read the data from the data bus. The master just needs to give a falling edge at both the enable signals and both the slaves read the data.

This process is also shown in the following timing diagram;

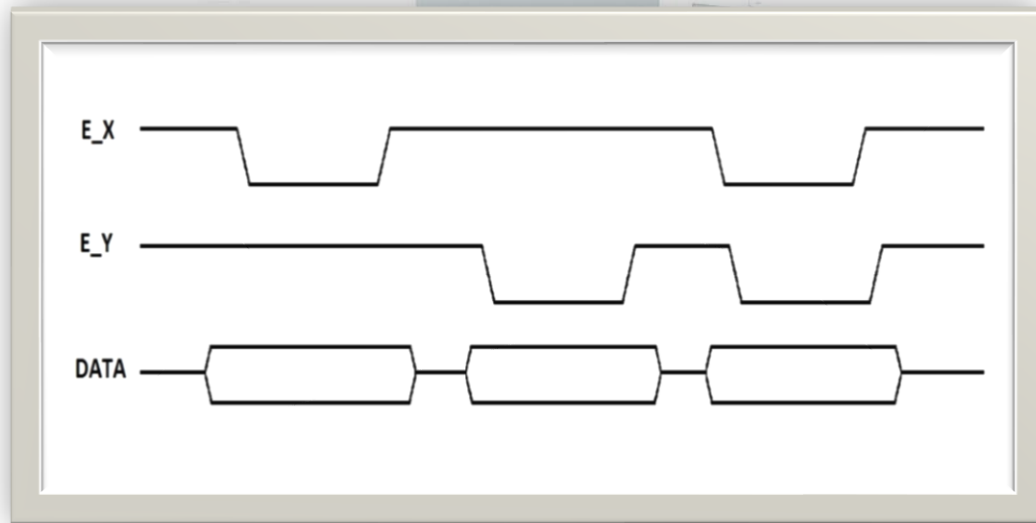


Figure 11: Timing Diagram

In the above diagram, first the master lowers the E_X signal which means that it wants slave X to read the data. Then the new data is placed at data bus and E_Y is activated, thus slave Y reads the data. Then both the enables are activated therefore both the slaves will pick up the data.

Prototype Testing

To test this Master-Slave configuration using parallel communication protocol, I developed a prototype robot with two motors.

The robot was to be controlled with a joystick connected to a PC (equivalent to surgeon PC) whereas the robot itself was connected to another PC (patient side). These two PCs were connected via internet.



In this section we'll mainly focus the microcontrollers' part at the patient side. As mentioned earlier, one master controlled two slaves and the two motors were controlled by two slave microcontrollers.

- + 89S52 microcontrollers were used.
- + Master controller was configured to receive data from PC via serial port at a baud rate of 4800.
- + Port 2 of all the microcontrollers were connected together as a data bus
- + External interrupt 1 of both slaves was acting as enable configured to generate an interrupt at the falling edge.
- + Pin 1.0 & 1.1 of the master controller were connected to the enables of slave X and slave Y respectively.



IMPLEMENTATION ON TELE-SURGICAL ROBOT

Design Details

The prototype tele-surgical robot which we have developed has four DOF. The details are as follows;

Shoulder joint Motion

This motion causes the structure to rotate in YZ plane and gives a range of 90 around degree angle to the surgical tool.

Elbow joint Motion

This motion causes the structure to rotate in XZ plane and gives a range of around 90 degree angle to the surgical tool.

Tool translational motion

It provides the flexibility to linearly move the tool forward & backward which is utilized in insertion of tool inside the body through the key-hole and further manipulation of the organs.

In this prototype robot, this motion is very important for Peg-Transfer Exercise.

Tool Gripper motion

This motion helps the jaw of the surgical tool to open and close to release and grab the pegs.

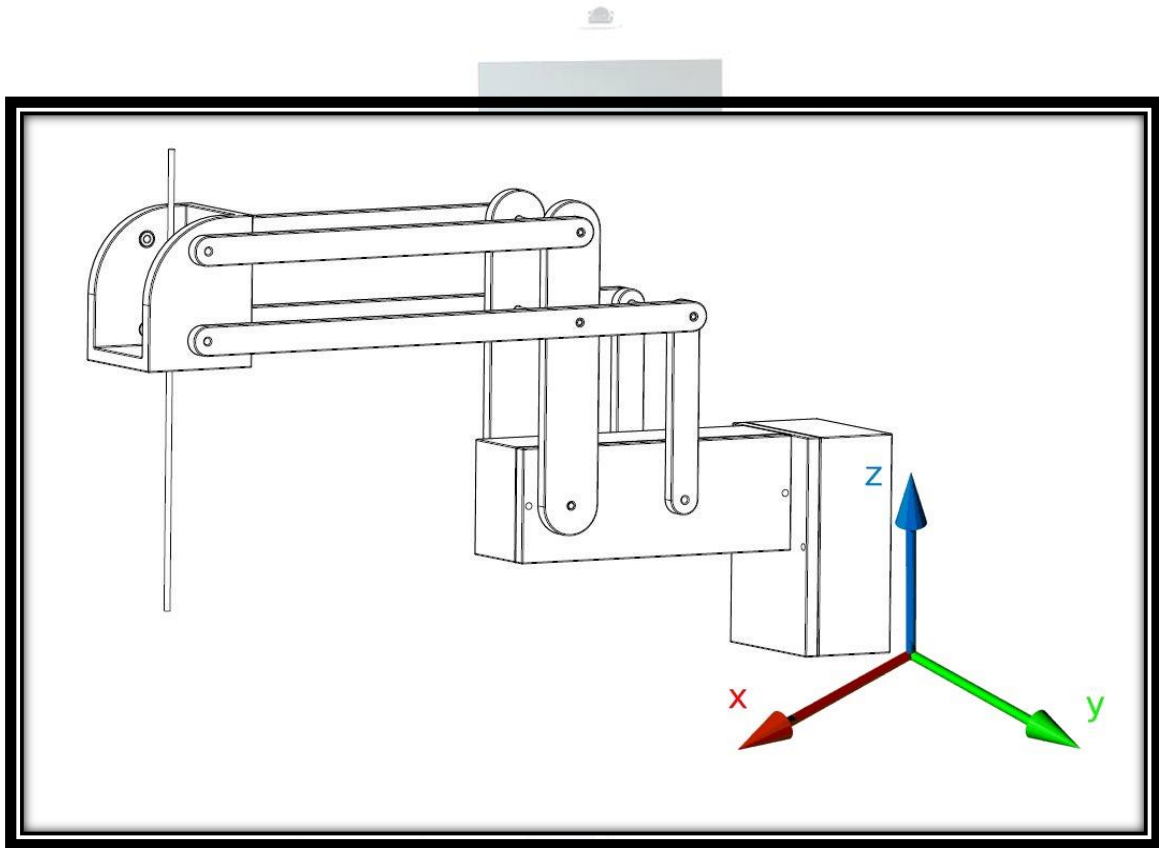
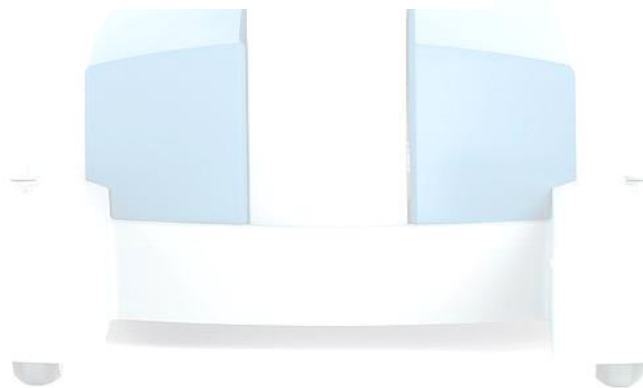
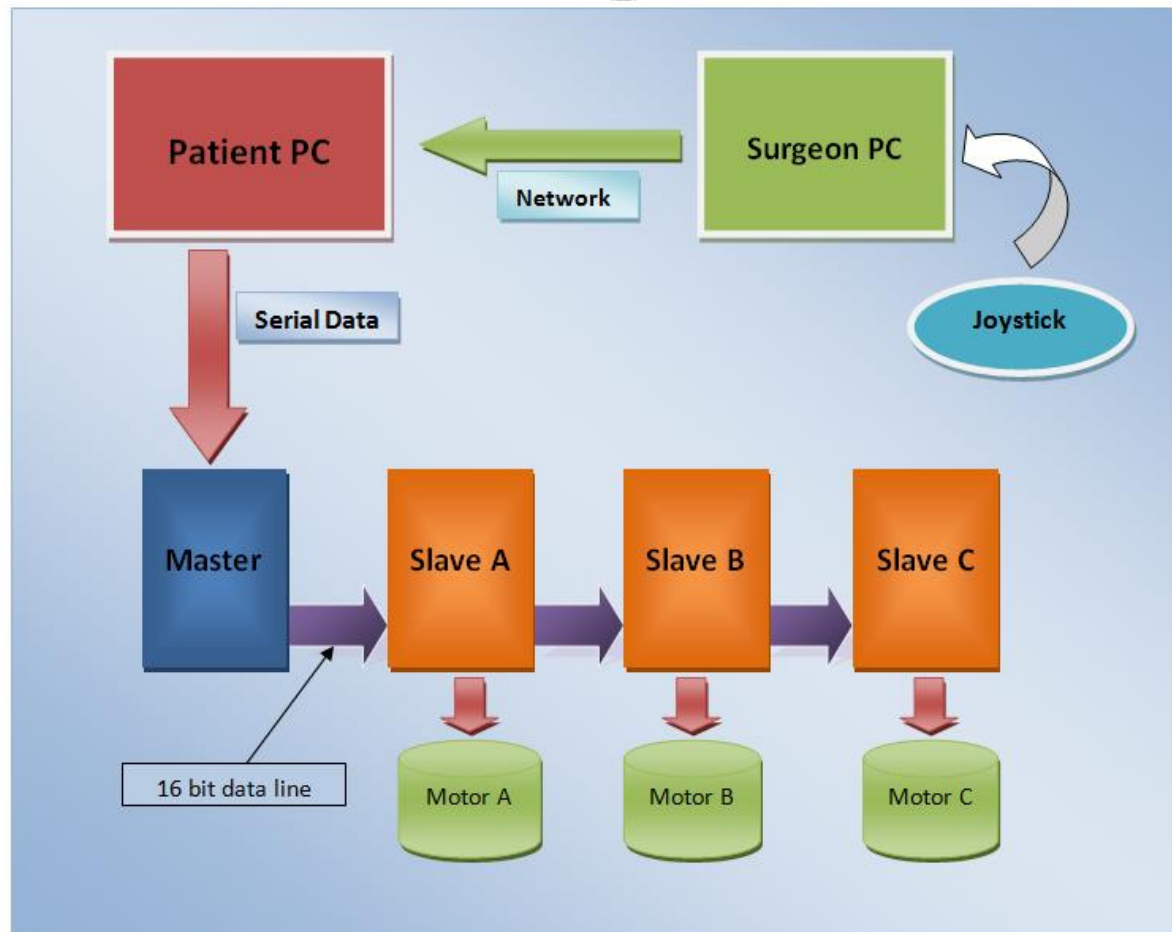


Figure 12: Mechanical Design



Block Diagram of the actual robotic system

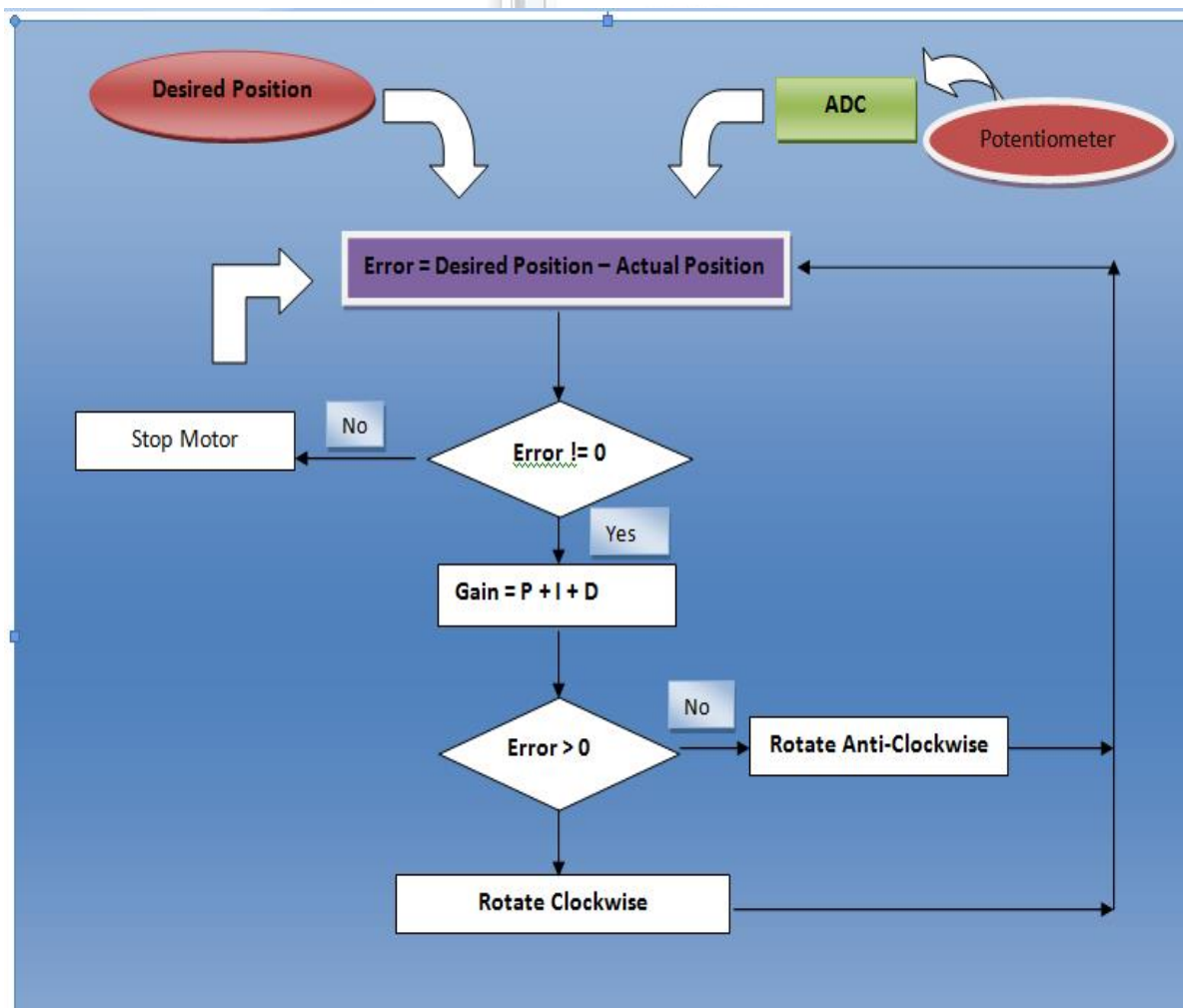


The same methodology has been used in the actual system as there was used in the prototype robotic system for the multiple processor communications, the only difference is that there are three slaves now and a 16 bit data line. Slave A & B control one motor each for the first two rotations, whereas the slave C controls the Tool Translational & Tool Gripper motion.

Implementation of Position Feedback

To control the position of the robotic arm, the Position PID control is implemented in the system.

The Position PID implementation for the shoulder joint is shown in the following diagram;



A potentiometer actually gives the feedback of the position of the robotic arm. For the whole range of the motion of this axis, the potentiometer gives an analogue value which is given to the controller through an ADC. The desired position of the arm is received by the Master controller and then the PID Position control algorithm is applied to achieve this desired position.

For other axis, there is a position encoder being used to give the feedback. The rest of the mechanism is same.

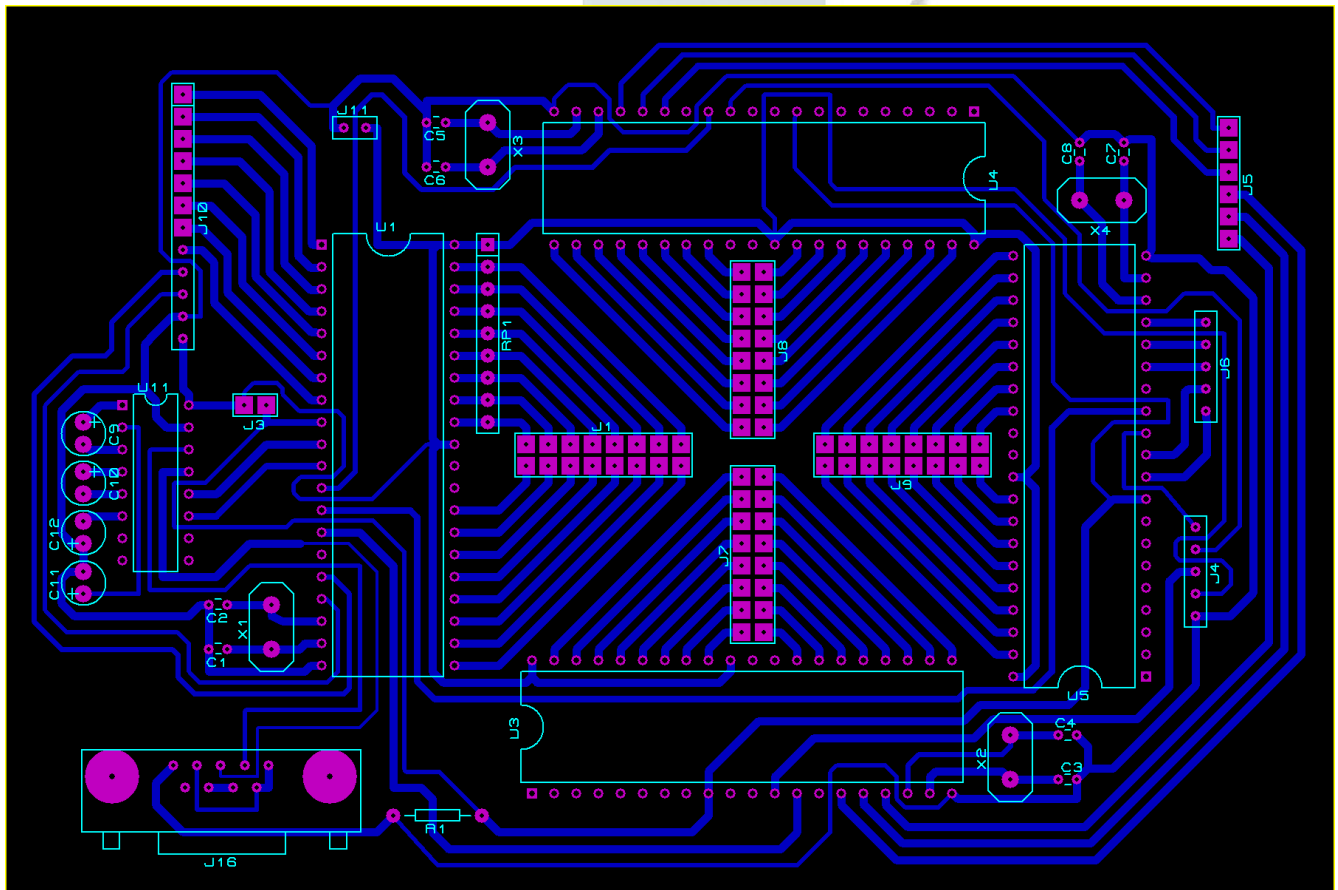
Layouts

The major circuit layouts involved in the system are;

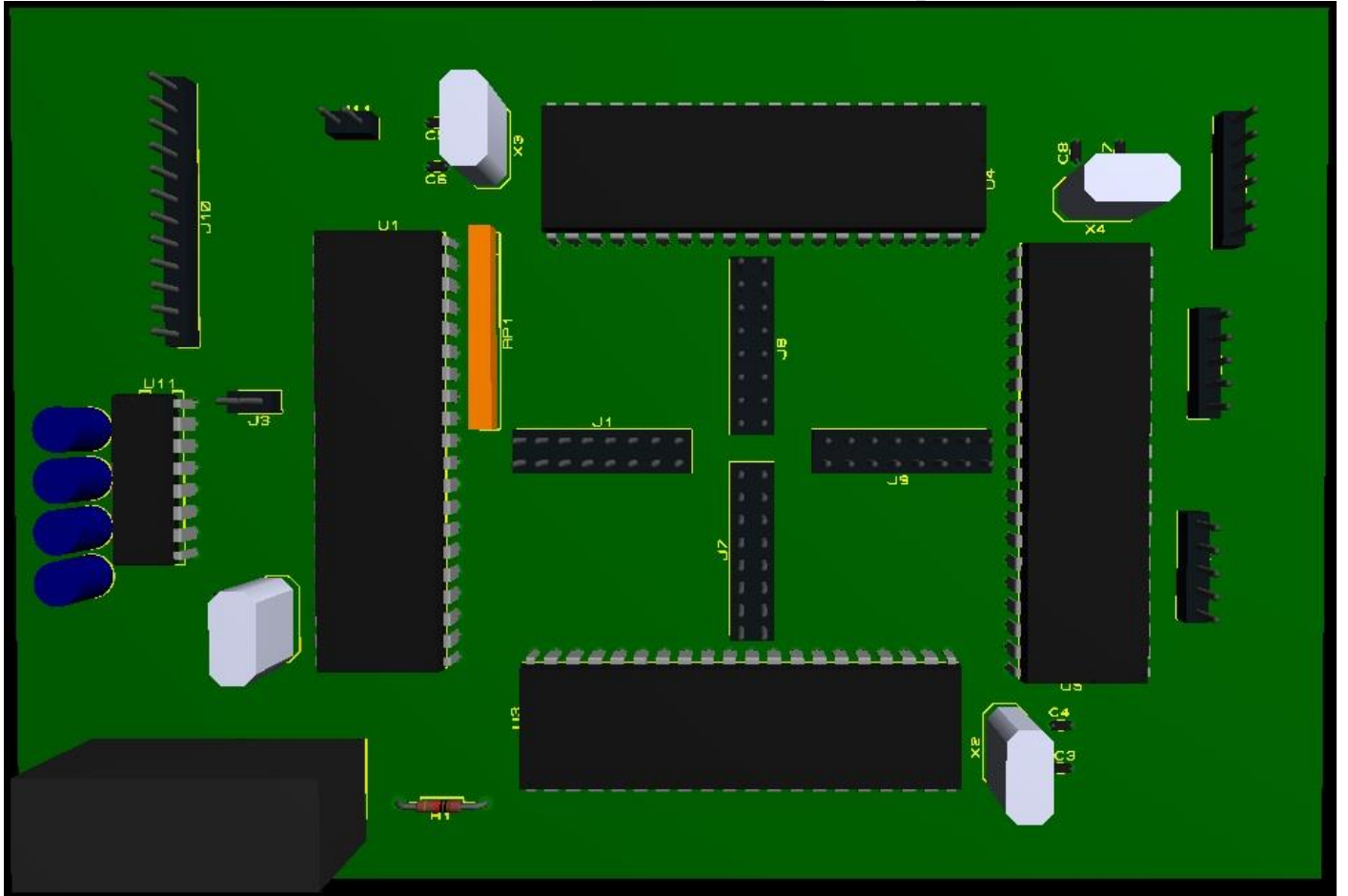
- Controller board
- H-Bridge Driver (using L298)
- Power Distribution Board
- ADC Module

Controller board

Layout:

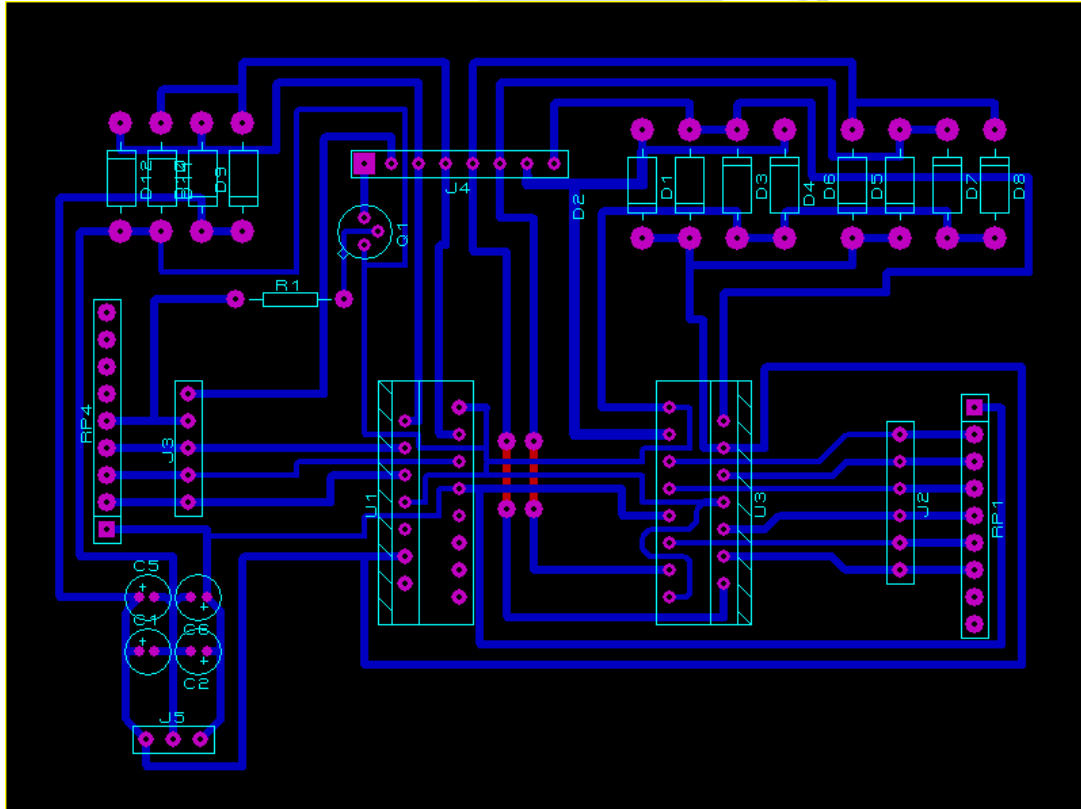


3D Visualization

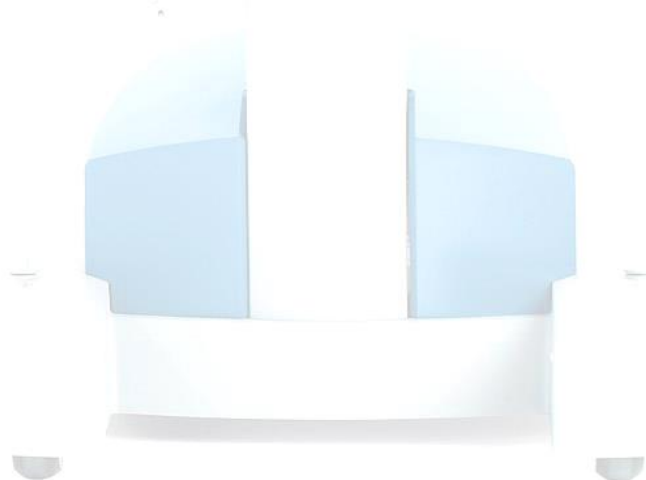
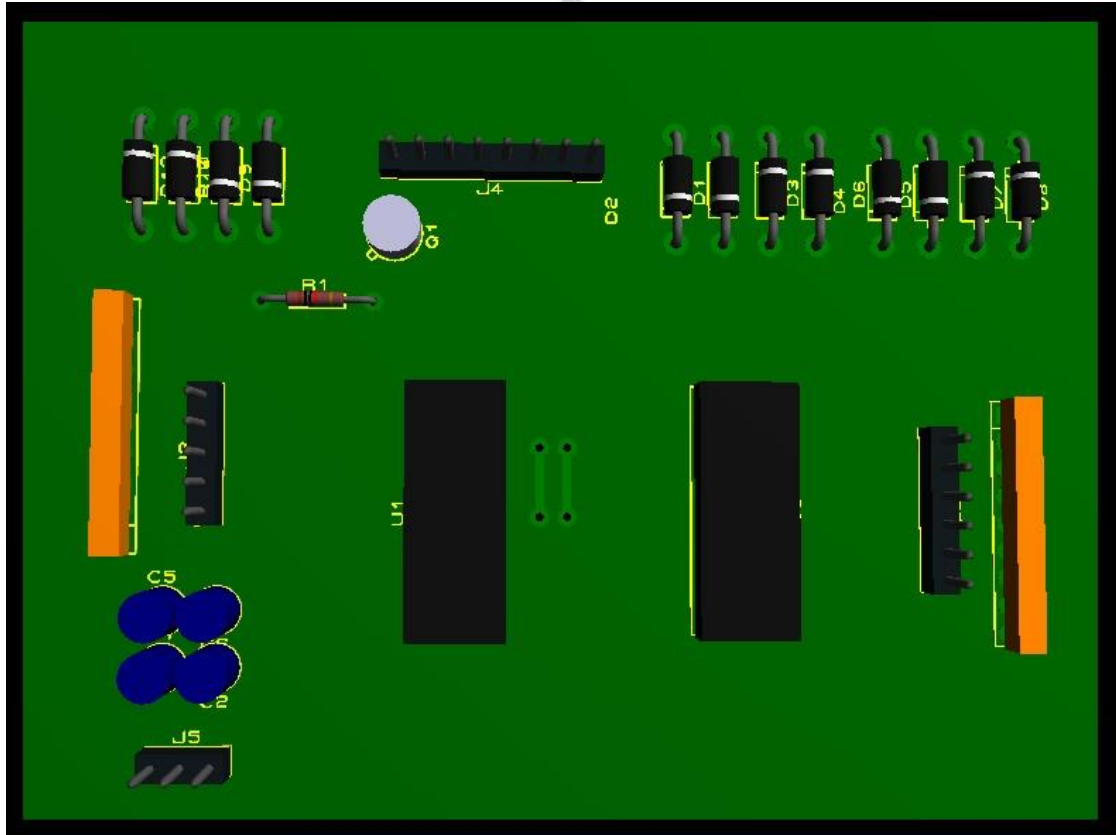


H-Bridge Driver

Layout:

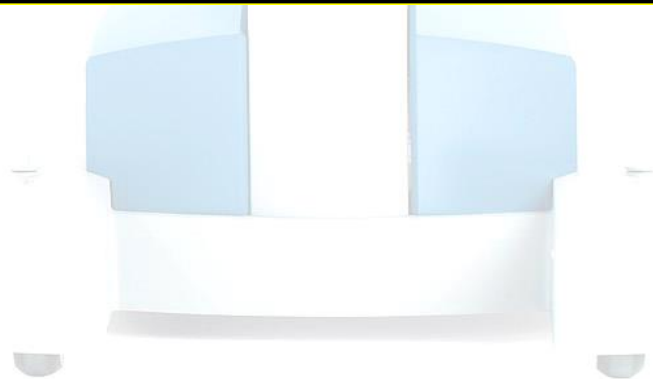
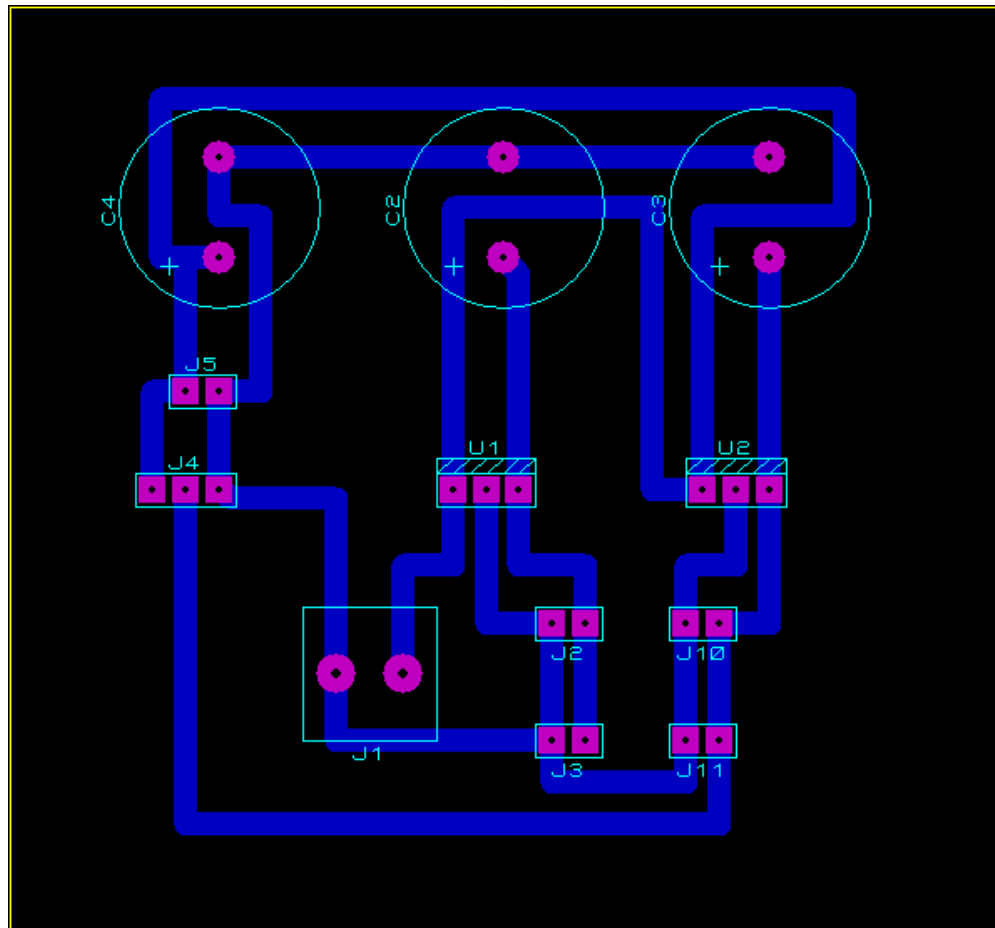


3D Visualization

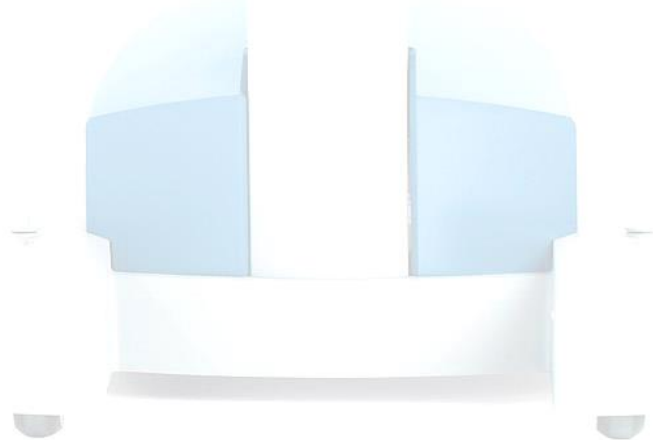
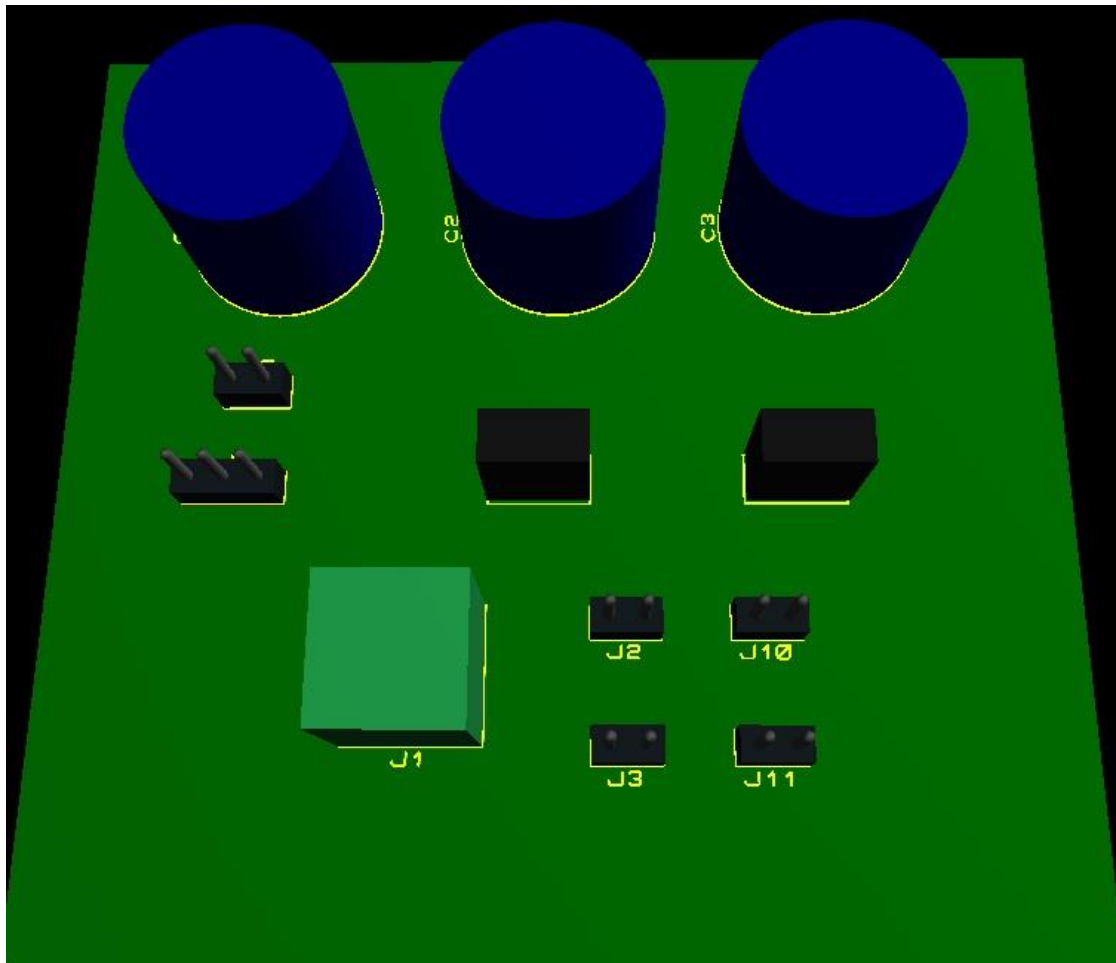


Power Distribution Board

Layout:

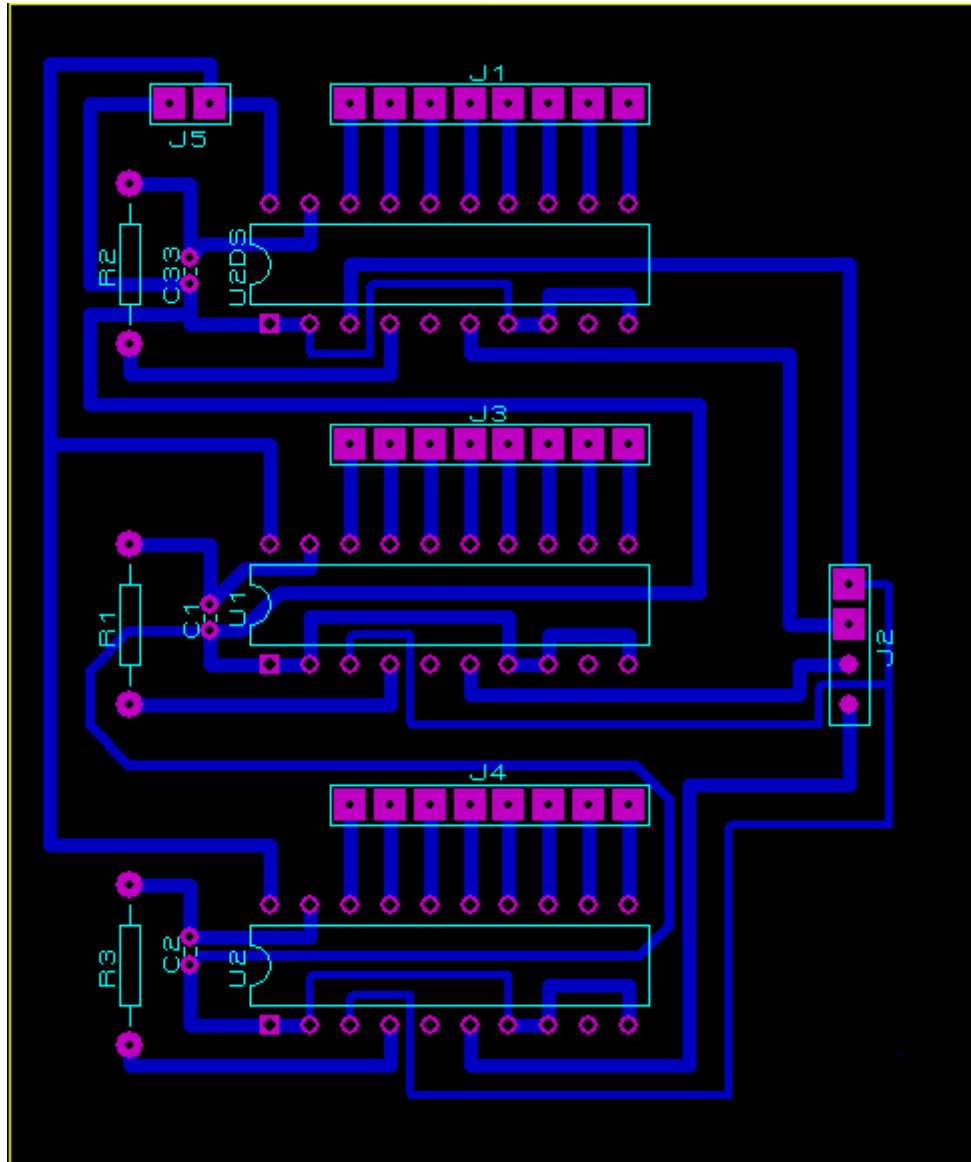


3D Visualization

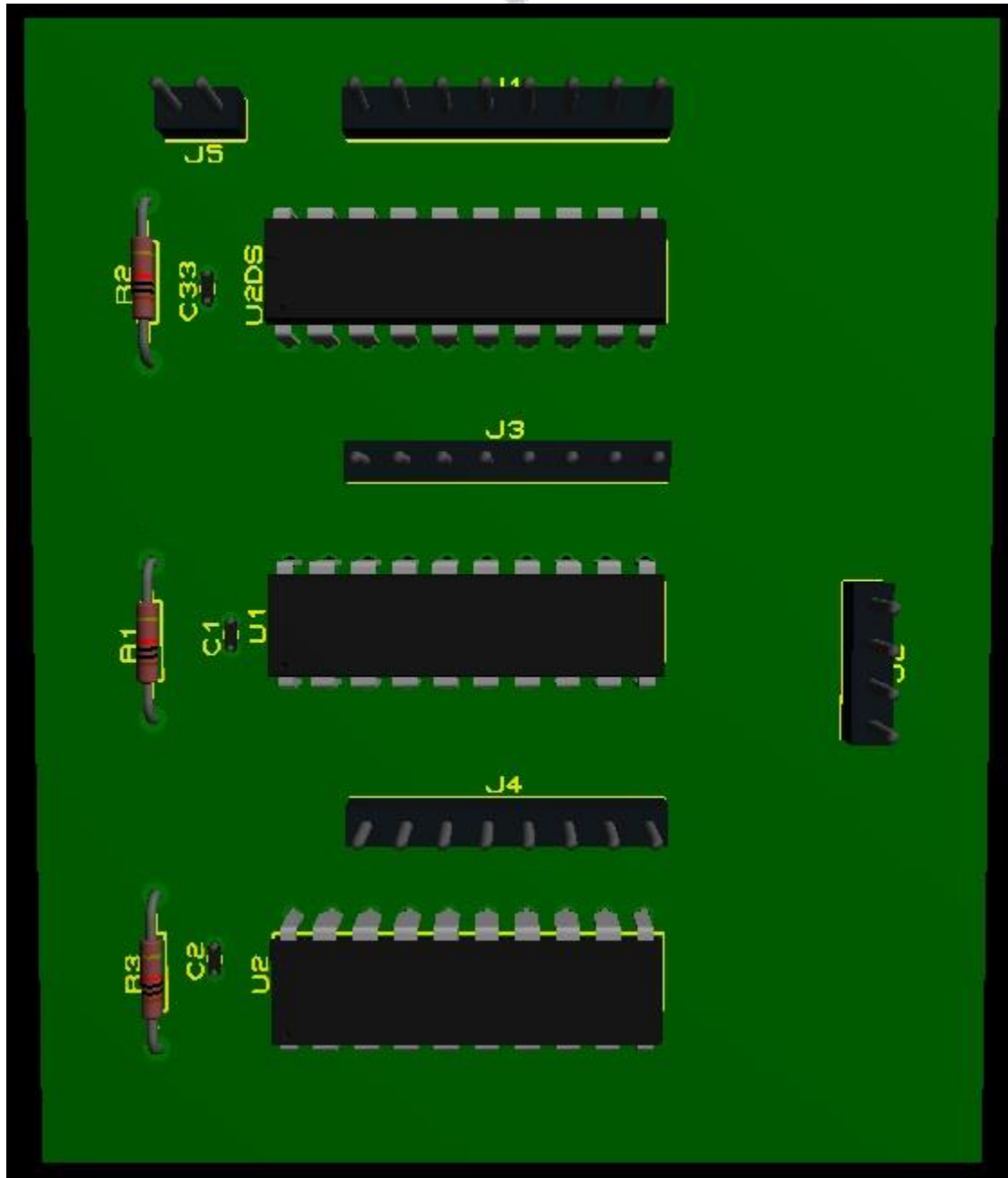


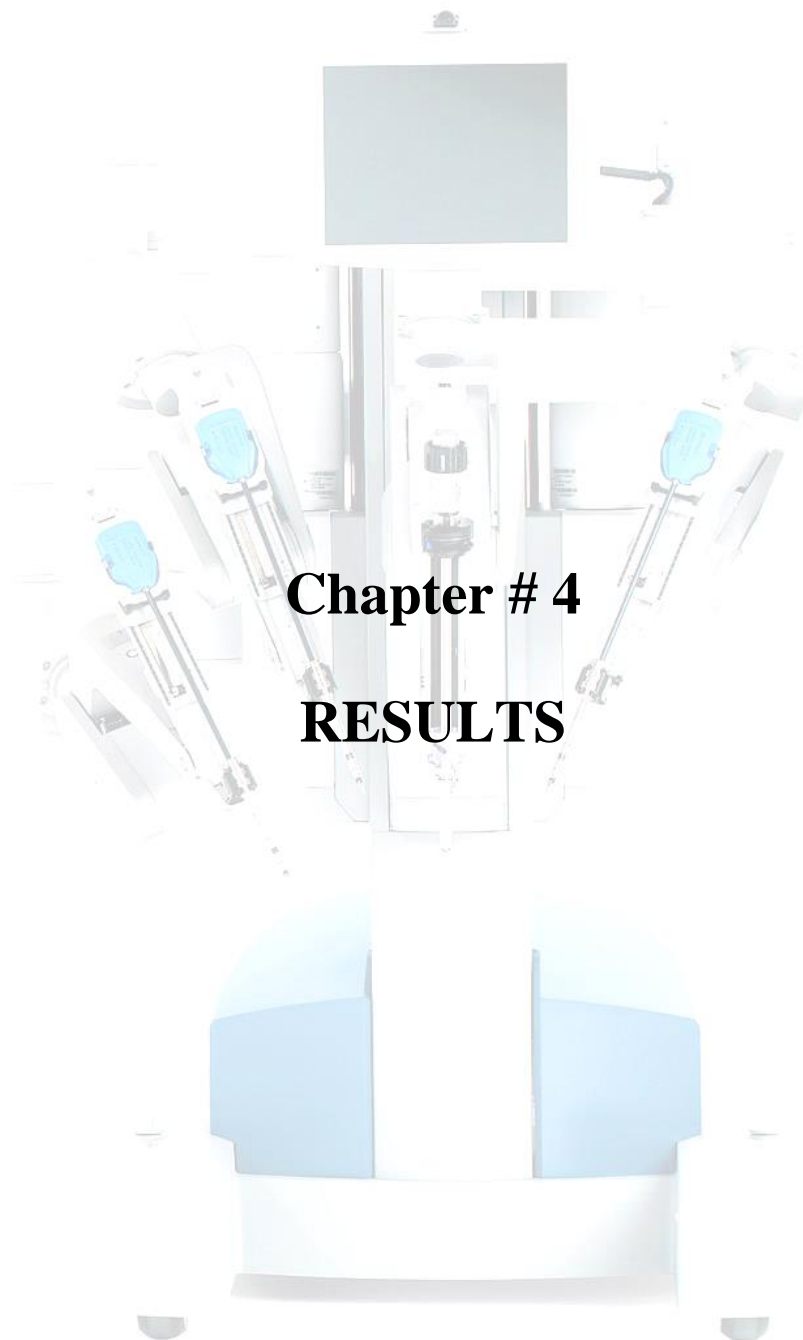
ADC Module

Layout:



3D Visualization



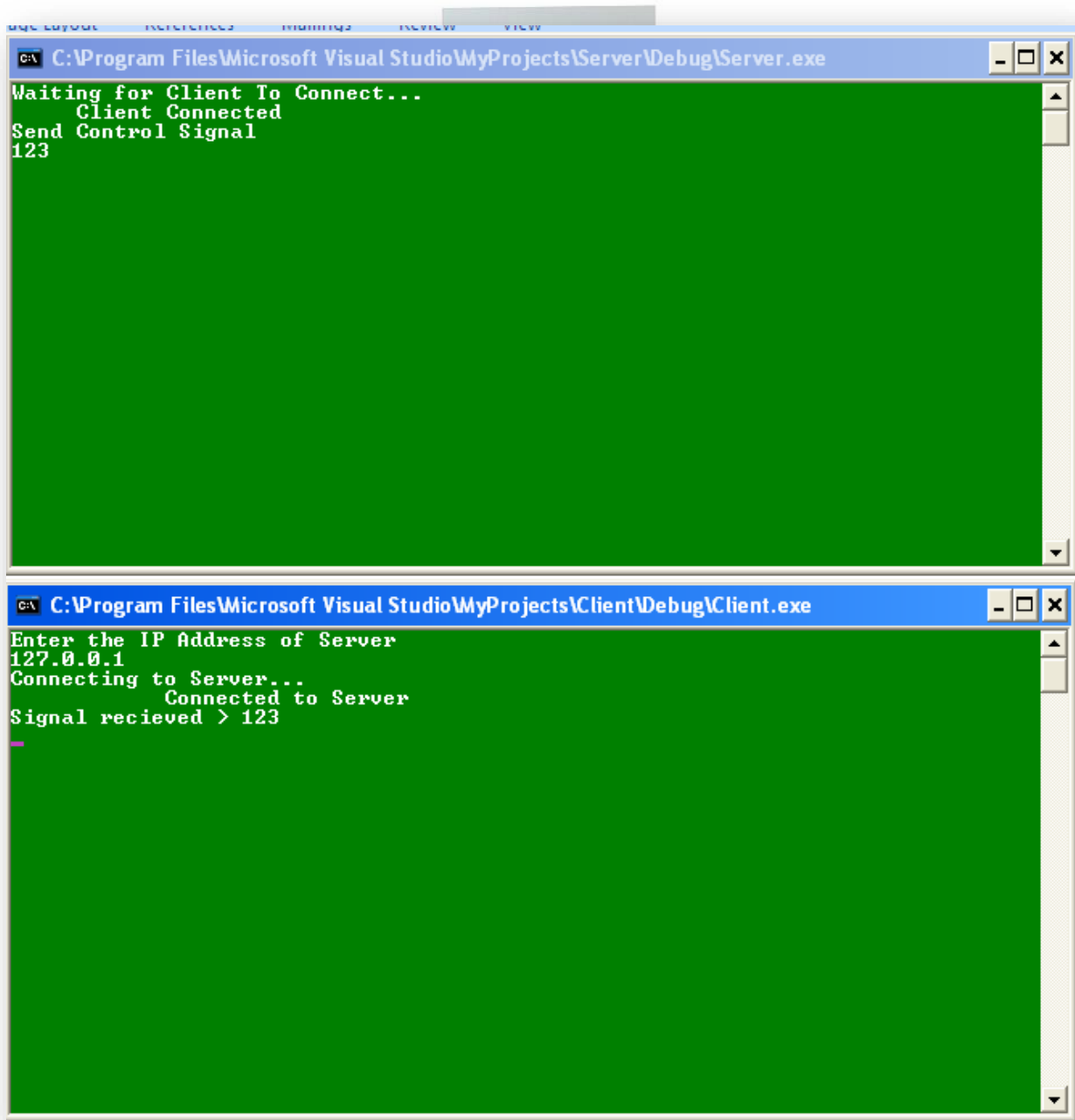


Chapter # 4

RESULTS

Network Communication

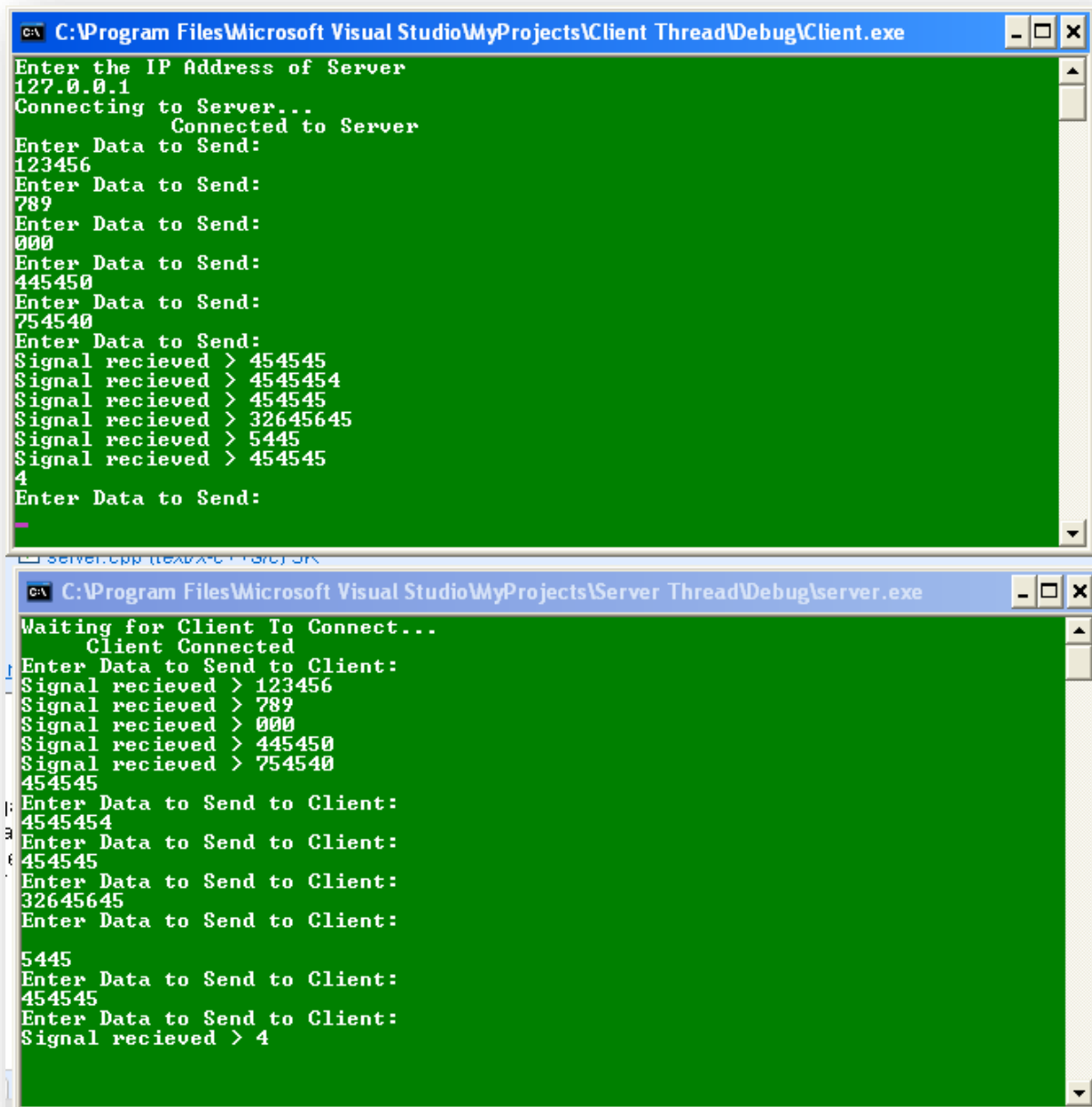
Screen Shots below shows the output of first TCP code:



The image displays two screenshots of the Visual Studio debug console, illustrating the execution of a TCP communication program. The top window, titled "C:\Program Files\Microsoft Visual Studio\MyProjects\Server\Debug\Server.exe", shows the server's output: "Waiting for Client To Connect...", "Client Connected", "Send Control Signal", and "123". The bottom window, titled "C:\Program Files\Microsoft Visual Studio\MyProjects\Client\Debug\Client.exe", shows the client's output: "Enter the IP Address of Server", "127.0.0.1", "Connecting to Server...", "Connected to Server", and "Signal recieved > 123". Both windows have a green background and a blue title bar.

```
C:\Program Files\Microsoft Visual Studio\MyProjects\Server\Debug\Server.exe
Waiting for Client To Connect...
Client Connected
Send Control Signal
123

C:\Program Files\Microsoft Visual Studio\MyProjects\Client\Debug\Client.exe
Enter the IP Address of Server
127.0.0.1
Connecting to Server...
Connected to Server
Signal recieved > 123
```



```
C:\Program Files\Microsoft Visual Studio\MyProjects\Client Thread\Debug\Client.exe
Enter the IP Address of Server
127.0.0.1
Connecting to Server...
Connected to Server
Enter Data to Send:
123456
Enter Data to Send:
789
Enter Data to Send:
000
Enter Data to Send:
445450
Enter Data to Send:
754540
Enter Data to Send:
Signal recieved > 454545
Signal recieved > 4545454
Signal recieved > 454545
Signal recieved > 32645645
Signal recieved > 5445
Signal recieved > 454545
4
Enter Data to Send:

C:\Program Files\Microsoft Visual Studio\MyProjects\Server Thread\Debug\server.exe
Waiting for Client To Connect...
Client Connected
Enter Data to Send to Client:
Signal recieved > 123456
Signal recieved > 789
Signal recieved > 000
Signal recieved > 445450
Signal recieved > 754540
454545
Enter Data to Send to Client:
4545454
Enter Data to Send to Client:
454545
Enter Data to Send to Client:
32645645
Enter Data to Send to Client:
5445
Enter Data to Send to Client:
454545
Enter Data to Send to Client:
Signal recieved > 4
```

Screen Shot below shows output of 2nd TCP code:

Data rate and Delay:

Datarate came out to be 18 Mbps when experiments were performed on commercial internet. The distance between two PCs was 5m.

When we sent packets to a laptop running wireless internet on it, data rate came out to be 5 Mbps.

RTT came out to be 90ms.

While using UDP, delay came out to be 5ms.

Serial Communication

- | | |
|--------------|-------|
| 1) Baud Rate | 38400 |
| 2) Data bits | 8 |
| 3) Parity | 0 |
| 4) Stop bit | 1 |

JOYSTICK INTERFACING

| Axis | Data Type | Range |
|------|-----------|---------|
| X | Int | 0-65535 |
| Y | Int | 0-65535 |
| Z | Int | 0-65535 |

VIDEO STREAMING

Camera: 5 Mega pixels

Frame rate: 15 frames/s

Bytes in each frame: 921600

Packet size: 20,000

Channels: 3

GUI



PID CONTROLLER

Sample PID Data

Here are a few samples of the PID data I obtained through serial port during the tuning process;

Table 3: Sample PID Data

| | | | | | |
|-----------------------------|---|------|-------|------|------|
| <i>K_d</i> | 0 | 0 | 1.00 | 2.00 | 2.00 |
| <i>K_p</i> | 0 | 0.04 | 00.02 | 0.01 | 0.04 |
| <i>K_i</i> | 0 | 3.00 | 03.00 | 3.00 | 3.00 |
| <i>Speed</i> | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 2 | 2 | 2 |
| | 2 | 5 | 5 | 5 | 5 |
| | 4 | 8 | 8 | 8 | 8 |
| | 6 | 10 | 10 | 11 | 11 |
| | 8 | 13 | 13 | 13 | 12 |
| | 9 | 15 | 15 | 15 | 16 |

| | | | | | |
|--|----|----|----|----|----|
| | 10 | 17 | 17 | 17 | 17 |
| | 16 | 19 | 20 | 20 | 20 |
| | 17 | 22 | 21 | 21 | 22 |
| | 17 | 22 | 22 | 21 | 22 |
| | 17 | 24 | 23 | 21 | 22 |
| | 18 | 23 | 22 | 21 | 22 |
| | 18 | 23 | 22 | 21 | 21 |
| | 19 | 23 | 21 | 20 | 21 |
| | 19 | 22 | 21 | 20 | 20 |
| | 19 | 22 | 21 | 20 | 21 |
| | 19 | 22 | 21 | 19 | 20 |
| | 19 | 21 | 20 | 19 | 19 |
| | 19 | 21 | 20 | 19 | 20 |
| | 20 | 21 | 20 | 18 | 19 |
| | 19 | 21 | 19 | 18 | 18 |
| | 20 | 20 | 20 | 18 | 19 |
| | 20 | 20 | 19 | 18 | 18 |
| | 20 | 20 | 18 | 17 | 18 |
| | 20 | 19 | 18 | 17 | 17 |
| | 20 | 19 | 18 | 17 | 17 |
| | 20 | 19 | 18 | 16 | 17 |
| | 21 | 18 | 17 | 17 | 17 |
| | 20 | 18 | 16 | 16 | 17 |
| | 20 | 17 | 17 | 15 | 16 |
| | 20 | 18 | 16 | 16 | 16 |
| | 20 | 16 | 15 | 15 | 15 |

PID Tuning Plots

After collecting the data for all the possible combinations of the three gains in the acceptable range, I plotted the data for graphical representation of numerical data of motor's speed. Around 150 data samples were plotted and the optimum gain values were determined.

In each plot, the K_d & K_i values are fixed and then K_p is varied from 0 to 3

Let's see the effects of different parameters.

Changing K_i

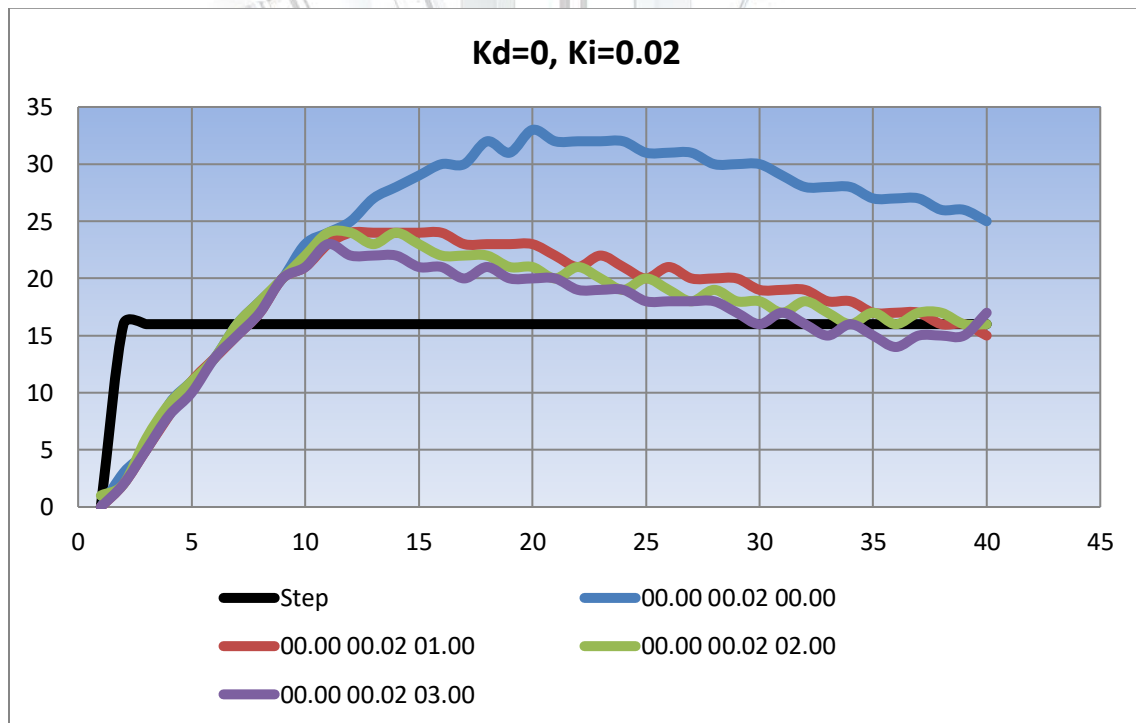


Figure 13: Response of motor with $K_d=0$; $K_i=0.02$

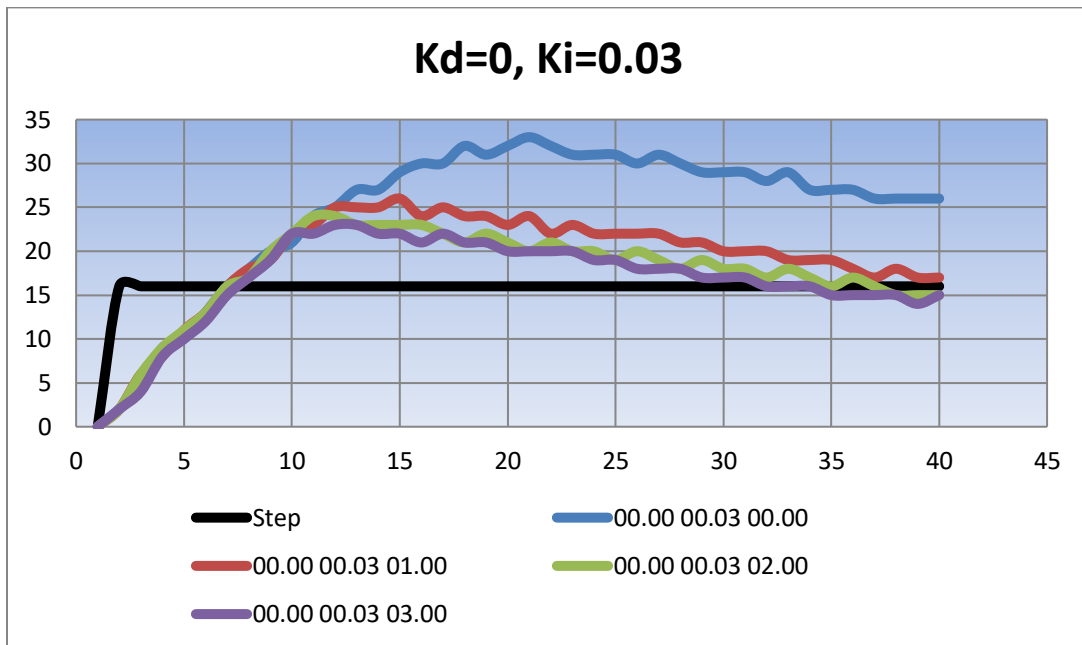


Figure 1411: Response of motor with $K_d=0$; $K_i=0.03$

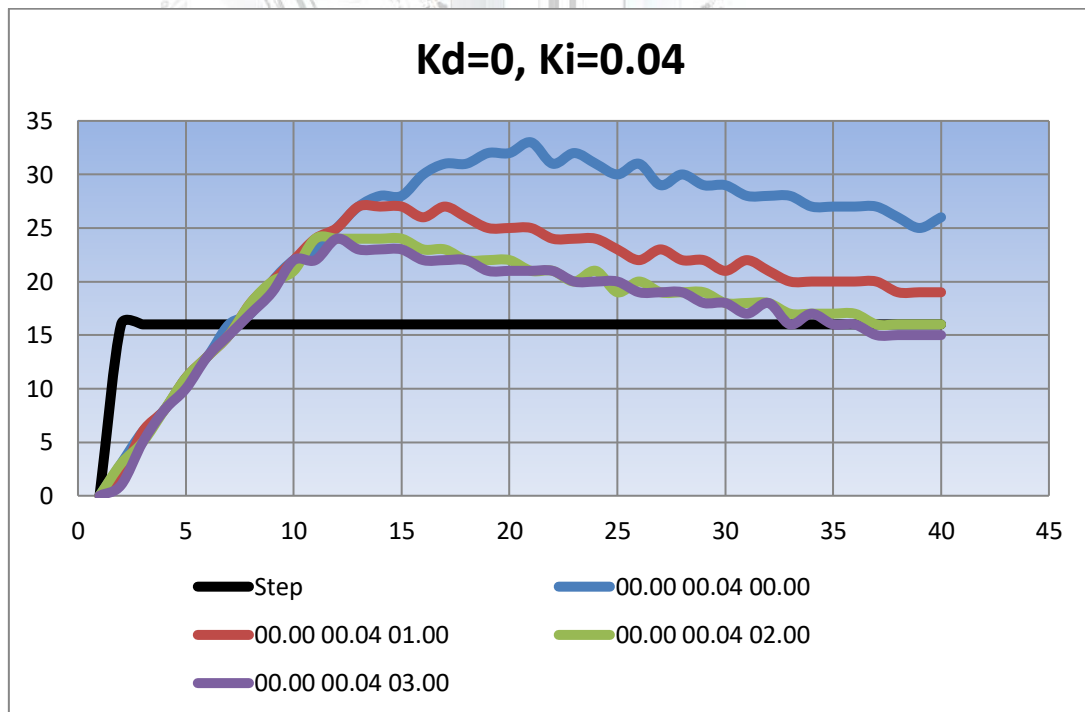


Figure15: Response of motor with $K_d=0$; $K_i=0.04$

It can be noticed that with increase in K_i , the overshoot increases but at the same time the steady state error also improves relatively.

Changing K_d

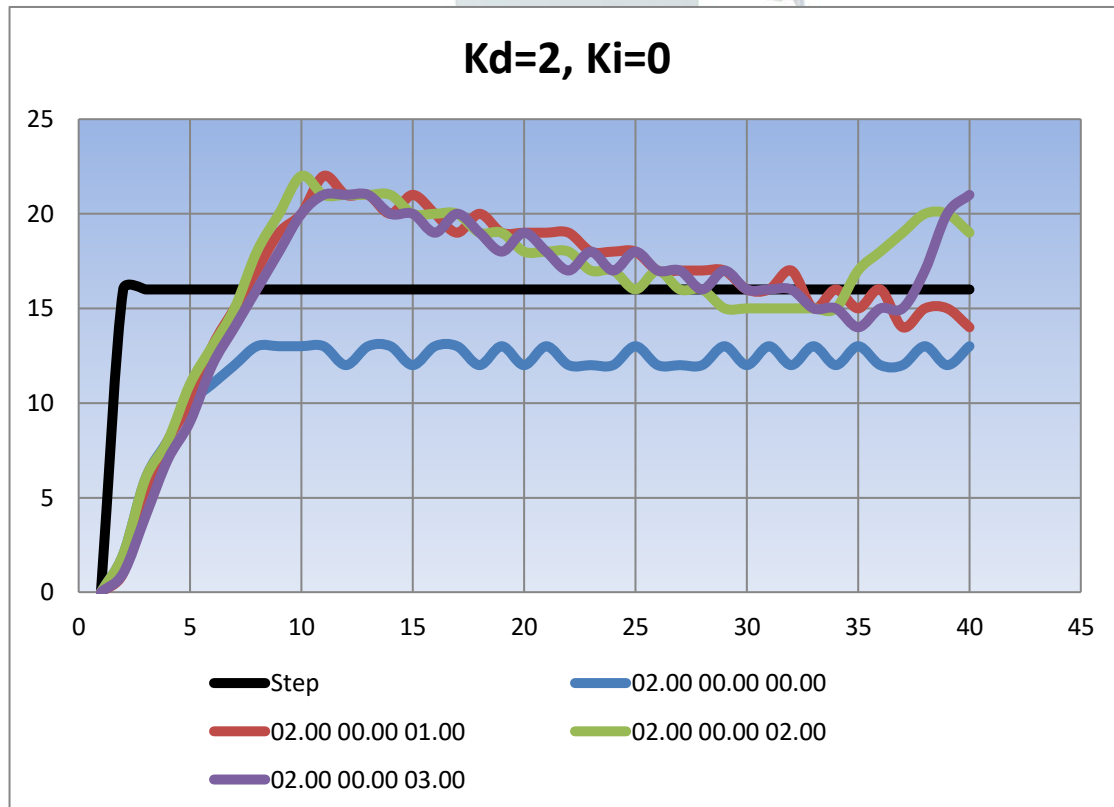


Figure16: Response of motor with $K_d=2; K_i=0$

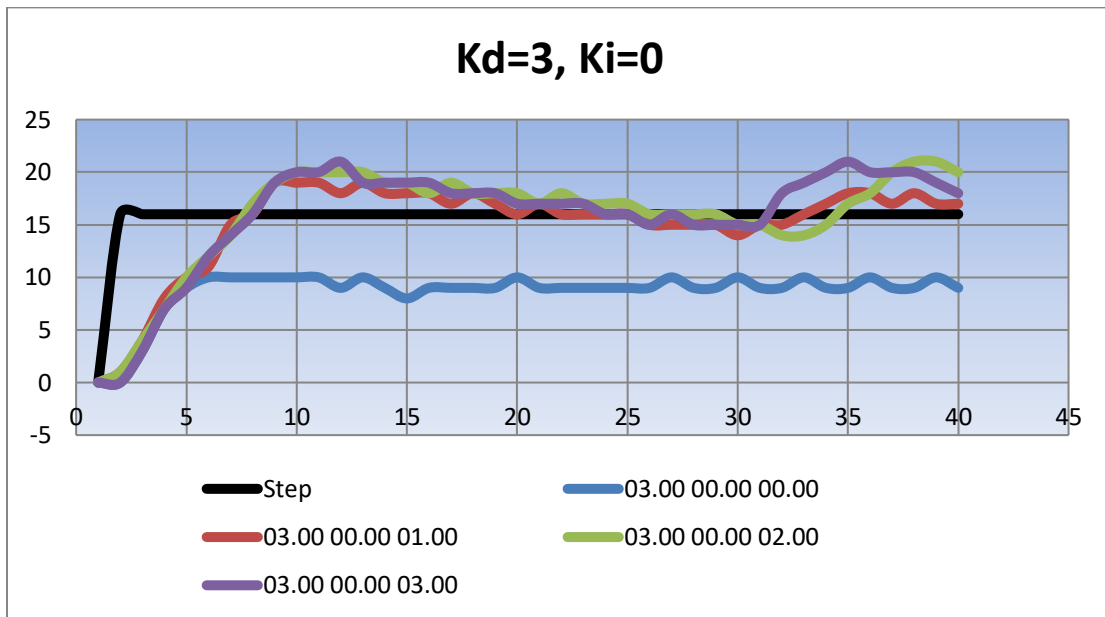


Figure 17: Response of motor with $K_d=3$; $K_i=0$

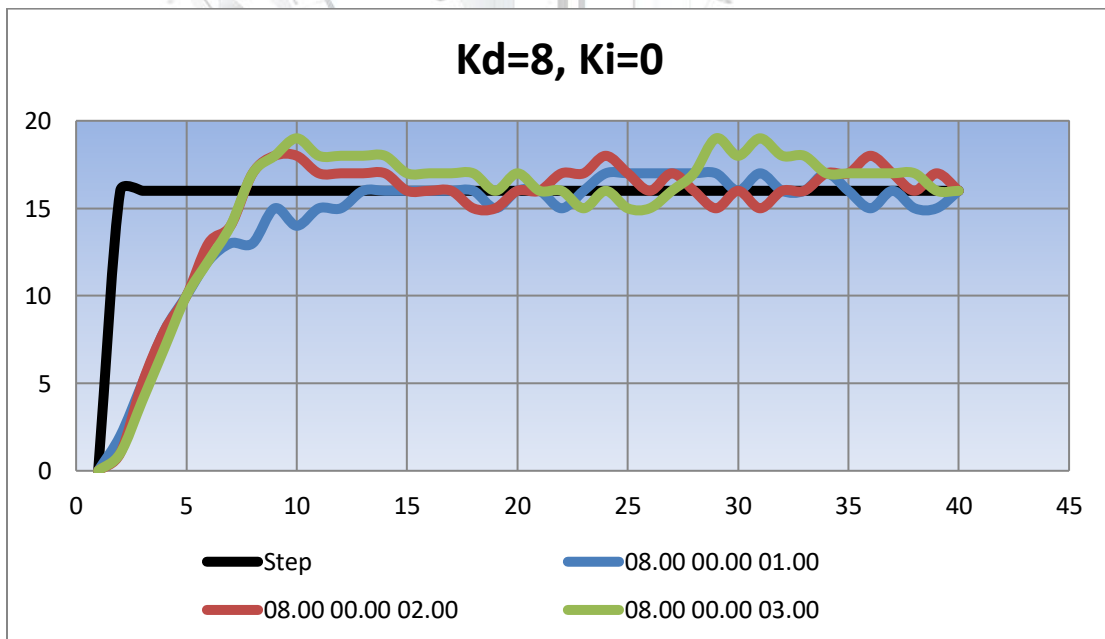


Figure 18: Response of motor with $K_d=8$; $K_i=0$

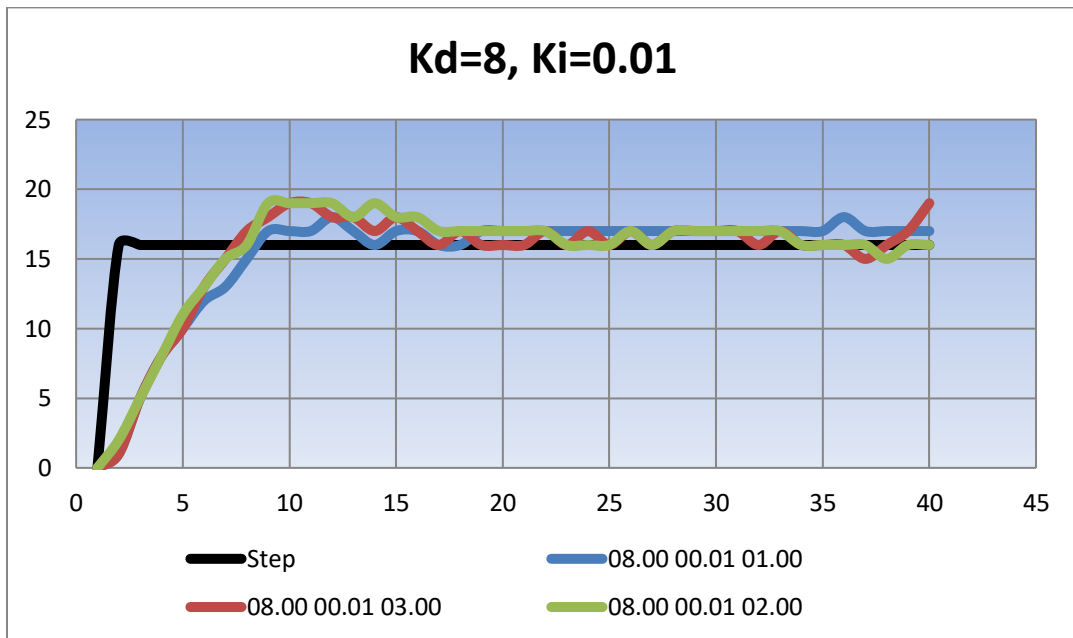


Figure 19: Response of motor with Kd=8; Ki=0.01

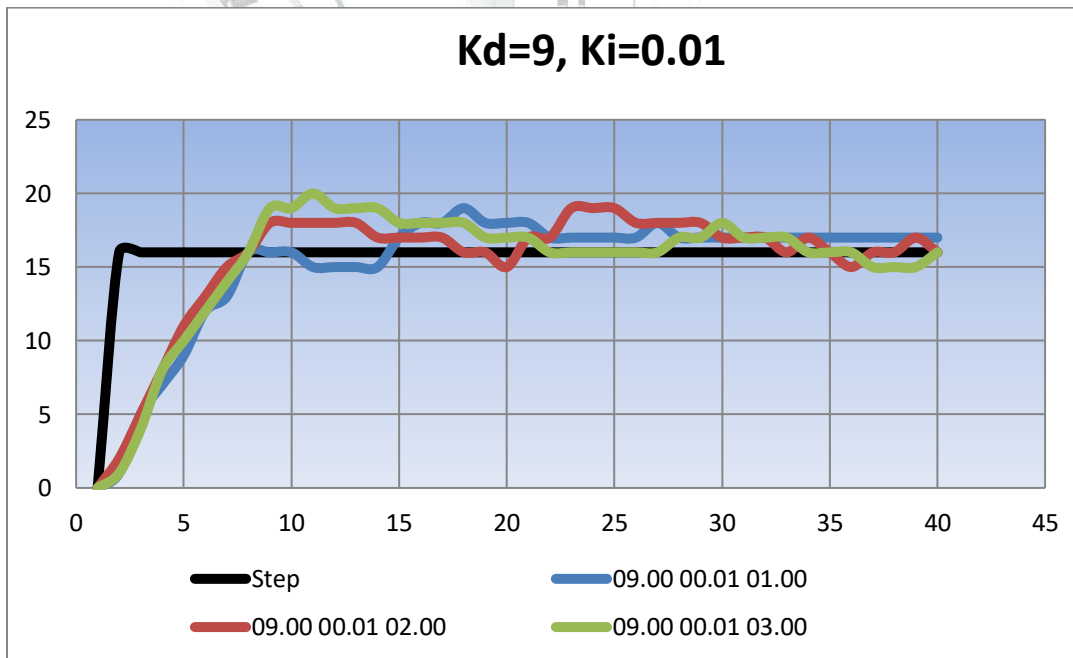
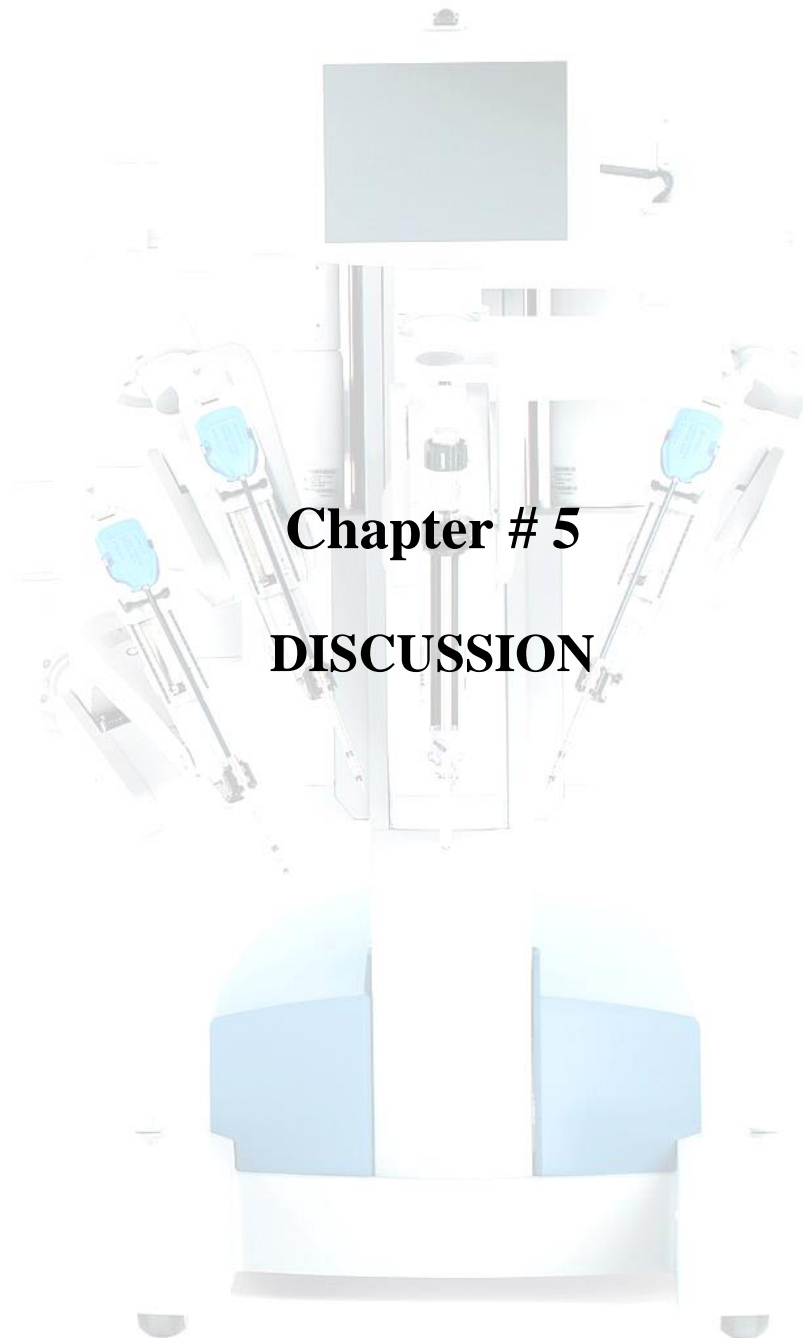


Figure 20: Response of motor with Kd=9; Ki=0.01



Chapter # 5

DISCUSSION

NETWORK COMMUNICATION

The 1st TCP code performed its function very well. It was reliable as a TCP should be. There was one very noticeable shortcoming of this code i.e. It can send only 1 control signal from each side on receiving signal from the other side.

This code was modified keeping in mind the improvements which were required.

Threading was the solution of our previous problem. TCP 2nd code embedded threading phenomenon and it yielded better results.

UDP yielded even better results. Delay measured from UDP was even less than 1ms at times.

JOYSTICK INTERFACING

X and Y axis are used to control the motion of the robotic arm. Z axis is used to lock the movement of the robot i.e. if value of z axis is less than 1000 then one motor is locked. Similarly if value of z axis is greater than 6000 then second motor is locked. Button valued 1 is used for gripper. Button valued 2 and 4 are used for translational motion. Similarly we have used a button to blocked motion of both x and y axis. We have added different modes of motion which can be selected using these buttons.

GUI

It has three buttons. One is “transmit” button which transmit joystick data. Second is “disconnect” button which is used to stop joystick transmission. Third button is “Quit” which is used to quit the surgeon program. It also shows data rate and delay.

PID CONTROLLER

It can be noticed that with the increase in K_d , the overshoots decreases significantly.

For $K_d=2$, the maximum overshoot was around 21 counts (with $K_p=3$) but as we increased K_d to 8, this overshoot was limited to around 18 counts. However, with further increase in K_d , the system becomes very sensitive and hence the value does not become stable.

It can also be seen that, keeping the other gains constant, when K_p is increased, the system gives more overshoot from the set-point value.

By analyzing all the plots for different gains, we conclude that the best combination for this controller is with;

- ✓ $K_p=2.00$
- ✓ $K_i=0.01$
- ✓ $K_d=8.00$

These values give the maximum possible stability and quickness of the response with minimum overshoot.

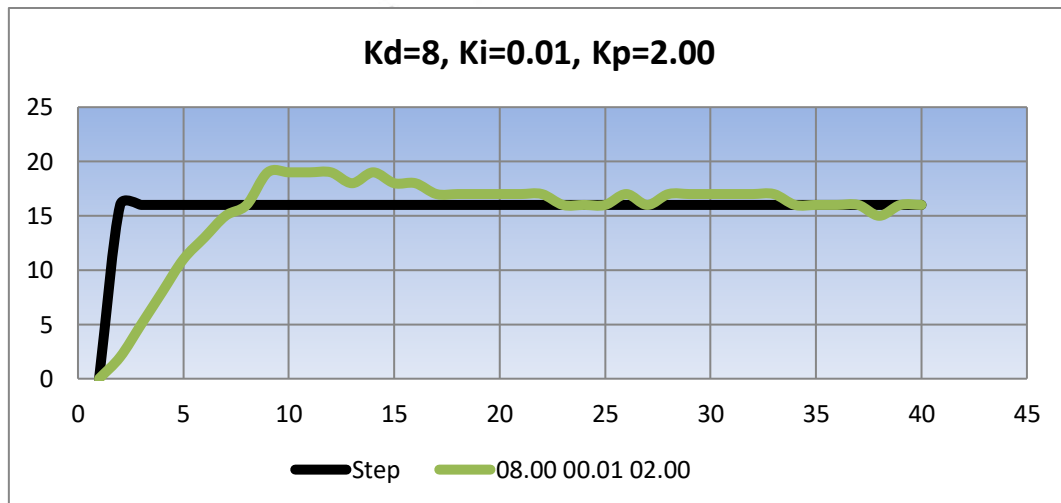


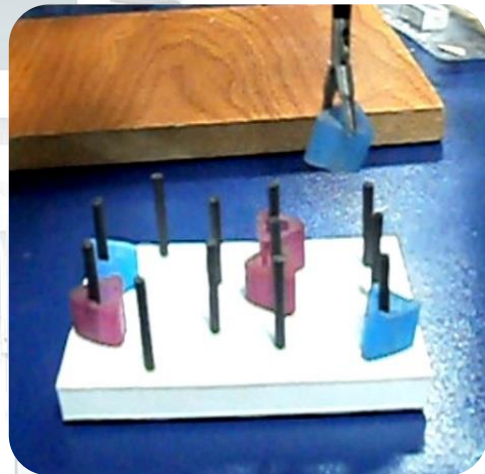
Figure 21: Best Response

Peg Transfer Exercise

Importance of Peg Transfer exercise cannot be denied in Tele-Surgery. It is used to train surgeons that how to control the robot effectively and efficiently. We also performed peg transfer exercise with our tele-surgical robot.

We were required to pick pegs from one side and place them on the other side in order i.e. same color pegs together in a same row.

We performed this exercise very successfully. We also faced some problems while doing so.

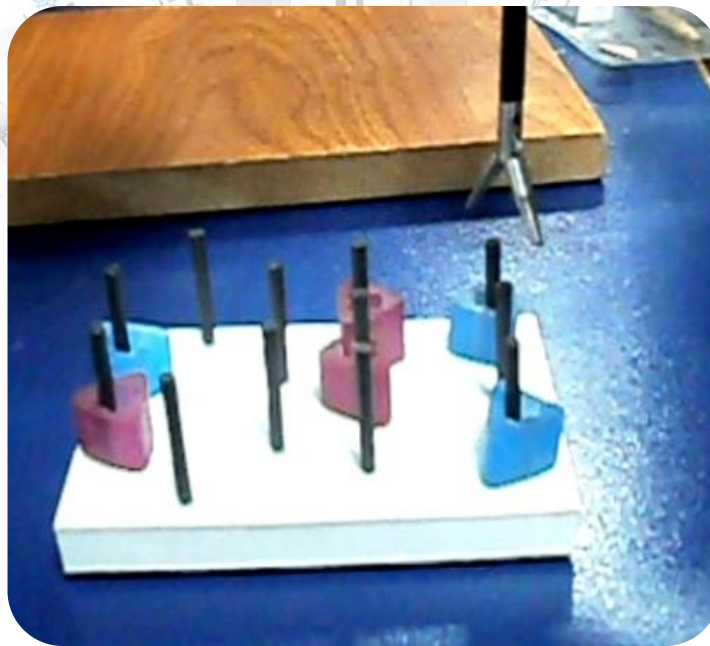


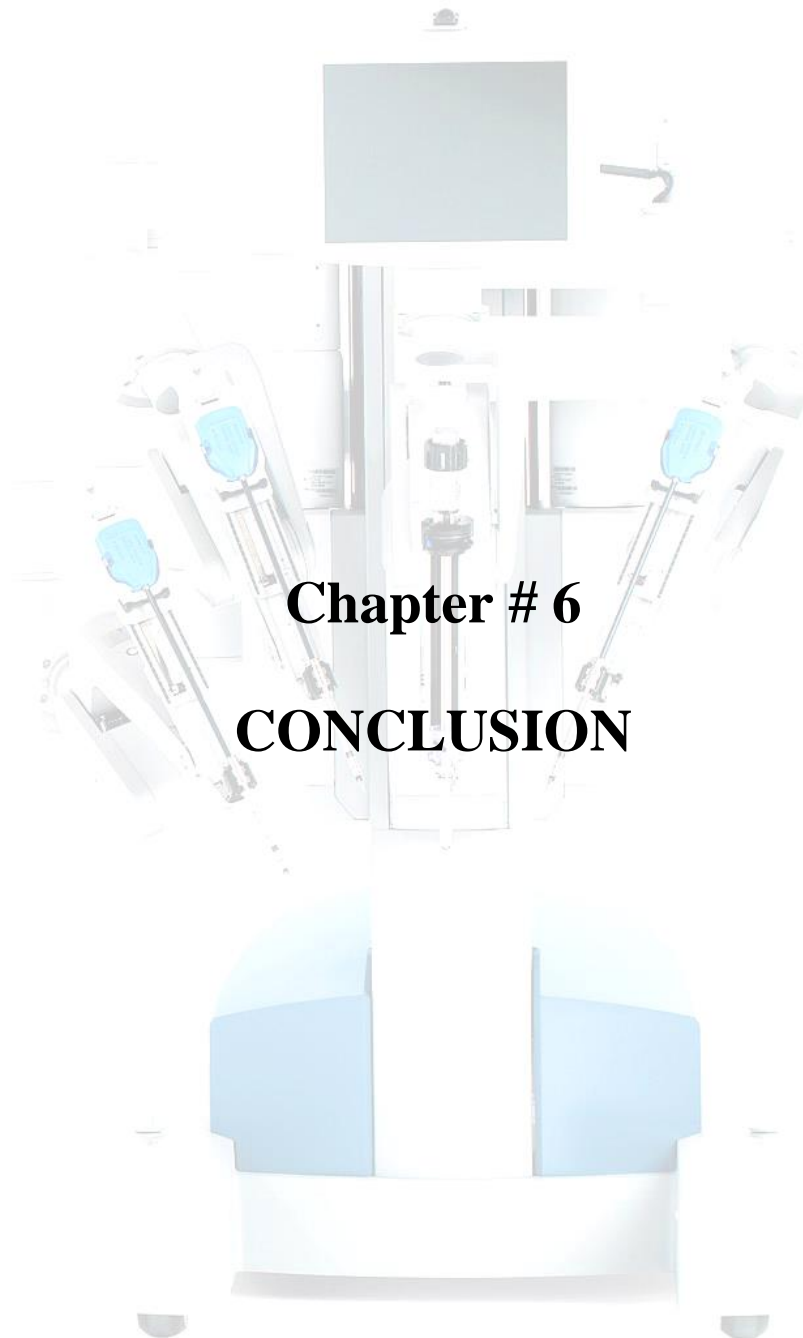
Problems faced during Peg Transfer

- Initially we were unable to control the robot. Joystick was very sensitive and its slightest movement was producing comparatively larger displacements.
- In our design we have tool's mouth opened at the front in default position. Whenever tool moved nearer to pegs, it collided with peg stands.
- Our speed was constant throughout our motion whereas it was required that motion near pegs should be slower than motion at larger distances from peg.

Solutions:

- First of all we reduced the sensitivity of our Joystick. It helped us to control the robot in better way.
- From the peg transfer exercise we concluded that in tele-surgery tool mouth should be kept shut until it is very close to the required object.
- We introduced different modes. In one mode it was operating at its original speed. In other mode we reduced the number of steps moved by robot with joystick movement, thus giving us more control.

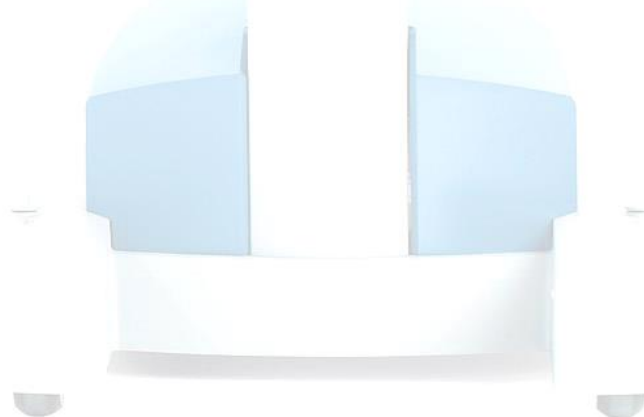




We developed a Tele-Surgical Training Robot prototype. Surgeon has to control the robot with some kind of input device which was joystick in our case. First of all we interfaced joystick with the surgeon PC. Secondly we mapped the joystick movement with robotic arm movement. Robot moved with joystick correspondingly. Master-Slave Configuration was used at patient side to control the Robotic Arm. Thirdly we developed a reliable hybrid UDP to perform as a Transport protocol for our network communication.

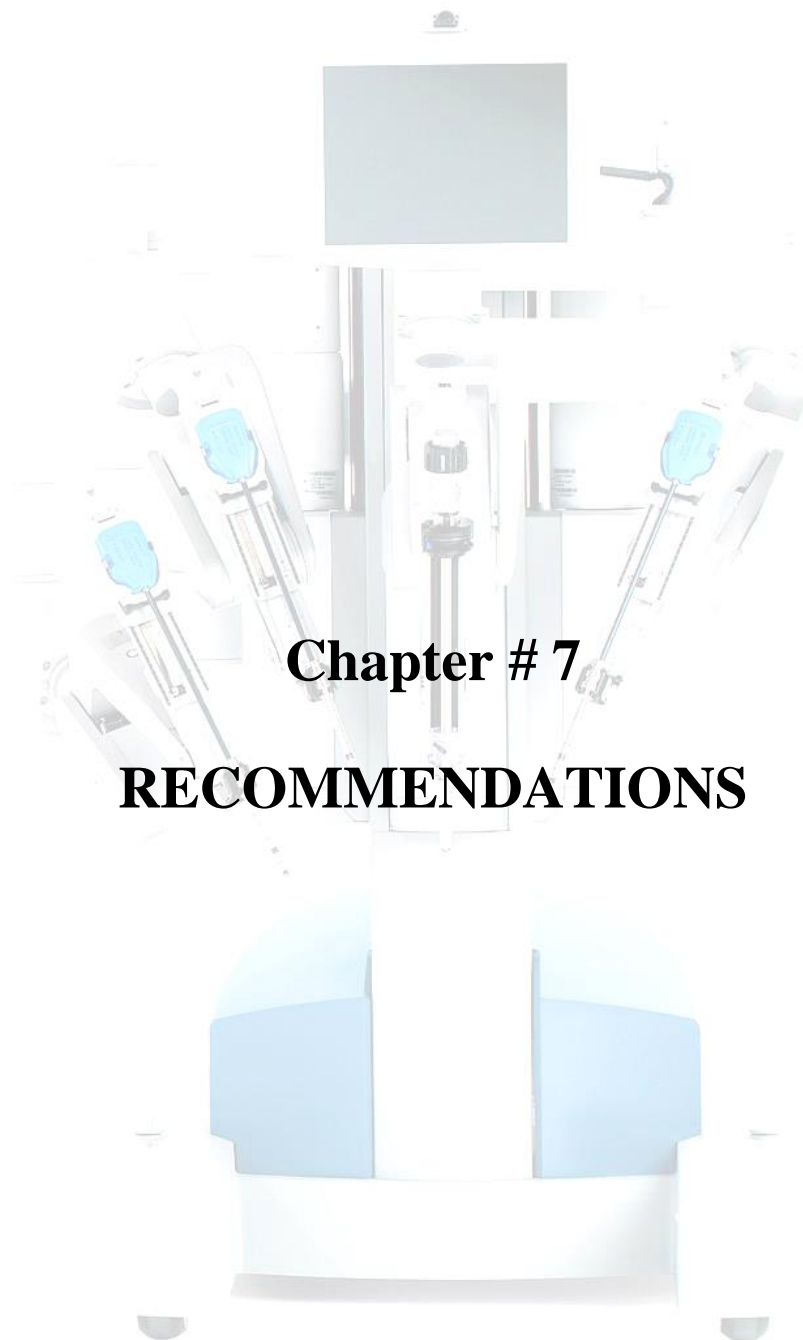
Now complete system can be very well understood. Surgeon is watching patient and the robot via camera and moving joystick to operate patient. Those joystick values are being transmitted to Patient PC via internet connection. Microcontrollers attached with Patient PC are controlling the motion of the robot according to the value of joystick they are receiving.

The system we have developed is a first prototype at SEECs NUST. It has performed the peg transfer exercise successfully. It will also act as a catalyst in the development of fully functional Tele-Surgical Robot.



The complete robotic system is shown below



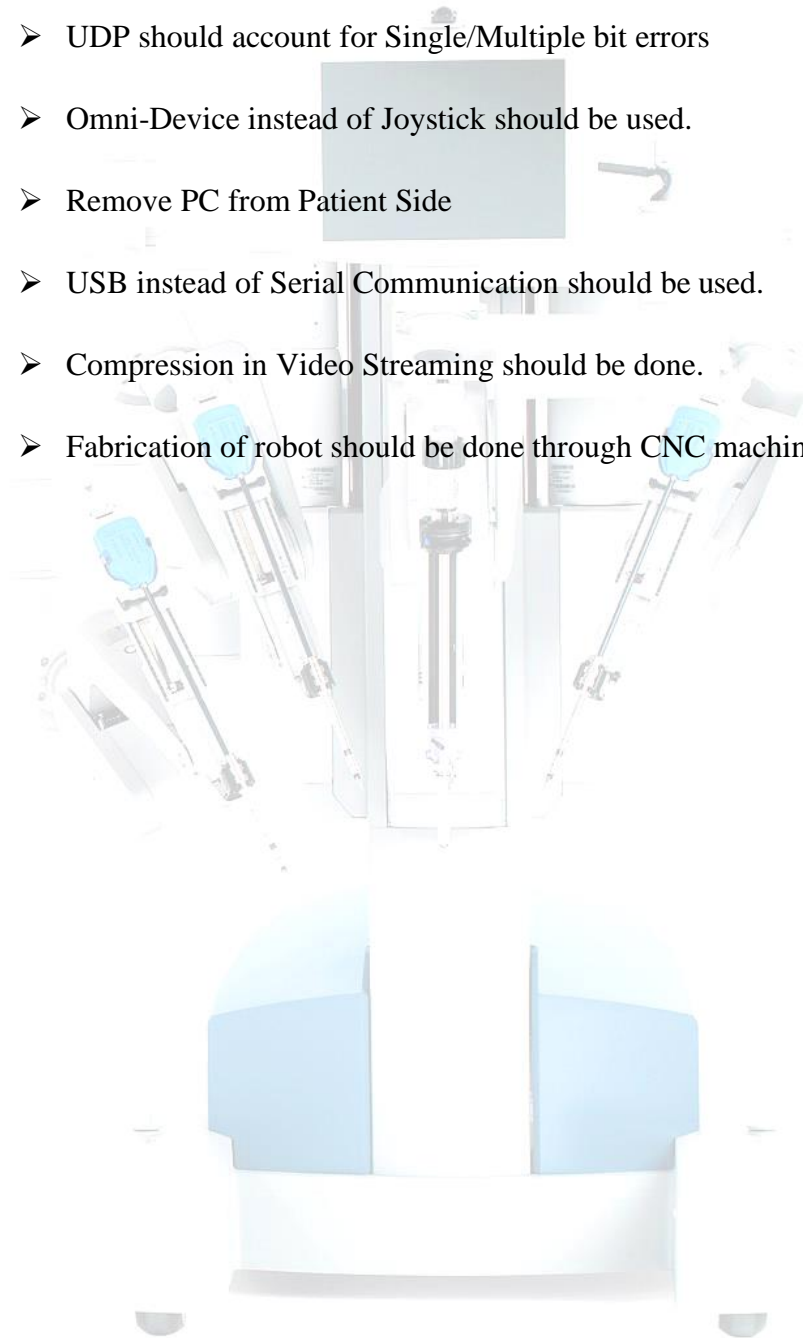


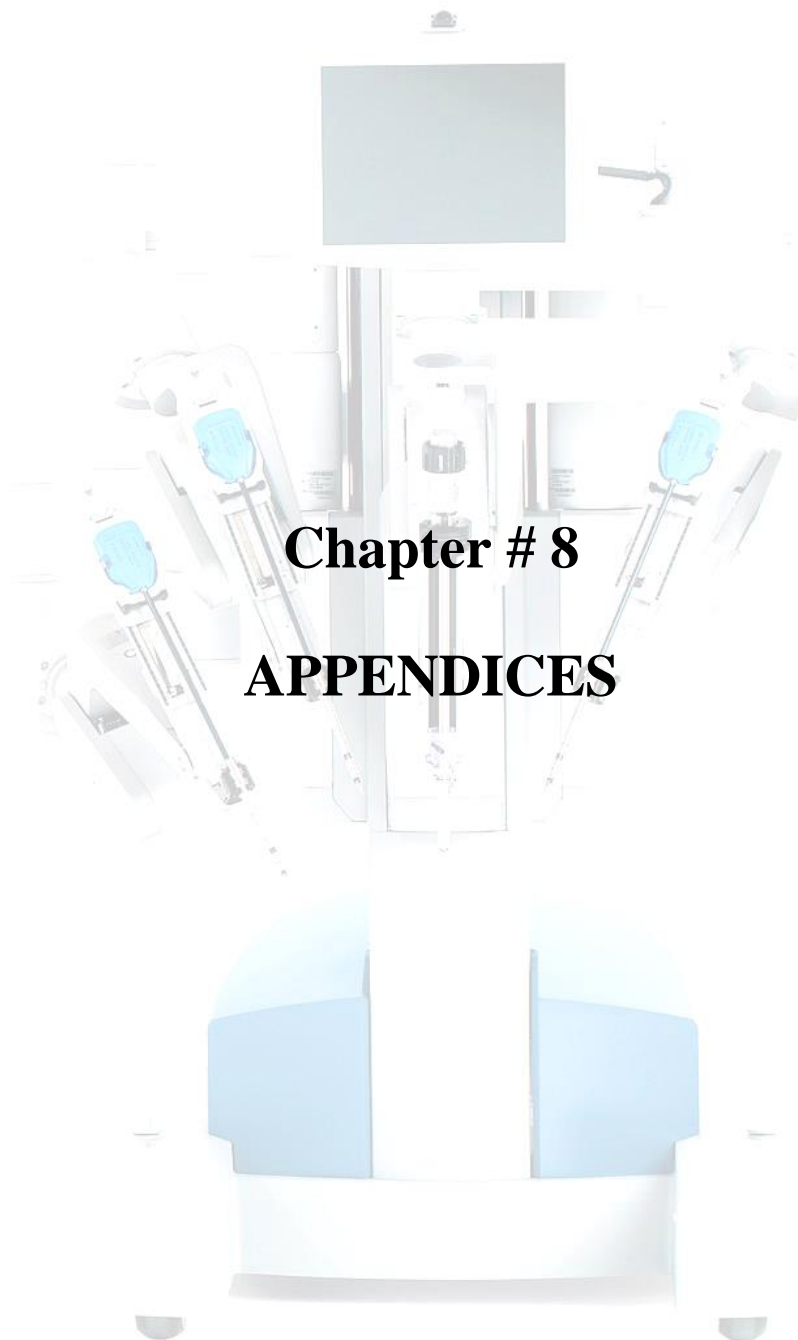
Chapter # 7

RECOMMENDATIONS

Following mentioned are some recommendations which can be implemented in future:

- UDP should account for Single/Multiple bit errors
- Omni-Device instead of Joystick should be used.
- Remove PC from Patient Side
- USB instead of Serial Communication should be used.
- Compression in Video Streaming should be done.
- Fabrication of robot should be done through CNC machines.





Chapter # 8

APPENDICES

Appendix A

Transmission Protocol Codes:

TCP-1

client code 1

```
#include <iostream>
#include "Socket.H"
#include <string>
using namespace std;

int main()
{
    int choice;
    int port = 80;
    string ipAddress;
        bool loops=false;
    char receive[STRLEN];
    cout<< "1) Connect to Server\n" <<endl;
    cout<< "2) Exit  ";
    cout<< "Enter Your choice  ";
        cin>>choice;

    if ( choice == 1 )
    {
        system("cls");

        //Client is connecting to server
        cout<<"Enter the IP Address of Server"<<endl;
        cin>>ipAddress;
        ClientSocket sockClient;
        cout<<"Connecting to Server..."<<endl;
        sockClient.ConnectToServer( ipAddress.c_str(), port );
        cout<<"        Connected to Server"<<endl;

        char* msg;
        char msgg [256];
```

```

        while ( !loops )
        {
sockClient.RecvData( receive, STRLEN );
cout<<"Signal recieved > "<<receive<<endl;
        cin.ignore();
        cin.get( msgg, STRLEN );
        msg=msgg;
sockClient.SendData(msg);

        //      system(receive);//dos commands executed on server
        //sockClient.Acknowledge(receive);
if ( strcmp( receive, "end" ) == 0)
        {
loops=true;
        }
        }

sockClient.CloseConnection();
        }
        else if ( choice == 2 )
exit(0);

        return 0;

}

```

Server Code 1

```

#include <iostream>
#include "Socket.H"
#include <string>
using namespace std;

int main()
{
int choice;
int port = 80;
string ipAddress;

```

```

char receive[STRLEN];
char send[STRLEN];
cout<<"\n1) Create Server\n"<<endl;
cout<<"2) Exit  ";
cout<<"Enter Your choice  ";
    cin>>choice;

if ( choice == 1 )
{
    system("cls");
    //SERVER
    ServerSocket sockServer;
    cout<<"Waiting for Client To Connect..."<<endl;
    sockServer.StartHosting( port );
    //Connected
    cout<<"  Client Connected"<<endl;

    char* msg;
    char msgg [256];
    //Connected

    cout<<"Send Control Signal"<<endl;

zed:
    cin.ignore();
    cin.get( msgg, STRLEN );
    msg=msgg;
    sockServer.SendData(msg);
    sockServer.RecvData( receive, STRLEN );
    cout<<"mesage send > "<<receive<<endl;

    goto zed;

}

else if ( choice ==2 )
exit(0);

return 0;

```

```
}
```

TCP-2

Client code 2:

```
#include <iostream>
#include "Socket.H"
#include <string>
using namespace std;
ClientSocket sockClient;

////////////////////////////////////

DWORD WINAPI StartThre(LPVOID iValue)
{
    bool loops=false;
    char receive[STRLEN];

    while ( !loops )
    {

        sockClient.RecvData( receive, STRLEN );
        cout<<"Signal recieved > "<<receive<<endl;

        if ( strcmp( receive, "end" ) == 0)
        {
            loops=true;
        }
    }

    sockClient.CloseConnection();

    return 0;
}

////////////////////////////////////
DWORD WINAPI StartThread(LPVOID iValue)
{
    bool loops=false;
```

```

        char* msg;
        char msgg [256];

while ( !loops )
{
        cout<<"Enter Data to Send: \n";
        cin>>msgg;
        msg=msgg;
        sockClient.SendData(msg);

if ( strcmp( msg, "end" ) == 0)
{
        loops=true;
}

}

sockClient.CloseConnection();

return 0;

}

////////////////////////////////////

void thread()
{
        HANDLE hThread1,hThread2;
        DWORD dwGenericThread;
        char lszThreadParam[3];
        strcpy(lszThreadParam,"3");
        hThread1 = CreateThread(NULL,0,StartThread,&lszThreadParam,0,&dwGenericThread);
        if(hThread1 == NULL)
        {
                DWORD dwError = GetLastError();
                cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
                return;
        }

        //Second thread creation
        strcpy(lszThreadParam,"30");
        hThread2 = CreateThread(NULL,0,StartThre,&lszThreadParam,0,&dwGenericThread);

```

```

if(hThread1 == NULL)
{
    DWORD dwError = GetLastError();
    cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
    return;
}
WaitForSingleObject(hThread2,INFINITE);
    WaitForSingleObject(hThread1,INFINITE);
}

////////////////////////////////////
void initial()
{
    int choice;
    int port = 80;
    string ipAddress;

    cout<< "1) Connect to Server\n"<<endl;
    cout<< "2) Exit  ";
    cout<< "Enter Your choice  ";
    cin>>choice;
    if ( choice == 1 )
    {
        system("cls");

        //Client is connecting to server
        cout<<"Enter the IP Address of Server"<<endl;
        cin>>ipAddress;

        cout<<"Connecting to Server..."<<endl;
        sockClient.ConnectToServer( ipAddress.c_str(), port );

        cout<<"    Connected to Server"<<endl;
    }
    else if ( choice == 2 )
        exit(0);
}
////////////////////////////////////
int main()
{
    initial();
    thread();
    return 0;
}

```

Server Code 2:

```

#include <iostream>
#include "Socket.H"
#include <string>
using namespace std;
ServerSocket sockServer;

////////////////////////////////////

DWORD WINAPI StartThre(LPVOID iValue)
{
    bool loops=false;
    char receive[STRLEN];

    while ( !loops )
    {
        sockServer.RecvData( receive, STRLEN );
        cout<<"Signal recieved > "<<receive<<endl;

        if ( strcmp( receive, "end" ) == 0)
        {
            loops=true;
        }
    }

    sockServer.CloseConnection();

    return 0;
}

////////////////////////////////////
DWORD WINAPI StartThread(LPVOID iValue)
{
    bool loops=false;

    char* msg;

```



```

        char msgg [256];

while ( !loops )
{
    cout<<"Enter Data to Send to Client: \n";

    cin>>msgg;
    msg=msgg;

sockServer.SendData(msg);

if ( strcmp( msg, "end" ) == 0)
{
    loops=true;
}

}

sockServer.CloseConnection();

return 0;

}

////////////////////////////////////

void thread()
{
    HANDLE hThread1,hThread2;
    DWORD dwGenericThread;
    char lszThreadParam[3];
    strcpy(lszThreadParam,"3");
    hThread1 =
    CreateThread(NULL,0,StartThread,&lszThreadParam,0,&dwGenericThread);
    if(hThread1 == NULL)
    {
        DWORD dwError = GetLastError();

```

```

cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
return;
}

//Second thread creation
strcpy(lszThreadParam,"30");
hThread2 =
CreateThread(NULL,0,StartThre,&lszThreadParam,0,&dwGenericThread);
if(hThread1 == NULL)
{
    DWORD dwError = GetLastError();
    cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
    return;
}
WaitForSingleObject(hThread2,INFINITE);
    WaitForSingleObject(hThread1,INFINITE);
}

////////////////////////////////////

void initial()
{
    int choice;
    int port = 80;
    string ipAddress;
    cout<<"\n1) Create Server\n"<<endl;
    cout<< "2) Exit  ";
    cout<< "Enter Your choice  ";
    cin>>choice;

    if ( choice == 1 )
    {
        system("cls");
        //SERVER

        cout<<"Waiting for Client To Connect..."<<endl;
        sockServer.StartHosting( port );

        //Connected
    }
}

```

```

        cout<<"    Client Connected"<<endl;
    }
    else if ( choice ==2 )
exit(0);

```

```

    }

int main()
{
    initial();
    thread();

    return 0;
}

```

SOCKET.H

```

#include <iostream>
#include "WinSock2.h"
#pragma comment(lib, "WS2_32.lib")

```

```
using namespace std;
```

```
const int STRLEN = 256;
```

```

class Socket
{
protected:
    WSADATA wsaData;
    SOCKET mySocket;
    SOCKET acceptSocket;
    sockaddr_in myAddress;
public:
    Socket();
    ~Socket();
    bool SendData( char* );
    bool RecvData( char*, int );
    void CloseConnection();
};

```

```

class ServerSocket : public Socket
{

```

```

public:
void Listen();
void Bind( int port );
void StartHosting( int port );
};

class ClientSocket : public Socket
{
public:
void ConnectToServer( const char *ipAddress, int port );
};

Socket::Socket()
{
if( WSASStartup( MAKEWORD(2, 2), &wsaData ) != NO_ERROR )
{
cerr<<"Socket Initialization: Error with WSASStartup\n";
system("pause");
WSACleanup();
exit(0);
}

//Create a socket
mySocket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

if ( mySocket == INVALID_SOCKET )
{
cerr<<"Socket Initialization: Error creating socket"<<endl;
system("pause");
WSACleanup();
exit(0);
}
}

Socket::~~Socket()
{
WSACleanup();
}

bool Socket::SendData( char *buffer )
{
send( mySocket, buffer, strlen( buffer ), 0 );
return true;
}

bool Socket::RecvData( char *buffer, int size )
{
int i = recv( mySocket, buffer, size, 0 );

```

```

buffer[i] = '\0';
return true;
}

void Socket::CloseConnection()
{
    closesocket( mySocket );
}

void ServerSocket::StartHosting( int port )
{
    Bind( port );
    Listen();
}

void ServerSocket::Listen()
{
    //cout<<"LISTEN FOR CLIENT..."<<endl;

    if ( listen ( mySocket,1) == SOCKET_ERROR )
    {
        cerr<<"ServerSocket: Error listening on socket\n";
        system("pause");
        WSACleanup();
        exit(0);
    }

    //cout<<"ACCEPT CONNECTION..."<<endl;

    acceptSocket = accept( mySocket, NULL, NULL );
    while ( acceptSocket == SOCKET_ERROR )
    {
        acceptSocket = accept( mySocket, NULL, NULL );
    }
    mySocket = acceptSocket;
}

void ServerSocket::Bind( int port )
{
    myAddress.sin_family = AF_INET;
    myAddress.sin_addr.s_addr = inet_addr( "0.0.0.0" );
    myAddress.sin_port = htons( port );
    if ( bind ( mySocket, (SOCKADDR*) &myAddress, sizeof( myAddress) ) ==
        SOCKET_ERROR )
    {
        cerr<<"ServerSocket: Failed to connect\n";
        system("pause");
        WSACleanup();
    }
}

```

```

exit(0);
    }
}

void ClientSocket::ConnectToServer( const char *ipAddress, int port )
{
    myAddress.sin_family = AF_INET;
    myAddress.sin_addr.s_addr = inet_addr( ipAddress );
    myAddress.sin_port = htons( port );
    if ( connect( mySocket, (SOCKADDR*) &myAddress, sizeof( myAddress ) ) ==
        SOCKET_ERROR )
    {
        cerr<<"ClientSocket: Failed to connect\n";
        system("pause");
        WSACleanup();
        exit(0);
    }
}

```

Data Rate:

```

clock_t endwait;
endwait = clock () + seconds * CLK_TCK ;
    while(clock()<endwait)
    {
    }
}

```

Delay:

```

__int64 ctrBefore = 0, ctrAfter = 0;
__int64 freq = 0;

sockServer.SendData( msg );
QueryPerformanceCounter( (LARGE_INTEGER *) &ctrBefore );

sockServer.RecvData( ireceive, STRLEN );

QueryPerformanceCounter( (LARGE_INTEGER *) &ctrAfter );
QueryPerformanceFrequency( (LARGE_INTEGER *) &freq );

cout<<"\n\n\tTime = "<<(((ctrAfter - ctrBefore)*1000) * 1.0 / freq) <<"
milliseconds"<<endl<<endl;

```

UDP Code:

Server:

```
#include <winsock2.h>
#include<iostream>
#include <stdio.h>
#include<string>
#include<time.h>
using namespace std;
#pragma comment(lib, "ws2_32.lib")

WSADATA      wsaData;
SOCKET       SendingSocket, ReceivingSocket;
SOCKADDR_IN  ReceiverAddr;
int          Port = 80;
char         ReceiveBuf[1024];
int          BufLength = 1024;
SOCKADDR_IN  SenderAddr;
SOCKADDR_IN  Sender1Addr;
int          SenderAddrSize = sizeof(SenderAddr);
int          ByteReceived = 10;
char         *SendBuf;
char         sendbuf[256];
int          TotalByteSent;
int          x=1;
string       store[256];
const char   *resend;
```

```

char                                *seq;
int                                y;
char                                *reseq;
////////////////////////////////////

DWORD WINAPI StartThre(LPVOID iValue)
{
    while(1)
    {
        int BufLengths=20;
        ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BufLengths, 0,
(SOCKADDR *)&SenderAddr, &SenderAddrSize);
        ReceiveBuf[ByteReceived]='\0';
        if(ReceiveBuf[0]=='~')
            exit(0);
        if(ReceiveBuf[0]=='o')
        {
            cout<<"Data received at "<<ReceiveBuf[1]<<" is: "<<ReceiveBuf<<endl;

        }

        if(ReceiveBuf[0]=='o')
        {

            y = static_cast<int>(ReceiveBuf[1]);
            resend=store[y].c_str();

            reseq = const_cast<char*>(resend);
            BufLength=strlen(reseq);
            TotalByteSent = sendto(SendingSocket, reseq, BufLength, 0,
(SOCKADDR *)&Sender1Addr, sizeof(Sender1Addr));

```



```

    }

    }
    return 0;
}

////////////////////////////////////
DWORD WINAPI StartThread(LPVOID iValue)
{
    while(1)
    {
        int *m=new int (x);

        seq = reinterpret_cast<char*>(m);

        cout<<"Enter Data to send: ";
        cin>>sendbuf;
        SendBuf=sendbuf;
        char end=*SendBuf;;
        strcat(seq,SendBuf);
        store[x]=seq;
        BufLength=strlen(seq);
        TotalByteSent = sendto(SendingSocket, seq, BufLength, 0, (SOCKADDR
*)&Sender1Addr, sizeof(Sender1Addr));
        if(end=='~')
            exit(0);

        if(x<=254)
        {

```

```

        x=x+1;
    }
    else x=1;

}

return 0;

}

////////////////////////////////////

void thread()
{

    HANDLE hThread1,hThread2;
    DWORD dwGenericThread;
    char lszThreadParam[3];
    strcpy(lszThreadParam,"3");
    hThread1
CreateThread(NULL,0,StartThread,&lszThreadParam,0,&dwGenericThread);
    if(hThread1 == NULL)
    {

        DWORD dwError = GetLastError();
        cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
        return;
    }

    //Second thread creation
    strcpy(lszThreadParam,"30");
    hThread2
CreateThread(NULL,0,StartThre,&lszThreadParam,0,&dwGenericThread);
    if(hThread1 == NULL)

```

```

    {
        DWORD dwError = GetLastError();
        cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
        return;
    }
    WaitForSingleObject(hThread2,INFINITE);
    WaitForSingleObject(hThread1,INFINITE);
}

////////////////////////////////////

int initial()
{
    // Initialize Winsock version 2.2
    if( WSStartup(MAKEWORD(2,2), &wsaData) != 0)
    {
        printf("Server: WSStartup failed with error %ld\n", WSAGetLastError());
        return -1;
    }

    SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (SendingSocket == INVALID_SOCKET)
    {
        printf("Client: Error at socket(): %ld\n", WSAGetLastError());
        // Clean up
        WSACleanup();
        // Exit with error
        return -1;
    }
}

```

```

    }

    // Create a new socket to receive datagrams on.
    ReceivingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    if (ReceivingSocket == INVALID_SOCKET)
    {
        printf("Server: Error at socket(): %ld\n", WSAGetLastError());
        // Clean up
        WSACleanup();
        // Exit with error
        return -1;
    }

    ReceiverAddr.sin_family = AF_INET;
    ReceiverAddr.sin_port = htons(Port);
    ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    // Associate the address information with the socket using bind.
    // At this point you can receive datagrams on your bound socket.
    if (bind(ReceivingSocket, (SOCKADDR *)&ReceiverAddr, sizeof(ReceiverAddr))
    == SOCKET_ERROR)
    {
        printf("Server: bind() failed! Error: %ld.\n", WSAGetLastError());
        closesocket(ReceivingSocket);
        WSACleanup();
        return -1;
    }

    ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BufLength, 0,
    (SOCKADDR *)&SenderAddr, &SenderAddrSize);

```

```

ReceiveBuf[ByteReceived]='\0';
cout<<"Length of Byte Received is: "<<ByteReceived<<endl;
cout<<"Data received is: "<<ReceiveBuf<<endl;
getpeername(ReceivingSocket, (SOCKADDR *)&SenderAddr, &SenderAddrSize);

Sender1Addr.sin_family = AF_INET;
Sender1Addr.sin_port = htons(Port);
Sender1Addr.sin_addr.s_addr = SenderAddr.sin_addr.s_addr;

////////////////////////////////////

SendBuf=ReceiveBuf;
__int64 ctrBefore = 0, ctrAfter = 0;
__int64 freq = 0;

BufLength=strlen(SendBuf);
TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0,
(SOCKADDR *)&Sender1Addr, sizeof(Sender1Addr));

QueryPerformanceCounter( (LARGE_INTEGER *) &ctrBefore );

ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BufLength, 0,
(SOCKADDR *)&SenderAddr, &SenderAddrSize);
ReceiveBuf[ByteReceived]='\0';

QueryPerformanceCounter( (LARGE_INTEGER *) &ctrAfter );
QueryPerformanceFrequency( (LARGE_INTEGER *) &freq );

cout<<"\n\n\tTime = "<<((((ctrAfter - ctrBefore)*1000) * 1.0 / freq) )<<"
milliseconds"<<endl<<endl;

```

////////////////////////////////////

```
int BuffingLength=256;
```

```
int f=0;
```

```
clock_t endwait;
```

```
endwait = clock () + 10 * CLK_TCK ;
```

```
while(clock())<endwait)
```

```
{
```

```
ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BuffingLength, 0,  
(SOCKADDR *)&SenderAddr, &SenderAddrSize);
```

```
ReceiveBuf[ByteReceived]='\0';
```

```
f=f+1;
```

```
}
```

```
cout<<"Number of Packets are: "<<f<<endl;
```

```
int datarate=f*256*8/10000;
```

```
cout<<"Data Rate is: "<<datarate<<" kbps "<<endl;
```

////////////////////////////////////

```
return 0;
```

```
}
```

```
////////////////////////////////////
```

```
int main()
```

```
{
```

```
    initial();
```

```
    thread();
```

```
    return 0;
```

```
}
```

```
////////////////////////////////////
```

CLIENT

```
#include <winsock2.h>
```

```
#include<iostream>
```

```
#include <stdio.h>
```

```
#include<string>
```

```
#include<time.h>
```

```
using namespace std;
```

```
#pragma comment(lib, "ws2_32.lib")
```

```
WSADATA    wsaData;
```

```
SOCKET     SendingSocket, ReceivingSocket;
```

```
SOCKADDR_IN ReceiverAddr;
```

```
int         Port = 80;
```

```

char      *SendBuf;
char      sendbuf[256];
int       BufLength = 1024;
int       TotalByteSent;
char      ipAddress[30];
char      *IPAddress;
char      ReceiveBuf[1024];
SOCKADDR_IN  Sender1Addr;
SOCKADDR_IN  SenderAddr;
int       SenderAddrSize = sizeof(SenderAddr);
int       ByteReceived = 1024;
int       x=1;
int       seq=0;
char      *resend;
char      *buffer;
int       *m;
int       r=1;
string    store[256];
char      putback[20];

int       received[256];
int       check=0;

```

```

////////////////////////////////////

```

```

DWORD WINAPI StartThre(LPVOID iValue)

```

```

{

```

```

    while(1)

```

```

    {

```

```

        cout<<"Enter Data to send: ";

```

```

        cin>>SendBuf;

```



```

        char end=*SendBuf;
        BufLength=strlen(SendBuf);

        TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0, (SOCKADDR
*)&Sender1Addr, sizeof(Sender1Addr));
        if(end=='~')
            exit(0);
    }
    return 0;
}

////////////////////////////////////
DWORD WINAPI StartThread(LPVOID iValue)
{
    while(1)
    {

        // At this point you can receive datagrams on your bound socket.
        int BufLengths=20;
        ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BufLengths, 0,
(SOCKADDR *)&SenderAddr, &SenderAddrSize);
        ReceiveBuf[ByteReceived]='\0';
        if(ReceiveBuf[1]=='~')
            exit(0);
        buffer=ReceiveBuf;
        seq = static_cast<int>(ReceiveBuf[0]);
        if(seq<=0)
        { seq=seq+256;
        }
        if(check<seq)
        {

```

```

        check=seq;
    }
    received[seq]=1;

    store[seq]=buffer;

again:
    if(x<check && received[x]!=1)
    {
        m=new int (x);

        resend = reinterpret_cast<char*>(m);

        char sending[256]="o";
        strcat(sending,resend);
        SendBuf=sending;
        BufLength=strlen(SendBuf);

        TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0,
(SOCKADDR *)&Sender1Addr, sizeof(Sender1Addr));

    }

    else if(x<check && received[x]==1)
    {

```

```

for(int z=1; z<store[x].size(); z++)
{
    if(z<=18)
        putback[z-1] = (char)store[x].at(z);

}
putback[store[x].size()-1]='\0';
cout<<"Putback is: "<<putback<<endl;
received[x]=0;
if(x<=254)
    x=x+1;
else
    x=1;
goto again;
}

else if(x==check)
{
for(int z=1; z<store[x].size(); z++)
{
    if(z<=18)
        putback[z-1] = (char)store[x].at(z);

}
putback[store[x].size()-1]='\0';
cout<<"Putback is: "<<putback<<endl;

```

```

        received[x]=0;
    if(x<=254)
        x=x+1;
    else
        x=1;
    }

    if(check==255)
        check=0;
}

return 0;

}

////////////////////////////////////

void thread()
{
    HANDLE hThread1,hThread2;
    DWORD dwGenericThread;
    char lszThreadParam[3];
    strcpy(lszThreadParam,"3");
    hThread1
    CreateThread(NULL,0,StartThread,&lszThreadParam,0,&dwGenericThread);
    if(hThread1 == NULL)

```

```

    {
        DWORD dwError = GetLastError();
        cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
        return;
    }

    //Second thread creation
    strcpy(lszThreadParam,"30");
    hThread2
CreateThread(NULL,0,StartThre,&lszThreadParam,0,&dwGenericThread);
    if(hThread1 == NULL)
    {
        DWORD dwError = GetLastError();
        cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
        return;
    }
    WaitForSingleObject(hThread2,INFINITE);
    WaitForSingleObject(hThread1,INFINITE);
}

```

////////////////////////////////////

```

int initial()
{
    // Initialize Winsock version 2.2
    if( WSStartup(MAKEWORD(2,2), &wsaData) != 0)
    {

```

```

        printf("Client: WSAShutdown failed with error %ld\n", WSAGetLastError());
        // Clean up
        WSACleanup();
        // Exit with error
        return -1;
    }

    // Create a new socket to receive datagrams on.
    SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (SendingSocket == INVALID_SOCKET)
    {
        printf("Client: Error at socket(): %ld\n", WSAGetLastError());
        // Clean up
        WSACleanup();
        // Exit with error
        return -1;
    }

    ReceivingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    if (ReceivingSocket == INVALID_SOCKET)
    {
        printf("Server: Error at socket(): %ld\n", WSAGetLastError());
        // Clean up
        WSACleanup();
        // Exit with error
        return -1;
    }

    cout<<"Enter IP Address: ";
    cin>>ipAddress;

```

```

IPAddress=ipAddress;

// Set up a SOCKADDR_IN structure that will identify who we
// will send datagrams to. For demonstration purposes, let's
// assume our receiver's IP address is 127.0.0.1 and waiting
// for datagrams on port 80.
ReceiverAddr.sin_family = AF_INET;
ReceiverAddr.sin_port = htons(Port);
ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(ReceivingSocket, (SOCKADDR *)&ReceiverAddr, sizeof(ReceiverAddr))
== SOCKET_ERROR)
{
    printf("Server: bind() failed! Error: %ld.\n", WSAGetLastError());
    // Close the socket
    closesocket(ReceivingSocket);
    // Do the clean up
    WSACleanup();
    // and exit with error
    return -1;
}

Sender1Addr.sin_family = AF_INET;
Sender1Addr.sin_port = htons(Port);
Sender1Addr.sin_addr.s_addr = inet_addr(IPAddress);
SendBuf="Start";
BufLength=strlen(SendBuf);
TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0, (SOCKADDR
*)&Sender1Addr, sizeof(Sender1Addr));
////////////////////////////////////
int BufLengths=20;

```

```
ByteReceived = recvfrom(ReceivingSocket, ReceiveBuf, BufLengths, 0,  
(SOCKADDR *)&SenderAddr, &SenderAddrSize);
```

```
ReceiveBuf[ByteReceived]='\0';
```

```
SendBuf=ReceiveBuf;
```

```
BufLength=strlen(SendBuf);
```

```
TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0, (SOCKADDR  
&Sender1Addr, sizeof(Sender1Addr));
```

```
clock_t endwait;
```

```
endwait = clock () + 10 * CLK_TCK ;
```

```
char datarate[256];
```

```
datarate[0]='&';
```

```
for(int i=1; i<256; i++)
```

```
{
```

```
    datarate[i]='&';
```

```
}
```

```
char * datarates;
```

```
datarates=datarate;
```

```
BufLength=strlen(datarates);
```

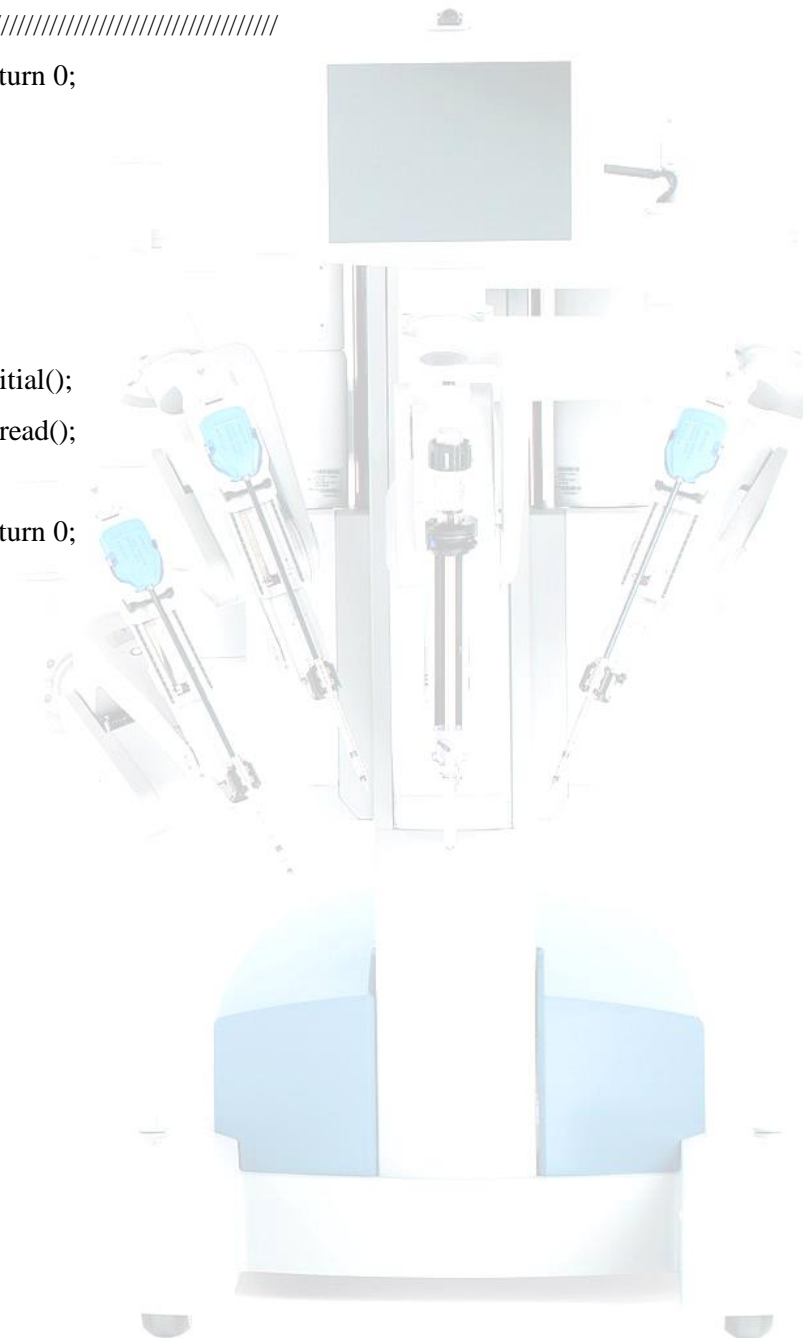
```
while(clock())<endwait)
```

```
{
```

```
TotalByteSent = sendto(SendingSocket, SendBuf, BufLength, 0, (SOCKADDR  
&Sender1Addr, sizeof(Sender1Addr));
```



```
    }  
  
    ///////////////////////////////////  
    return 0;  
}  
int main()  
{  
  
    initial();  
    thread();  
  
    return 0;  
}
```



Appendix B

Serial Communication

CODE

```
#include<iostream>
#include<Windows.h>
#include<string.h>
usingnamespace std;
HANDLE hPort,hPort1;
void write(char*);
void initial();
////////////////////////////////////
DWORD WINAPI StartThre(LPVOID iValue)
{
    char datas;
    while(1)
    {
        cin>>datas;
        write(&datas);
    }
    return 0;
}
////////////////////////////////////
DWORD WINAPI StartThread(LPVOID iValue)
{
    DWORD dwRead;
    char chRead;
    while(1)
    {
        if (ReadFile(hPort, &chRead, 1, &dwRead, NULL))
        {
            if(chRead!='~')
            {
                cout<<chRead; dwRead=0;
                chRead='~';
            }
        }
    }
}
```

```

        }
    }
    return 0;
}

////////////////////////////////////
void write (char* data)
{
    DWORD byteswritten;
    bool retVal = WriteFile(hPort,data,1,&byteswritten,NULL);
}

////////////////////////////////////
void main()
{
    initial();
    HANDLE hThread1,hThread2;
    DWORD dwGenericThread;
    char lszThreadParam[3];
    //First thread creation
    strcpy(lszThreadPara,"3");
    hThread1 =
    CreateThread(NULL,0,StartThread,&lszThreadPara,0,&dwGenericThread);
    if(hThread1 == NULL)
    {
        DWORD dwError = GetLastError();
        cout<<"SCM:Error in Creating thread"<<dwError<<endl ;
        return;
    }

    //Second thread creation
    strcpy(lszThreadPara,"30");
    hThread2 =
    CreateThread(NULL,0,StartThre,&lszThreadPara,0,&dwGenericThread);
    if(hThread2 == NULL)
    {
        DWORD dError = GetLastError();
        cout<<" Error in Creating thread"<<dError<<endl ;
        return;
    }
    WaitForSingleObject(hThread2,INFINITE); //Wait till thread 2 is completed
    WaitForSingleObject(hThread1,INFINITE); //Wait till thread 1 is completed
    CloseHandle(hPort); //close the handle
}

////////////////////////////////////

```

```

void initial()
{
    DCB dcb;
    hPort
=CreateFile("\\\\.\\COM11",GENERIC_WRITE|GENERIC_READ,0,NULL,OPEN_E
XISTING,0,NULL);
    if (!GetCommState(hPort,&dcb))
    { cout<<"can't get rate "<<endl;
    }
    dcb.BaudRate = CBR_9600; //9600 Baud
    dcb.ByteSize = 8; //8 data bits
    dcb.Parity = NOPARITY; //no parity
    dcb.StopBits = ONESTOPBIT; //1 stop
    if (SetCommState(hPort,&dcb) == 0)
    {
        cout<<"can't set rate "<<endl;
    }
    COMMTIMEOUTS comTimeOut;
    // Specify time-out between character for receiving.

    comTimeOut.ReadIntervalTimeout = 5;
    // Specify value that is multiplied

    // by the requested number of bytes to be read.

    comTimeOut.ReadTotalTimeoutMultiplier = 5;
    // Specify value is added to the product of the

    // ReadTotalTimeoutMultiplier member

    comTimeOut.ReadTotalTimeoutConstant = 5;
    // Specify value that is multiplied

    // by the requested number of bytes to be sent.

    comTimeOut.WriteTotalTimeoutMultiplier = 5;
    // Specify value is added to the product of the

    // WriteTotalTimeoutMultiplier member

    comTimeOut.WriteTotalTimeoutConstant = 5;
    // set the time-out parameter into device control.

    SetCommTimeouts(hPort,&comTimeOut);

```

```

        if (!SetCommState(hPort,&dcb))
        {
            cout<<"can't set time "<<endl;
        }
    }
}

```

Joystick Interfacing

Code:

```

#include<stdlib.h>
#include<glut.h>
#include<string>
#include<windows.h>
#include<iostream>
usingnamespace std;

HWND hWnd; // Window's Handle
// Global variables for the Joystick
bool joy_ok;
UINT joy_ID;
UINT joy_num;
JOYCAPS joy_caps;
JOYINFO joy_info;
DWORD joy_xcenter; // joy x axis center
DWORD joy_ycenter; // joy y axis center
DWORD joy_zcenter; // joy Z axis center

//JoyStick Functions
bool InitJoystick();
void ReleaseJoystick();

////////////////////////////////////
void JoyFunc(unsignedint buttonMask,int x, int y, int z)
{
    joyGetPos(joy_ID,&joy_info);
    cout<<" "<<joy_info.wXpos<<" , "<<joy_info.wYpos<<" , "<<joy_info.wZpos<<
" "<<joy_info.wButtons<<endl;
    int x1= joy_info.wXpos; // x coordinate of joystick
    int y1= joy_info.wYpos; // y coordinate of joystick
    int z1= joy_info.wZpos; // z coordinate of joystick
    Sleep(10000);
}

```

```

////////////////////////////////////
void display(void)
{
    glutForceJoystickFunc();
}

////////////////////////////////////

int main(int argc, char** argv)
{
    system("cls");
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (00, 00); // size is in pixels
    glutInitWindowPosition (-100, -100); // location is in pixels
    glutCreateWindow ("joystick"); // change the title of the window
    joy_ok=InitJoystick();
    glutDisplayFunc(display);
    glutJoystickFunc(JoyFunc,10);
    glutMainLoop();
    return 0;
}

////////////////////////////////////
void ReleaseJoystick()
{
    if (joy_ok)
        joyReleaseCapture(joy_ID); // release joystick
}

////////////////////////////////////

bool InitJoystick()
{
    if ((joy_num = joyGetNumDevs()) == 0)
        return FALSE;

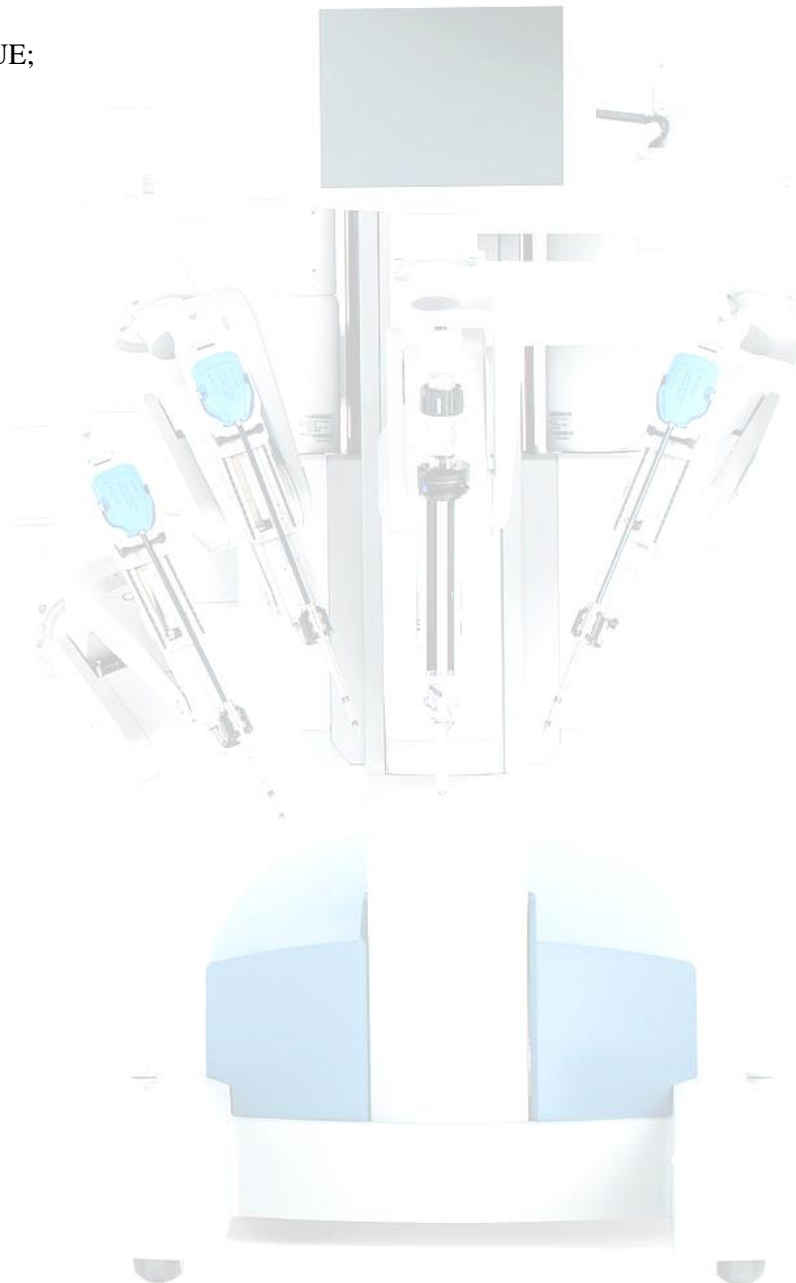
    if (joyGetPos(JOYSTICKID1, &joy_info) != JOYERR_UNPLUGGED)
        joy_ID = JOYSTICKID1;
    else
        return FALSE;

    joyGetDevCaps(joy_ID, &joy_caps, sizeof(JOYCAPS));
}

```

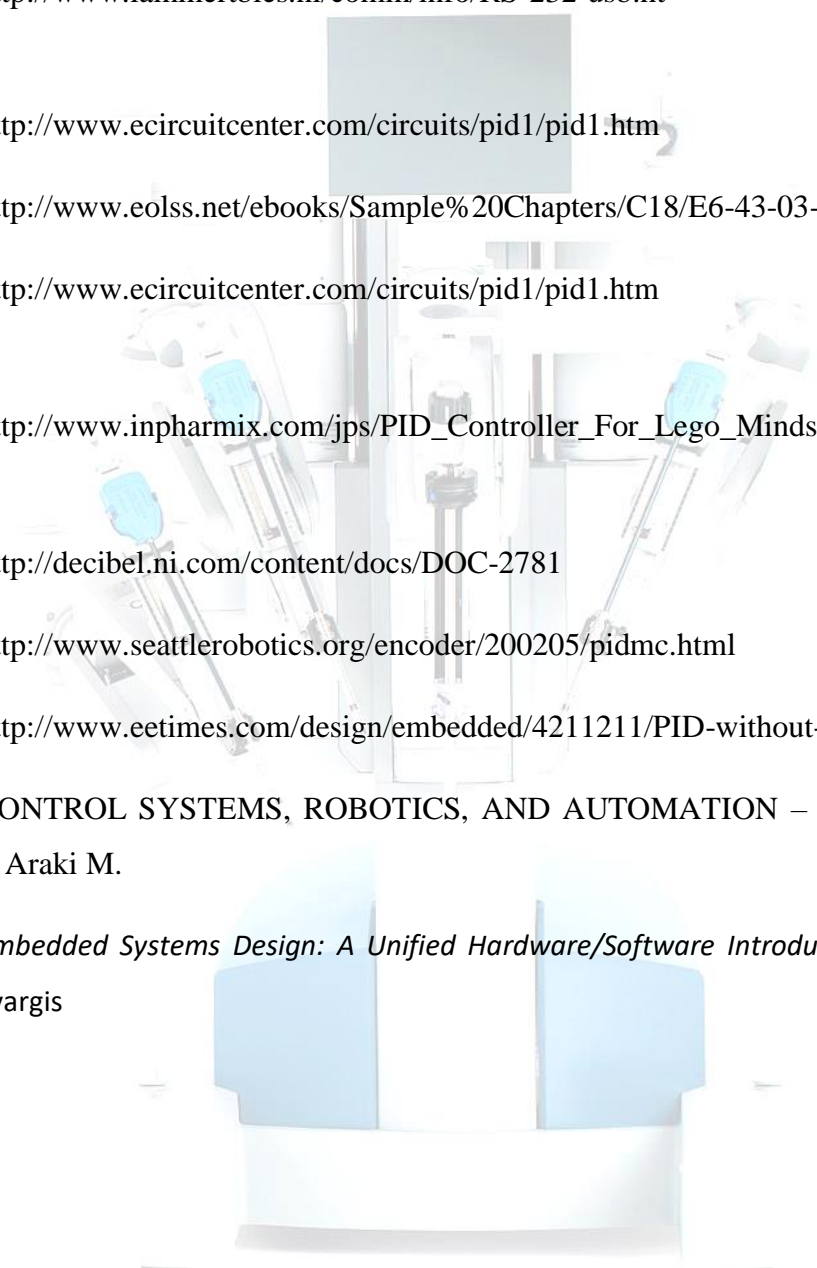
```
joy_xcenter = ((DWORD)joy_caps.wXmin + joy_caps.wXmax) / 2;  
joy_ycenter = ((DWORD)joy_caps.wYmin + joy_caps.wYmax) / 2;  
joy_zcenter = ((DWORD)joy_caps.wZmin + joy_caps.wZmax) / 2;  
cout<<joy_caps.wMaxButtons;  
joySetCapture(hWnd, joy_ID, NULL, TRUE);
```

```
return TRUE;  
}
```



References

- [1] **Telesurgery of Microscopic Micromanipulator System “NeuRobot” in Neurosurgery: Interhospital Preliminary Study**
Tetsuya Goto¹, Takahiro Miyahara¹, Kazutaka Toyoda³, Jun Okamoto²,
Yukinari Kakizawa¹, Jun-ichi
Koyama¹, Masakatsu G. Fujie² and Kazuhiro Hongo¹
¹Department of Neurosurgery, Shinshu University School of Medicine,
Matsumoto, Japan.²Faculty of Science and Engineering, Waseda University,
Tokyo, Japan.³Graduate school of Science and Engineering, Waseda
University, Tokyo, Japan. Email: khongo@shinshu-u.ac.jp.
- [2] **Transforming a Surgical Robot for Human Telesurgery**
Steven E. Butner, *Senior Member, IEEE*, and Moji Ghodoussi
- [3] **The RAVEN: Design and Validation of a Telesurgery System**
Mitchell J. H. Lum, Diana C. W. Friedman, Ganesh Sankaranarayanan, Hawkeye
King, Kenneth Fodero II, Rainer Leuschke, Blake Hannaford
- [4] http://en.wikipedia.org/wiki/Transmission_Control_Protocol
- [5] http://www.websurg.com/event/Operation_Lindbergh/lindbergh_presse_en.pdf
- [6] <http://www.surgeryencyclopedia.com/St-Wr/Telesurgery.html>
- [7] <http://www.ispub.com/ostia/index.php?xmlFilePath=journals/ijh/vol5n1/davinci.xml>
- [8] <http://zone.ni.com/devzone/cda/tut/p/id/3782>
- [9] <http://tech-wonders.blogspot.com/2009/02/phantom-omni-haptic-device.html>

- 
- [10] <http://www.sensable.com/industries-application-development.htm>
- [11] <http://www.totalphase.com/support/kb/10048/>
- [12] <http://www.lammerbries.nl/comm/info/RS-232-usb.ht>
- [13] <http://www.ecircuitcenter.com/circuits/pid1/pid1.htm>
- [14] <http://www.eolss.net/ebooks/Sample%20Chapters/C18/E6-43-03-03.pdf>
- [15] <http://www.ecircuitcenter.com/circuits/pid1/pid1.htm>
- [16] http://www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html
- [17] <http://decibel.ni.com/content/docs/DOC-2781>
- [18] <http://www.seattlerobotics.org/encoder/200205/pidmc.html>
- [19] <http://www.eetimes.com/design/embedded/4211211/PID-without-a-PhD>
- [20] CONTROL SYSTEMS, ROBOTICS, AND AUTOMATION – Vol. II - PID Control - Araki M.
- [21] *Embedded Systems Design: A Unified Hardware/Software Introduction*,(c) 2000 Vahid/Givargis