A Distributed Robot Of Intelligent Devices (DROID): Conception, Design, and Implementation for Autonomous Multiple Robotic Welding

by

Luke Ng

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical Engineering

Waterloo, Ontario, Canada, 2001

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signature of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Sensor-based control of multiple industrial robot systems requires a large number of sensors and robot manipulators to be integrated. As the demand for agile manufacturing cells to become more adaptable and reconfigurable in terms of both hardware and software increases, data management and coordination of the numerous connected units becomes difficult.

To attain this flexibility or agility, an operating system environment tailored for complex robot systems, called a Distributed Robot of Intelligent Devices (DROID) has been developed. Using low-cost/high performance computing hardware made available by the personal computer market, the DROID system is a distributed control system where sensors or robots can be added or removed based on the task being performed. The DROID system's purpose is to act as a research development platform on which to base future sensor-based multiple robotics research.

This thesis begins with the description of the system architecture which promotes extensive integration, reconfiguration and expansion. It then proceeds to describe the DROID system's implementation of our chosen real-world application, autonomous robotic welding or dynamic seam-tracking which involves 2 robotic arm manipulator units and a laser profiling sensor (effectively a 12-degree of freedom system driven by a sensor). We then discuss its successful tracking performance and the future of the DROID system.

Acknowledgements

I would like to thank my supervisor Dr. Jan P. Huissoon for his support, knowledge, advice, patience, sense of humor and troubleshooting skills during my studies.

I would like to thank my friends and colleagues, Paul Gray, S.J. Park, Micheal Kim, Derry J. Crymble, Trevor Tsang, Aaron Geiseberger, Peter Routledge, and Alex Parlour for their comradery, advice, colorful knowledge and also their sense of humor throughout my studies.

I would also like to thank my father for teaching me the value of hard work and perseverance.

Especially, I would like to thank my wife Sharon for her relentless encouragement, advice and support. Without her there would be no thesis.

The financial support of this project was provided by I.R.I.S. Without their continued support, this project would not have been possible

Dedication

This thesis is dedicated to my loving wife Sharon. Thank you for your never-ending patience and encouraging me to dream beyond my wildest imaginations.

Table of Contents

Abstract iii

Table of Contents vii

List of Tables xi

List of Figures xii

Chapter 1. Introduction 1

1.1. Background 2
1.2. Rationale and Objectives 7

Chapter 2. Hardware and Software Technologies 10

2.1. Current Computer and Robot Technology 10
2.1.1. Computer Technology 10
2.1.2. Robot Technology 13
2.2. Operating Systems 15
2.3. The Reis RobotStar V15 17
2.4. The GM Fanuc S400 20

2.5. The MVS Line Laser Sensor 23
2.6. The Brain Computer 24

Chapter 3. System Architecture 27

3.1. Overview 27
3.2. Network Architecture and Communications Protocol 29
3.3. Subunit Interfaces 31
3.4. Task Programs 36

Chapter 4. Generic Motion Control 38

4.1. Prototype Implementation, the Reis Robotstar V15 38
4.1.1. Robot Mechanics and Kinematics 38
4.1.1.1. Robot Forward Kinematics 39
4.1.1.2. The Inverse Kinematic Solution 42
4.1.1.3. Encoder Gearing and Interconnectivity 44
4.1.1.4. Optical Calibration Procedure 44
4.1.2. Robot Hardware Interfacing 46
4.1.3 Software Design 47
4.1.4 Functionality 51
4.1.5. The Servo Program 54
4.1.5.1. The Servo Routine 55
4.1.5.2. The Servo Planner 58
4.1.5.3. The Homing Routine 60

4.1.6. The Servo Agent 62
4.1.7. The Planner 64
4.1.8. The Reis Agent and the Remote Reis Programs 70
4.2. GM Fanuc S400 Motion Controller 71
4.2.1. Overview 71
4.2.2. Shared Memory 73
4.2.3. GMF1 Agent and Remote GMF1 programs 73
4.2.4. The Serial Program 74
4.2.5. The Encoder Program 75

Chapter 5. Seam Tracking Implementation 76

5.1. Kinematics of Seam Tracking 77
5.2. Coordinated Motion Control 80
5.3. MVS System Software 83
5.4. Application Software 84
5.5. Dynamic Seam Tracking Task Program 88
5.5.1. Overview 88
5.5.2. Software Implementation of Dynamic Seam Tracking 89

Chapter 6. Experimentation and Performance 98

6.1. Experimental Methodology 98
6.1.1. Experimental Workpieces 100
6.1.2. Variable Description and Ranges 102

6.2. Experimental Results & Data Analysis 107
6.2.1. Experiment 1: Position, Orientation and Seam Travel Speed 107
6.2.2. Experiment 1: Observations 110
6.2.3. Experiment 2: Stability with Respect to the Radial Distance 125
6.2.4. Experiment 2 Observations 126
6.2.5. Experiment 3: Gains and Transport Delays 134
6.2.6. Experiment 3 Observations 137
6.2.7. Experiment 4: Repeatability 151
6.2.8. Experiment 5: Alternate Wrist Setup 154
6.2.9. Experiment 5: Observations 155

Chapter 7. Discussion and Recommendations 157

7.1. Discussion 158
7.2. Limitations of Design 160
7.3. Recommendations 161

References 163

List of Tables

Table 3.1. The Status shared memory interface 34
Table 3.2. The Command shared memory interface 34
Table 3.3. The Sensor Status shared memory interface 35
Table 4.1. Link table for D-H convention 40
Table 4.2. The Status shared memory interface of the motion controller 50
Table 4.3. The Command shared memory interface of the motion controller 50
Table 6.1. Initial Exploration: Travel Speed and Position/Orientation 108
Table 6.2. Stability with respect to Radial Distance and Travel Speed 126
Table 6.3. Stability with respect to Gains and Transport Delays 135

List of Figures

Figure 1.1. Photograph of Mobile Autonomous Robot Stanford (MARS).[3] 3
Figure 1.2. Photograph of the Dynamic Welding System courtesy Huissoon and Strauss. . . 4
Figure 1.3. Photograph of Sojourner on Martian surface (NASA JPL 1996). 5
Figure 2.1. ABB S4C Industrial Robot Controller 14
Figure 2.2. A screen capture of Workspace in an arc welding application 15
Figure 2.3. The Reis Robotstar V15 in a seam tracking application 18
Figure 2.4.. The VMEbus chassis for the Reis Robotstar V15 18
Figure 2.5. Schematic of control system for one axis of the Reis Robotstar V15 19
Figure 2.6. The GM Fanuc S-400 robot arm manipulator 21
Figure 2.7. Computer Boards Quadrature Encoder 4 Channel Card (CIO-QUAD04) 22
Figure 2.8. The Laser/Camera assembly 23
Figure 2.9. The ISA processor board 24
Figure 2.10. Thrustmaster analog joystick 26
Figure 3.1. Overview of DROID 28
Figure 3.2. The three elements of a subunit 30
Figure 3.3. Message Passing for a robotic subunit 31
Figure 3.4. Comparison of Reis Robotstar V15 to GM Fanuc S400 32
Figure 4.1. Reis Robotstar V15 geometry 39
Figure 4.2. Reference frame assignment based on D-H convention 40
Figure 4.3. Geometric solution for angles 1,2,3 based on position only 42
Figure 4.4. Optical Calibration 46

Figure 4.5. Datagram of the Reis motion controller 49
Figure 4.6. Flowchart of motion controller operation 53
Figure 4.7. Flowchart of the Servo program 55
Figure 4.8. Flowchart of the Servo function 57
Figure 4.9. Flowchart of the Servo planner 59
Figure 4.10. Flowchart of the Homing function 61
Figure 4.11. Flowchart of the Servo Agent program 63
Figure 4.12. Planner program 65
Figure 4.13. Anatomy of the GM Fanuc S400 motion control software 72
Figure 5.1. Reference frames used in Seam Tracking software 78
Figure 5.2. Kinematic diagram 79
Figure 5.3. The Dynamic Seam Patch 81
Figure 5.4. Interpolation scheme for determining target seam position and orientation 82
Figure 5.5. The P1C30 Control program 84
Figure 5.6. Screen capture of the sensor.exe 85
Figure 5.7. Flowchart of the application software operation 87
Figure 5.8. Hardware setup for a Autonomous Multiple Robotic Welding system 89
Figure 5.9. Anatomy of the “Seamtrack” Task program 91
Figure 5.10. Flowchart of the Seam Tracking software operation 93
Figure 6.1. Seam reference frame 99
Figure 6.2. Step Input Experimental Workpiece 100
Figure 6.3. Ramp Input Experimental Workpiece 101
Figure 6.4. (i) Position within positive workspace and 0 degree Roll and Pitch 103
Figure 6.4. (ii) 300mm lateral offset of the track to position (i) 103
Figure 6.4.(iii) Same as position (i) but with a 30 degree Roll of the track 104

Figure 6.4. (iv) Same as position (i) but 45 degree Roll of the track 104
Figure 6.4. (v) Same as position (i) but 30 degree incline 105
Figure 6.4. (vi) Position and orientation along radial 105
Figure 6.4. (vi) Same as position (i) but vertical. 106
Figure 6.5: Step input response for all axis' with 30 degree Roll at 8mm/s 111
Figure 6.6: Step input response for all axis' with 30 degree Roll angle 112
Figure 6.7. Step input response for all axis' with changing position/orientation 115
Figure 6.8: Typical Seam Tracking system response to ramp input 118
Figure 6.9: Lateral Seam Tracking System response to Ramp Input Radius of 500mm . . . 119
Figure 6.10: Lateral Seam Tracking System response to Ramp Input Radius of 400mm . . 120
Figure 6.11: Lateral Seam Tracking System response to Ramp Input Radius of 300mm . . 121
Figure 6.12: Lateral Seam Tracking System response to Ramp Input Radius of 200mm . . 122
Figure 6.13: Selected X axis/Depth response to ramp input 123
Figure 6.14: Determining the Yaw and Pitch angles 125
Figure 6.15: Lateral stability along a radial line as speed increases 127
Figure 6.16: Lateral stability along a radial line as speed increases 129
Figure 6.17: Lateral/Yaw axis stability on a radial line at 16mm/s 131
Figure 6.18: Lateral/Yaw axis stability on a radial line at 8 mm/s 132
Figure 6.19: Depth/Pitch axis stability on a radial line at 16 mm/s 133
Figure 6.20: Pitch Gain Experiment, Depth stability with a step input 138
Figure 6.21: Yaw Gain Experiment, Lateral stability with a step input 141
Figure 6.22: Stability data for 1 cycle Transport Delay 144
Figure 6.23: Stability data for 2 cycle Transport Delay 145
Figure 6.24: Stability data for 4 cycle Transport Delay 146
Figure 6.25: Stability data for 5 cycle Transport Delay 148

Figure 6.26: Stability data for 6 cycle Transport Delay 148
Figure 6.27: Stability data for 8 cycle Transport Delay 149
Figure 6.28: Stability data for 16 cycle Transport Delay 150
Figure 6.29: Lateral/Yaw Axis data for Repeatability Experiment 151
Figure 6.30: Depth/Pitch Axis data for Repeatability Experiment 152
Figure 6.31: Standard deviation for Lateral data 153
Figure 6.32: Standard deviation for Depth data 153
Figure 6.33: Repeatability Experiment 154
Figure 6.34: Alternate Wrist Configuration 155
Figure 6.35. Alternate Wrist Setup Experiment 156

Chapter 1

Introduction

Sensor-based control of multiple robot systems requires a large number of sensors and robotic motor units to be integrated. In the past, these types of robot systems were limited to the mobile robotics research area or to the space and military industry. As for industrial applications, the use of multiple robotic work cells has been limited to robot units with programmed paths. Collision avoidance techniques for industrial robots which work in close proximity to one another usually comprises running the programmed paths through an offline simulation. Hence, coordination is not a real-time phenomenon in the industrial setting. Recent demands for agile manufacturing systems used primarily for low-volume, high accuracy manufacturing require that the cell be flexible in terms of both hardware and software. Hence, the ideal agile manufacturing cell should be able to add or remove sensors or robot manipulator units easily depending on the multiple tasks it can perform.

In the past, sensor-based multi-robot systems used distributed control to manage the multiple sensors and robotic units connected. These early distributed control systems used a single processor running a multi-tasking operating system to interpret the sensor data and then send higher-level move commands to each motion micro-controller. To make the most of the large array of sensors and to cooperate effectively, division of labor is needed in order to alleviate the computational load required to interpret all this incoming data and to coordinate the robotic units properly using a distributed parallel-processor model. According to Yasuda

and Tachibana [1] only a truly parallel multiprocessor control system could run a multi-robot system properly. However, such control systems using conventional processors cannot handle the full complexity of multi-robot systems due to the following reasons:

> i) in addition to event driven scheduling, round robin scheduling, where the period of task switching is desired to be shorter than the time required for processing one motion command, is needed for two robots to move simultaneously;
>
> ii) the sampling interval for robot motion control largely depends on the time required for processing of one motion command; and
>
> iii) executable code is inefficient and the safety cannot be proven because control software is written as polling loops or with interrupt routines.

At the time, conventional processors were not very powerful; hence multitasking performance consisted of non-deterministic delays due to the above reasons. However, recent improvements in microprocessors, network technologies and real-time operating systems have allowed distributed control systems using conventional processors to approach performance levels to that of parallel systems. Thus a Distributed Robot of Intelligent Devices (DROID) was developed based on the recent advances in computer hardware and software technology to further research in the area of sensor-based multiple robotic systems [2]. The conception, design and implementation to the application of autonomous multiple robotic welding is the topic of this thesis.

1.1. Sensor based Robotics Research

Early sensor-based robotics research focused primarily on mobile robots and the task of navigation. These early mobile robot platforms such as the Stanford Robot (known as the Mobile Autonomous Robot Stanford , MARS) used primitive distributed control [3]. The control system consisted of a single National Semiconductor 32016 16-bit processor to act at the

coordination level. Sensors consisting of 12 Polaroid acoustic sensors, bumpers, pan/tilt sensors and two video cameras were linked to this microprocessor. Three 8-bit micro-controllers were responsible for driving/steering the platform and provided encoder/odometry data. This system made attempts to build a model of the world throughout its path, resulting in high computation loads prior to each move.

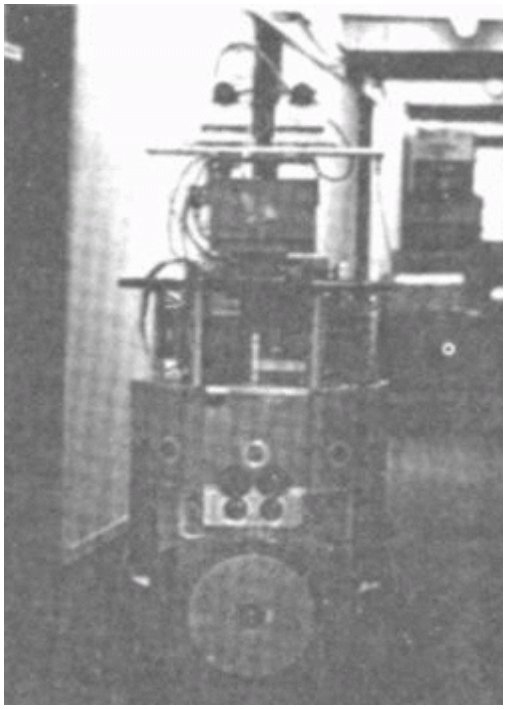


Figure 1.1. Photograph of Mobile Autonomous Robot Stanford (MARS) [3].

Early industrial autonomous robotics research was pursued with emphasis on intelligent agile robotic work-cells for low volume and high precision manufacturing. Strauss and Huissoon in 1991, [4], [5], [6] , developed an integrated sensor based dynamic welding system which followed a weld seam using computer vision techniques and manipulated the position of the jigged workpiece to allow optimum welding parameters. This system was autonomous in that it only required the tool-robot to be positioned along the seam; once the welding began, it created a dynamic piece-wise model of the workpiece and continued without further

programming until the weld was complete. This system was limited to a 6 DOF robot with a 2 axis workpiece positioning system. It demonstrated that with conventional hardware and a distributed control system, autonomy could be achieved using a sensor reaction- based control paradigm. However, due to technological limitations, heavy optimization was necessary to achieve this level of control and coordination between the robot controller and the sensor computer; hence it became very specialized for the task of welding and reconfiguration was limited.

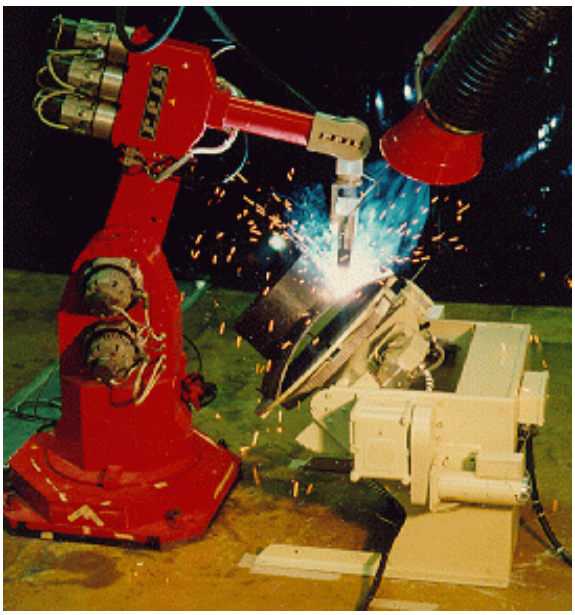


Figure 1.2. Photograph of the Dynamic Welding System
Courtesy Huissoon and Strauss (1991).

Modern autonomous mobile robots such as Sojourner developed at NASA JPL and Carnegie Mellon University [7], for exploration of Mars have a more distributed approach to robot control. Obstacle avoidance and local navigation is handled autonomously based on sensor data reaction. Since Sojourner is wheeled with active suspension (wheels can be lifted to overcome obstacles) the kinematics are quite simple and can easily be handled by a single controller. This controller communicates with tilt sensors, gyroscopes and move commands using network messaging. The move commands are calculated using stereoscopic cameras for

obstacle avoidance as well as supervisory navigation commands from mission control. This distributed robot architecture has a proven record based on the success of 1996 Mars Pathfinder mission and should be extended to industrial multiple robotic systems.

Figure 1.3. Photograph of Sojourner on Martian surface (NASA JPL 1996).

Research in the area of distributed control architectures for multiple robot systems were pursued in the early 1990's. These research efforts strove to make the robotic system reconfigurable and modular so as to facilitate autonomous sensor-based research for both mobile and industrial robotics. Stewart et al [8] in 1992 at Carnegie Mellon University's Robotics Institute developed Chimera, a real-time operating system designed for advanced sensor-based robotic applications. This operating system ran on workstation-class Sun Microsystems hardware and allowed both robot and sensor modules to be used in a very modular fashion. Gertz et al [9] in 1994 added a graphical user interface called ONIKA to represent the robot configurations where icons were used to represent and monitor each robot or sensor element.

Although this system has a rich history and existing application libraries, it remains a custom operating system originally designed for Motorola-based hardware in the early 1990's. This research shows that the need for such a system was identified early in the 1990's; however the computing technology was not mature enough. This led the research group to develop their own real-time operating system and develop their own set of libraries and device drivers for the robot systems of that epoch.

Yasuda and Tachibana [1], [10] in 1996 suggested the need for parallel multi-robot control architectures and specified a computer network based control architecture based on modeling each autonomous robot unit as an object oriented Petri net. A Petri net is a directed graph with two kinds of nodes (places and transitions) such that no two places or transitions can be connected directly (a state machine can be considered a subclass of a Petri net) [11]. The authors modeled each robot task or discrete event as a Petri net transition and implemented this object oriented modeling in a simulation based on a transputer network. They showed that the model of parallelism and synchronous communication can work efficiently within this type of system.

The MARTHA project was developed by Alami et al [12] in 1998 at the Robotics and Artificial Intelligence group at LAAS/CNRS; its focus was on high level mobile robotic cooperation and coordination for the application of navigation. Using a shared database of information, each robot plans its path to avoid each other. These 3 robots operating using WindRiver VxWorks were used to validate the system. Simulations were run using up to 10 emulated mobile robots in a simulated environment to assess the scalability of their system. This system shows the promise of using multiple mobile robot systems in high traffic areas for collision avoidance. The study also shows the importance of intercommunication between robots and the importance of the integrity of a central database for the robots' positions.

Jung and Zelinsky [13] in 1998 at the Australian National University's Robotic Systems Laboratory designed an architecture for distributed cooperative planning in a behavior-based multi-robot system called ABBA. ABBA is a task independent architecture that supports learning, and action selection. It serves as a general framework whereby a collection of simple

behaviors or tasks can be embedded, such as navigation, planning, cooperation and communication, so that a complex tasks can be built up. Using a network of cameras and robots, they were able to build up simple behaviors to accomplish a complex task such as cleaning a room. A rule-based action-selection algorithm ties the navigation of the room with picking up litter and dumping it into a dust-bin. A similar architecture was developed at MIT's Artificial Intelligence Laboratory by Parker in 1998 using mathematical models for motivation to achieve adaptive action selection for each robot [14]. These high-level architectures demonstrate that complex autonomous tasks are a viable pursuit, providing that the underlying robots and sensors can be abstracted into intelligent agents. For the case of ABBA, the robot units and sensor were built up using WindRiver Systems VxWorks real-time operating system.

Very recent work at the University of Coimbra [15], shows investigation of a distributed network of robots connected on Ethernet. This research effort uses the latest ABB S4 controller which supports Ethernet and the TCP/IP protocol. High level commands are sent to the controller from an interface computer via remote process commands (RPCs). These RPCs are preprogrammed behaviors which can be triggered by a signal sent from the central computer. The interface computers are Microsoft Windows 95/98/NT/2000-based and the robot units appear in the Windows Environment as ActiveX communication objects. The authors have conducted some performance tests for network delays and have shown that delays as long as 20ms exist for Ethernet TCP/IP initiated RPCs to take action. For high-level coordination this may be acceptable for RPCs; however, real-time sensor-based control requires reaction delays which are much less (at least 10ms to achieve 60Hz sensor integration).

1.2. Rationale and Objectives

Any of the research efforts of the past have been limited due to technological restrictions. Sensors were expensive to build and integration was complicated. In addition, computing power was expensive and interconnect technologies were slow. Hence the amount of human effort

required to build autonomous systems was significant.

In recent years, the explosive growth of the consumer computing industry has made the technologies which were once vital to the autonomous robotics field into commodities. Lasers, cameras, microphones, rangefinder technology, GPS technology and accelerometers have been miniaturized drastically due to the consumer electronics industry. These sensors such as the CCD cameras and accelerometers are available on a single integrated circuit.

High speed inexpensive processors which surpass the performance of those found in supercomputers of the mid 1990's are now available in every personal computer. Memory technology has advanced to the point that entire programs and operating systems can be stored in RAM itself without the need for external storage. This RAM is also at commodity prices and at access speeds which were unheard of 5 years ago. Hard drive capacities now exceed even the most demanding applications. All these advances mean that computationally expensive control algorithms can be attempted using very inexpensive computer hardware.

In addition, the proliferation of the Internet has both advanced network technologies as well as allowing globalization of our knowledge base. Ethernet cards, Hubs and Routers are available at commodity prices offering speeds of up to 100Mb/s. Future Ethernet cards will have bandwidths in the Gb/s range allowing very rich data streams to be sent. The Internet has also given rise to alternative operating systems built primarily with the efforts of the Open Source Movement. This movement believes that software source code should be made available to everyone so that it can be shared and modified for anyone's needs. This has led to the emergence of the Linux Operating System, an Open Source and free implementation of the UNIX operating system. Due to the large momentum behind Linux, many hardware vendors who formerly would not reveal their hardware specifications are now forced to do so to allow adequate driver support of Linux [16]. Thus we are no longer limited by the hardware manufacturers in our integration of their hardware for our systems.

In order to advance autonomous robotic systems research at the University of Waterloo a stable, modern and modular architecture was developed for multiple robots which are sensor-driven. This robot architecture is called DROID which stands for "Distributed Robot of

Intelligent Devices". It features a distributed robotic architecture where modular intelligent devices are connected to a central intelligence computer (Brain Computer) which coordinates its behavior via a high speed Ethernet network. This system is reconfigurable and promotes plug and play performance for the devices such as robot manipulators, sensors and propulsion units. By using a modern operating system, this system also has the advantage of being remotely controlled on the Internet using network software technologies such as TCP/IP, Sockets and JAVA. In addition, it is based on commodity-priced personal computer technology; therefore it can remain at the cutting edge and still be fairly inexpensive.

Although one can design such an architecture, it must be realized through an application to prove its effectiveness. In our case, the DROID system was implemented in the industrial setting for the application of sensor-based autonomous multiple robotic welding. In particular, we have developed a two robot work-cell which utilizes a laser profiling sensor to dynamically seam track for welding applications. Robots and sensors were extensively modified to be used in the DROID system.

This thesis describes the conception of the DROID system, its system architecture design, the software developed to successfully perform the task of Autonomous Multiple Robotic Welding, its performance and the future of the DROID system itself.

Chapter 2

Hardware and Software Technologies

This chapter focuses on the technologies used to create the DROID system for the application of Autonomous Multiple Robotic Welding. Much of the design of the DROID system is based on advances in computer hardware. This chapter begins with a background description of the existing computer technology at the time of inception, and later robotic technologies are discussed. This is followed by a brief description of the actual hardware used in the DROID implementation which includes the i) welding robot, ii) the workpiece positioning robot, iii) the sensor, and the coordinating Brain Computer. The objective of this chapter is to give the reader insight into the rationale for choosing certain hardware and software technologies.

2.1. Current Computer and Robot Technology

2.1.1. Computer Technology

Technology moves especially fast in the computer world and if we design complex systems based on the current technology, we find that our systems become obsolete by the time of deployment. With this in mind, we must look ahead to the future and forecast the trends in technology. From these trends we can decide on using certain technologies that can be upgraded in the future so that our systems remain at the cutting edge. This section attempts to paint a

picture of the current technologies which guides our system design.

The conception of the DROID system began in early 1998. At the time, Intel Corporation had just released its Pentium II processor. The Pentium II processor would replace its slower predecessor by offering almost twice the processing power at the equivalent clock speed; as well, the clock speed would increase up to 450MHz compared to the 233MHz speed of the Pentium. With the increased clock frequency, so increased the operating temperature, and reliability in the absence of active cooling decreased; hence the Intel Pentium II was marketed primarily as a workstation-class processor.

Intel processor clone companies such as Advanced Micro Devices (AMD) and Cyrix developed CPU's which mimic the x86 instruction set of an Intel Pentium processor and which would run cooler. With the coming of the Pentium II and the proliferation of Pentium clones, this drove the price of Pentium-class processors down. Eventually, these Intel Pentium-class processors found their way into industrial applications which had formerly been dominated by Motorola processors.

In the mid 1980's the Motorola Corporation was a dominant force in the microprocessor industry, clearly at the leading edge, delivering well designed Complex Instruction Set Computer (CISC) processors with rich instruction sets and compilers (the 680X0 family). Both industrial and military system integrators chose Motorola processors along with their peripherals and bus architecture. Conversely, Intel's dominance of the personal computer market allowed it to proliferate its architecture designs onto desktops everywhere. Eventually, Motorola's technological lead would be eliminated as high power, low cost Intel processors became dominant. By the late 1990's integrators were looking to Intel instead for lower-cost computing systems destined for industrial and military applications.

The processor platform of the DROID system is the Intel Pentium-class processor. Due to their low-cost, abundance and exceptional floating point mathematics performance, they provide a stable architecture with which to build future software control systems. In addition, the Intel architecture has an abundance of peripherals allowing us to build very complex control systems with off-the-shelf components. However, as Intel processor modules are being adopted into existing industrial and military systems, they must interface to the existing Motorola

peripherals.

In the mid 1980's, with the adoption of the Motorola processors into military and industrial applications, the demand for peripheral devices for these applications grew. Since Motorola could not supply this niche market themselves, they set out to develop an open bus standard compatible with their processors. This bus standard known as the Versa-Modula-Europa bus (VMEbus) standard was developed in conjunction with other circuit board manufacturers. This allowed any third party board manufacturer to interface to any VMEbus system. Therefore, very modular, complex and upgradeable systems could be built, with manufacturer independence. The VMEbus architecture was very similar to the 68000's bus architecture. It began as a peripheral memory-mapped architecture with a 20-bit addressing space. Later this was increased to 32-bits and eventually to 64-bits. Many of the military and industrial applications which utilized VMEbus are still in operation today.

At the conception of the DROID system, the VMEbus architecture was still very much in use both in military and industrial applications. The new Intel bus architecture at the time was the Intel Peripheral Computer Interface (PCI) whose bus controller was embedded into every Intel Pentium processor support chip [17]. Hence all Pentium-class processor computers came equipped with a PCI bus. With a 32-bit address space and a transmission bandwidth of 133Mb/s it was proclaimed as the future high-performance bus. Since the Intel bus architecture has fundamental differences with the VMEbus system, bridging technology is required to integrate Intel processor modules with VMEbus systems. Fortunately, the Tundra Semiconductor Corporation built a PCI-to-VME bridge integrated circuit (Universe and Universe II) which made memory transfers from the PCI bus system to the VMEbus system (and vice-versa) transparent [18].

Industrial settings present adverse and demanding conditions for interconnect technologies. This resulted in a plethora of interconnect technologies and protocols being used in industry such as RS-232, RS-485, DeviceNet, ControlNet, PROFIBUS, and many other proprietary networking technologies all promising low signal to noise transmission ratios and high bandwidth. RS-232 is a simple point-to-point twisted pair serial network protocol with transmission speeds of up to 56Kb/s. RS-485 offers multiple connections of up to 128 nodes

with a shared bandwidth of 10Mb/s. DeviceNet allows 64 nodes to be connected with a bandwidth of 500Kb/s and a packet size of 8 bytes. ControlNet offers a bandwidth of 5Mb/s. PROFIBUS is a network protocol which rides on top of Ethernet or RS-485 delivering messages at a maximum length of 246 bytes. Most of these network technologies with the exception of RS-232 and RS-485, require costly interface cards. In addition, the cost and lack of development tools/libraries make these technologies very unattractive [19], [20]. The overwhelming presence of the Internet has made very low-cost and robust ethernet interfaces available. As well, the dominance of TCP/IP has given integrators a very stable and robust code base. With speeds starting at 10Mb/s with well over 1Gb/s transmission rates in the future, it is the ideal interconnect for any application. The DROID system utilizes Ethernet as the network interconnect coupled with a high-speed Native Message Passing protocol built into the QNX Operating System kernel [21].

The emphasis of our system design is to use computer technology which is readily available from the Personal Computer market. By using this type of hardware, we incorporate leading edge technology very quickly into our system and reap the tremendous benefits of high performance, and high storage capabilities.

2.1.2. Robot Technology

Many of the advances in the industrial robotics industry have been in the level of integration of the robotic controller. The mechanical designs of the 6 degree of freedom robots developed in the late 1980s have had little change. The controllers, however, have been drastically affected by computer technology, resulting in a reduction of the overall size of the controller. Faster processors have allowed a degree of collision detection to be implemented at the controller level. In addition, the advent of the personal computer has allowed programs to be written offline and downloaded to the controller using very standard interfaces such as RS-232 and Ethernet.

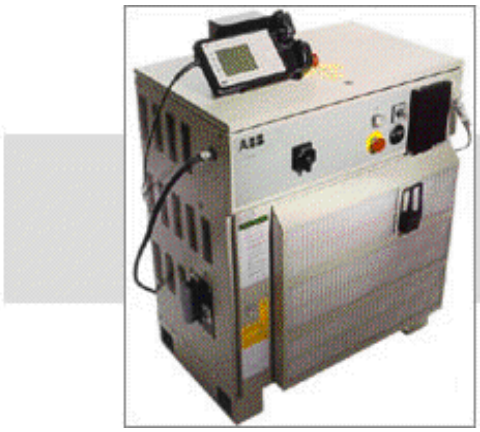


Figure 2.1. ABB S4C Industrial Robot Controller

Although the recent advances in industrial robotics have been in the area of ease of use, they remain proprietary in their modes of communication. Each robot manufacturer has its own preferred language (Fanuc Karel, CRS RAPL 3, Motoman Inform II, etc.), and its modes of communication remain proprietary although they may share the same medium (Ethernet, or Fieldbus). In addition, the integration of sensors to update the robot's path in real-time still requires extensive integration.

Robotic visualization tools are available such as Workspace (Flow Software Technologies, Windsor, Ont.) which allow multiple robotic units to work together. These tools allow multiple robots to be programmed offline using a single program; however, cooperation among robots is not a real-time task. In addition, sensor integration in these simulations is limited to extremely simple devices such as limit switches. These visualization programs must also support a multitude of robot languages in order to be a viable tool in any industrial setting.

The emphasis of our system design in terms of advancing robot technology is to provide a generic interface to existing robots, propulsion units and sensors. This generality allows universal communication to exist and cooperation among these units. Thus, very complex multiple robotic systems can be built up, programmed and monitored remotely using advanced network technologies.

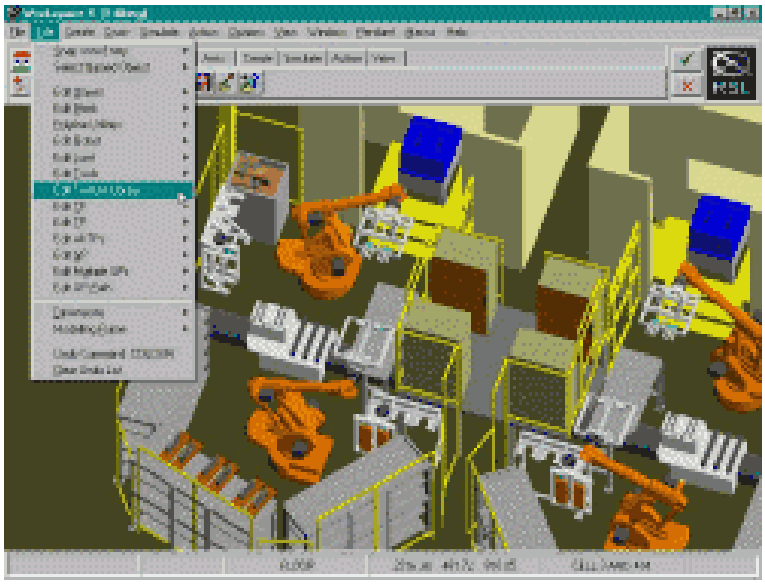


Figure 2.2. A screen capture of Workspace in an arc welding application

2.2. Operating Systems

Once the hardware platform had been established for the DROID system, a suitable software platform or operating system to build our software on had to be chosen. Using Intel x86 hardware allows the use of almost any operating system. However, not every operating system is suitable for multiple robotic control due to the inherent designs of the operating system.

Control of the DROID system requires that signals be read and written at a fixed rate; any fluctuations in the control rate gives rise to instabilities and unpredictability in the control system. Hence, we must have complete control of the operating system, in that we must be able to preempt the operating system kernel with our own control program; this is known as "Hard Real-Time Performance". A limited number of operating systems exist which provide this capability due mainly to security issues.

To connect the many nodes involved in the DROID system, a solid network platform must be available to the operating system. This includes stable device driver software for

modern network adapters as well as solid network protocol libraries (i.e. TCP/IP, WINSOCK, etc). Most Unix-like operating systems also have support for a Message Passing Interface protocol or Parallel Virtual Machine protocol. This is a high speed low-level network protocol which makes a cluster of machines linked by Ethernet appear as a single parallel processing virtual machine.

The DROID system must be able to access multiple devices or sensors which may not be supported by any operating system. Therefore, we must be able to build device drivers for the sensors and robots which we attach to our system. Since hardware programing is a very tedious task, we must choose an operating system which is easy to develop hardware interfacing software on. Although many operating systems offer high-level application development tools that "almost write by themselves", these operating systems may not grant access to any particular device including a physical memory address.

Lastly, we must consider the portability and the future of the code base which we develop. Operating systems can be completely abandoned very quickly if the software industry will not write programs for it, although it may be perfectly suitable for many applications. In addition some operating systems are so proprietary, a simple upgrade to the operating system may render a crucial operating system component useless. Hence, we must choose an operating system with both a strong lineage to the past such as a POSIX-compliant operating system (i.e. UNIX variant), and which has a promising future.

Although very prominent, Microsoft Windows 95/98/NT/2000/ME and UNIX do not allow application programs or even device drivers to be written with privileges higher than the operating system kernel; hence they are considered non-real-time operating systems. Only very recently (late1998), specialized kernels have been developed for Windows NT and LINUX (an Open Source Unix Implementation) which provide a degree of real-time control. However, at the time of inception of the DROID system, they were still in alpha stages of development.

Wind River System's VxWorks, MTOS and Realtime C Environment provide operating system libraries which allow the control program to be compiled into a monolithic executable to be downloaded at run-time. Since the program code is compiled with the operating system, a high degree of control is retained and is considered real-time. However, their offline

development method makes the development of hardware interfacing software somewhat tedious.

Microsoft DOS and QNX 4.25 provide an environment to program applications which can preempt the operating system. Software is usually developed on the actual hardware target, hence it is termed host-based development. However, due to the short-sighted design in DOS (i.e. the 640KB conventional memory limitation) it was abandoned in the mid 1990's by the Microsoft Corporation. QNX 4.25 is a 32-bit POSIX compliant real-time operating system. It is a true multi-tasking operating system which provides low-level access to hardware resources. The operating system kernel is both powerful and small and it allows applications to preempt it based on priority levels. It also has a powerful network layer that allows powerful distributed systems to be built [21].

At the heart of QNX 4.25 is a Message Passing system which allows processes to communicate with each other. Hence, a QNX 4.25 application consists of many small programs which provide a dedicated service and are debugged extensively. These small programs then communicate with each other via Message Passing to become a system. Hence, the code base is very modular and can easily be upgraded. Therefore it was chosen as the operating system for the DROID system.

2.3. The Reis RobotStar V15

The Reis Robotstar V15 is a 6 revolute joint manipulator arm manufactured by Reis Machines Inc. in the late 1980's [22]. It was purchased by the Department of Mechanical Engineering of the University of Waterloo as a research robot. The key advantages of the Reis Robotstar V15 are simple robot kinematics and ease of upgrading due to the use of the VMEbus architecture as a back-plane for the robot controller. Figure 2.3. shows the 6 links and revolute joints of the Reis Robotstar V15 while Figure 2.4. shows the VMEbus chassis which contains the processor board and Input/Output boards.

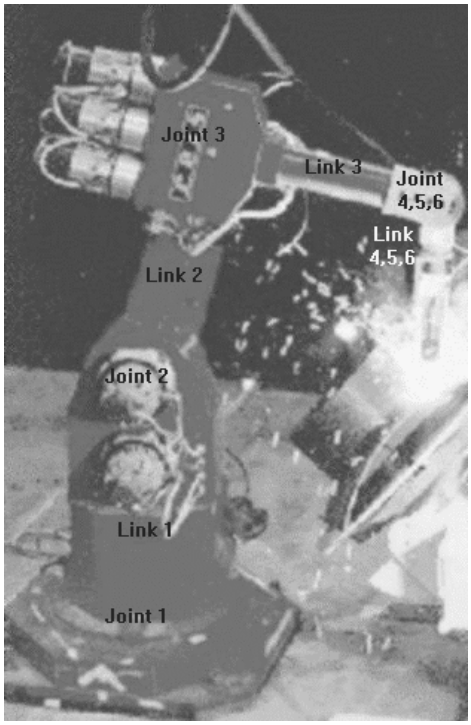


Figure 2.3. The Reis Robotstar V15 in a seam-tracking application

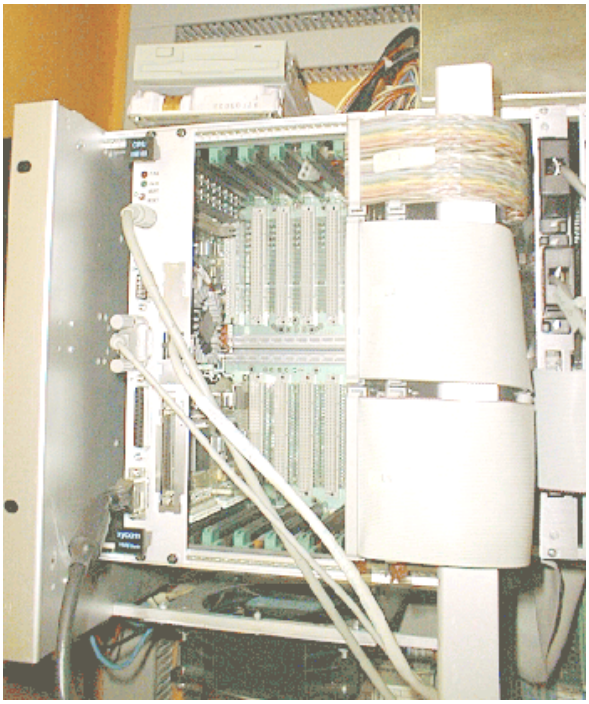


Figure 2.4.. The VMEbus chassis for the

Reis Robotstar V15

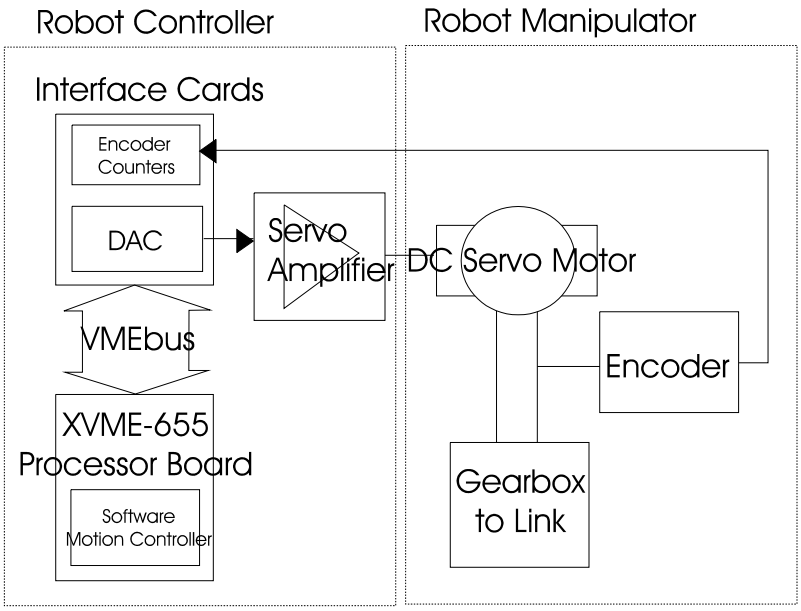


Figure 2.5. Schematic of control system for one axis on the Reis Robotstar V15

Figure 2.5. shows the control scheme for a single axis on the Reis Robotstar V15. Each of the six joints uses a harmonic geared DC-servo motor which is powered by an independent servo-amplifier. This servo amplifier obtains its command input signal ($\pm$5VDC) from the I/O board which has a digital to analog converter (DAC) for each of the 6 joints. These DACs are mapped into the VMEbus memory space. An encoder resides on each DC-servo motor providing an encoder signal to the I/O board encoder counter, again these encoder counters are mapped into VMEbus memory space. Hence, a closed-loop controller for the 6 joints can be implemented in software by reading the encoder memory locations and writing to the DAC memory locations. In addition to the DC-servo motors, a pneumatic piston is used to balance link 2, however, this is considered a passive balancing system for the Reis Robotstar V15. One further note, as a safety mechanism, the presence of a Watchdog timer on the VMEbus Interface Boards requires that the Watchdog timers be reset with a trigger prior to expiry; otherwise the power amplifiers are disabled, and no further motion is possible without resetting the power system. Therefore, a Watchdog module is executed at startup to trigger these timers periodically. If a system failure occurs, this module would be delayed and the timers would

expire and cause a shutdown.

Originally the Reis Robobstar V15 came equipped with an Elan Motorola 68000 processor board running a simple control program burned on EPROMs (Erasable-Programable Read Only Memory) running at 8MHz. In its stock form the Reis Robotstar V15 cannot provide the high performance and sensor integration needed to fulfill its role as a research robot. To upgrade the Reis Robotstar V15 to work with the DROID system, only the processor board was replaced with a Xycom XVME-655 processor board, since the I/O boards provided by Reis Machines are more than adequate for interfacing to the robot itself.

The Xycom XVME-655 Processor Board is a complete personal computer (PC) with VMEbus addressing capabilities. Manufactured by Xycom Automation Inc., [23] it contains the following:

-Intel Pentium MMX processor operating at 200MHz
-64MB of Extended Data-Out RAM
-IDE disk controller attached to a 4GB hard drive
-floppy controller
-PS-2 keyboard port, parallel, 2 serial ports
-XGA graphics adapter
-100TX capable Ethernet adapter
-Ultra-Wide SCSI adapter
-Tundra Universe II PCI-VME bridge chip.

The XVME-655 has all the capabilities of a desktop PC with the ability to access VME peripheral cards. Hence any operating system designed for the PC could be used on the VMEbus as long as device drivers are available for the Tundra Universe Chip [18].

2.4. The GM Fanuc S400

The GM Fanuc S-400 is a 6 revolute joint manipulator arm manufactured by Fanuc Robotics Inc in the late 1980's for General Motors [24]. Its large size and extensive reach make it an ideal robot for heavy assembly duty in very harsh industrial environments. The Fanuc S-

400 has simple robot kinematics similar to the Reis Robotstar V15 with the exception of the robot link lengths. Unlike the Reis Robotstar V15, the GM Fanuc S400 has a very proprietary controller. Powered by a single Motorola 68000 8MHz processor, it performs adequate motion control with the option of external communication via a RS-232 serial port. All aspects of high-level controls are accomplished using Fanuc KAREL interpreted programs. Figure 2.6. identifies the 6 links and revolute joints of the GM Fanuc S-400 robot arm manipulator.

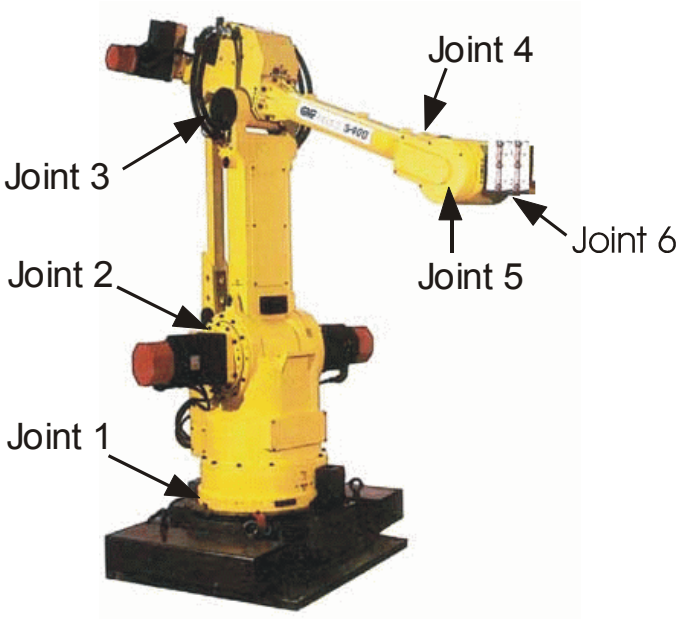


Figure 2.6. The GM Fanuc S-400 robot arm manipulator

Due to the proprietary bus architecture of the Fanuc S-400 controller, extensive hardware intervention is required to build an adequate interface to the DROID system. For Phase 1 of the integration of the Fanuc S-400 to the DROID system, a less invasive interface was attempted. Since the Fanuc S-400 is used primarily as a work-piece positioning system and not for welding, only the position and orientation of the Wrist-Center-Point is required. Gross movement commands can be sent to the Fanuc S-400 via the low bandwidth RS-232 interface. To access the position of the Fanuc S-400, two 4-channel ISA quadrature encoder cards are used (CIO-QUAD04, Computer Boards Inc. Middleboro, MA). They are attached to the encoder signals

located on each joint controller. Figure 2.7. shows a photograph of the Computer Boards CIO-QUAD04 Encoder Card. These cards are hosted on a separate computer which runs the QNX 4 real-time operating system and contains an Ethernet card which is in turn connected to the Brain Computer via the Ethernet connection.

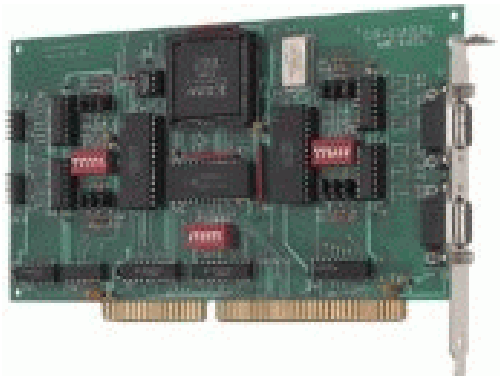


Figure 2.7. Computer Boards Quadrature Encoder 4 Channel Card (CIO-QUAD04)

The host computer for the GM Fanuc S-400 DROID System interface contains the following equipment.

-Intel Celeron processor operating at 466MHz
-64MB of Extended Data-Out RAM
-IDE disk controller attached to a 4GB hard drive
-floppy controller
-PS-2 keyboard port, parallel, 2 serial ports
-XGA graphics adapter
-100TX capable Ethernet adapter
-2 CIO-QUAD04 Encoder Cards

This is a typical Intel x86 personal computer running the QNX 4 real-time operating system, hence inexpensive hardware can be used to build the interface to the DROID system.

2.5. The MVS Line Laser Sensor

The MVS Line Laser Sensor is a profiling system developed by MVS Inc. which casts a laser line onto a surface and captures the reflected image using a CCD camera adjacent to the line laser generator [25]. Connected to the laser/camera assembly is the ISA processor board which processes the incoming image and filters it digitally to reveal the feature. Using host-based software, further feature analysis can be accomplished.

The laser/camera assembly is an independent system. The laser is an infrared laser diode equipped with a cylindrical lens which produces the line. The camera provides a 512X480 pixel grayscale image which is refreshed at 30Hz and is RS-170 compliant.

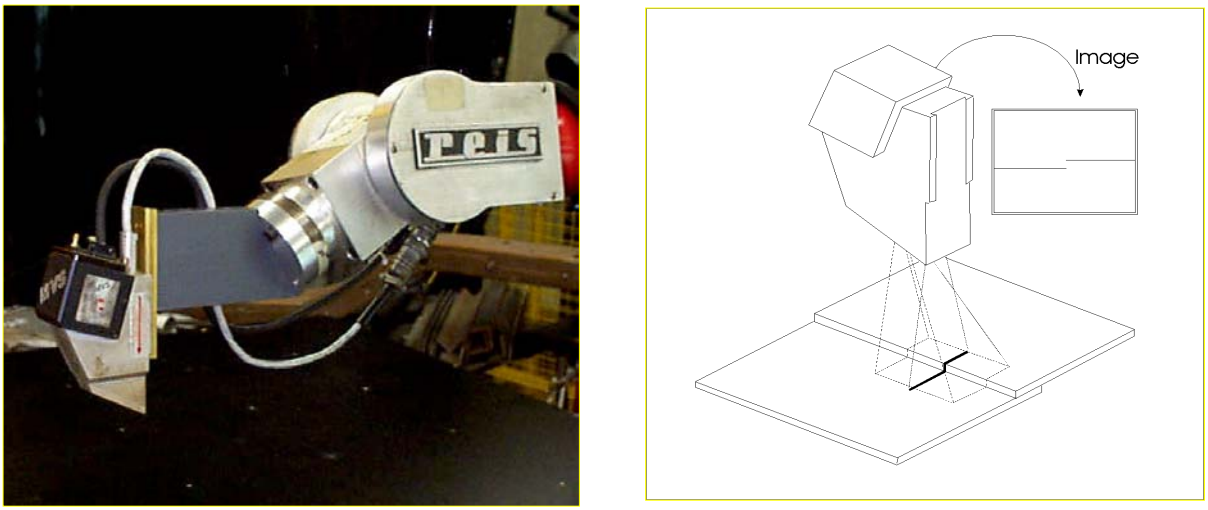


Figure 2.8. The Laser/Camera assembly

The ISA processor board is a proprietary board designed by MVS Inc.; it is equipped with an IMSA 100 RS-170 camera interface and memory. Unfortunately, the only working device drivers which exist are for the Microsoft DOS 16-bit Operating System. Therefore, this system will only function on an Intel x86 personal computer equipped with at least 2 ISA expansion slots. Currently, the MVS host computer is an Intel 486 ISA-bus personal computer with a 500MB IDE hard drive running MS-DOS 6.22. This computer is also equipped with a

serial port via which the processed feature location is exported to the Brain Computer.

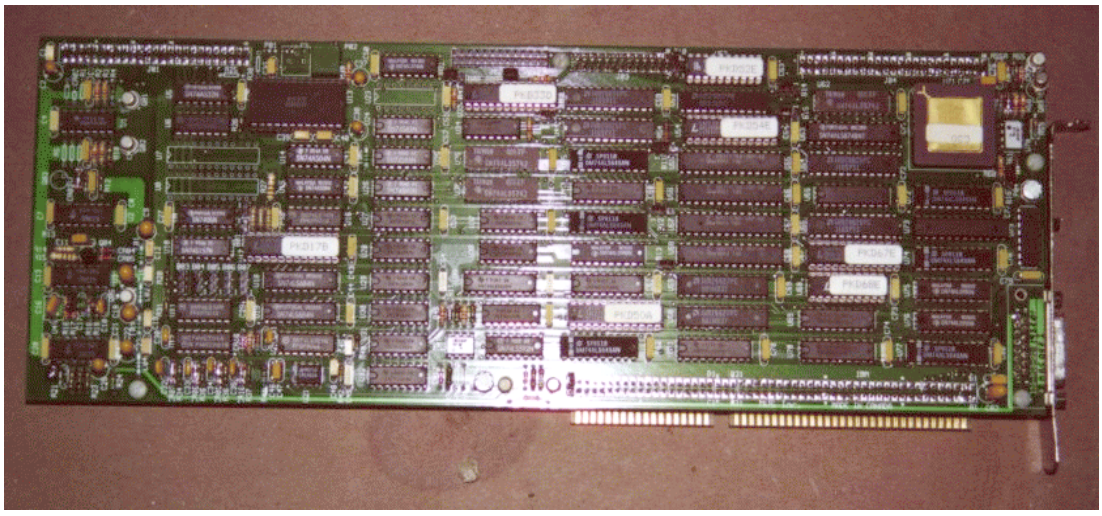


Figure 2.9. The ISA processor board

To determine the feature location such as the center of the seam to be welded, a host-based program must be written to i) access the filtered image on the processor board, ii) determine the feature location using statistical and numerical methods, and iii) to translate this location into real world coordinates. This program is developed using both Microsoft Assembler and Microsoft QuickBASIC 4.5. Assembler routines are used for high-speed low-level access of the frame buffer while QuickBASIC routines are used for statistical computation and data translation.

2.6. The Brain Computer

Like the brain in living systems, the Brain Computer acts as the main information processing unit. It must interface to the numerous inputs (sensor units) and outputs (motor units) of the DROID system. Once interfacing is established to the sensor and motor units, memory caches are created to house the streams of data of each unit. These streams of data can then be

processed to accomplish an autonomous coordinated task or maneuver. The actual equipment which makes up the Brain Computer is as follows:

-Intel Pentium II processor operating at 333MHz
-64MB of SDRAM
-ATA-33 disk controller attached to a 5GB hard-drive
-PS-2 keyboard port, 1 parallel, 2 serial ports
-PCI XGA graphics adapter
-2 100TX capable Ethernet adapter
-1 Computer Boards CIO-DAS08 Data Acquisition Card
-1 Thrustmaster analog joystick

To interface to the sensor and motor units of the DROID system, a QNX network interface is required. The Brain Computer runs the QNX 4 real-time operating system and is equipped with a supported PCI 10/100Mbps Ethernet card. This allows the Brain Computer to be connected to up to 255 units/nodes on the QNX Local Area Network. If more units/nodes need to be attached, multiple Ethernet cards can be equipped on the Brain Computer to act as a bridge to another QNX network.

To provide access via the Internet, an additional PCI 10/100Mbps Ethernet card is equipped on the Brain Computer. Since the TCP/IP protocol is supported by QNX, socket-based communication can be established with any machine on the Internet. Hence, the Brain Computer can be considered as an Internet Gateway which can provide bi-directional access to any sensor/motor unit in the DROID system from any Internet computer in the world.

To attach to legacy devices, the Brain Computer has 2 serial ports and a parallel port which is a standard on conventional Intel x86 personal computers. QNX provides access to these devices through its own device drivers and mounts these devices onto the file system. Hence access to these ports can be achieved using simple file access methods.

Since certain devices do not come equipped with a serial/parallel/USB/Ethernet port, interface analog voltages and TTL logic is supported on the Brain Computer using a Computer Boards Data-Acquisition Board (CIO-DAS08, Computer Boards Inc.). This allows simple devices such as welding controllers or simple work-piece positioning systems to be used in the DROID system. A modified Thrustmaster analog joystick is currently attached to the DROID

system using this data-acquisition board which provides 2 analog axis of control and 8 digital input channels.

The Brain Computer also acts as a development machine and central repository of the DROID system's software. Hence it is equipped as a graphics terminal with multiple window support provided by the Photon Windows Manager of QNX. To run the Photon Windows Manager, a supported video graphics adapter is required. For actual code development, the Brain Computer is equipped with a WATCOM C/C++ Compiler Version 11 (Waterloo, Ontario) and a host of Unix ported development utilities such as GNU make, vi, emacs, cvs, etc.

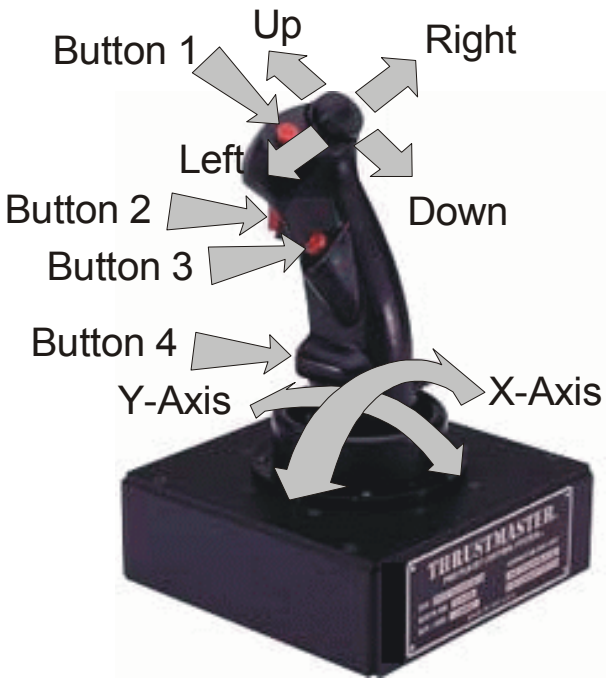


Figure 2.10. Thrustmaster Analog Joystick

Chapter 3

System Architecture

This chapter focuses on the system design/architecture of the DROID system. It begins with an overview of the DROID system. It then proceeds to explain the network protocol used to establish communication between each subunit with the Brain Computer. This is followed by a description of what a subunit must fulfill in order to connect to the Brain Computer. Lastly, we discuss the task programs used to coordinate the various sensor subunits with the motor subunits. This chapter's objective is to give the reader an understanding of the DROID system's functionality. However, the actual implementation of the functions described is left for the next chapter.

3.1. Overview

The paradigm of the Distributed Robot of Intelligent Devices (DROID) system is to have a system architecture which can extend the robotic system for any given task while having the complexity hidden. The addition or removal of robotic arms or sensors should be transparent to the system integrator. In addition programming task programs should be simple due to the abstraction of the underlying hardware.

To achieve this flexibility, we use "life" or "nature" as a model. Central to our system is a "brain", which plays a supervisory role in our integrated system. Each "subunit", such as

the robotic arm, sensor or propulsion unit is connected to the Brain Computer using a high speed interconnect, in this case, 10Mbps Ethernet. In order for a "subunit" to be made available to the Brain Computer, it must be able to communicate in a standard method. It is through this standard communication method that status and command data are related from subunit to the Brain Computer. These status and command data are held in a database on the Brain Computer and are refreshed at a fixed clock rate governed by the "heartbeat". Therefore, a task program need only deal with accessing the database on the Brain Computer in order to perform a coordinated task between multiple robots and sensors.

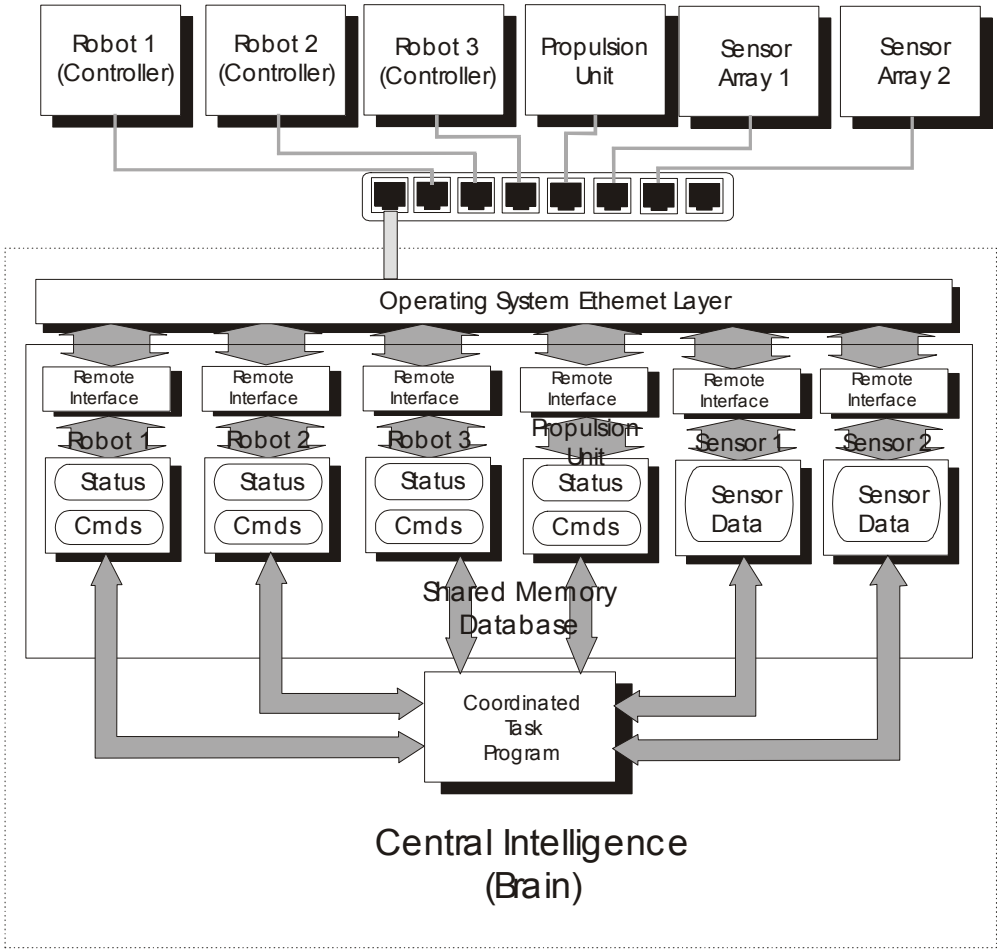


Figure 3.1. Overview of DROID

3.2. Network Architecture and Communications Protocol

Subunits, are defined as autonomous entities which process and provide data to the Brain Computer. Robotic arm subunits process data by receiving command data and executing some movement while providing positional data. Sensory subunits interpret sensor data and can provide target information to the Brain Computer. Depending on the functional class of the subunit, a class specific database will be created for that subunit. This database of known subunits is created at the start of execution of the initialization software resident on the Brain Computer. In order for a subunit to be connected to the database it must identify itself using a unique identifier or name which is placed in the system's name space. Ideally, a subunit should be powered and identified prior to starting the initialization software. Each subunit has a corresponding remote client which resides on the Brain Computer. It is this client which searches the name space for the subunit's server, establishes a "Message Passing Conduit" and refreshes the Brain Computer's subunit database.

The initialization software begins execution by building the appropriate database of subunits for a given set of tasks. It then executes each remote subunit client needed to maintain each subunit's database. Next, it executes a synchronization program known as the "heartbeat" program.

The "heartbeat" program is a fixed timer tied to the computer's real-time clock generating a synchronization event at a rate of 60 Hz. The "heartbeat" program is in fact a server which registers its own unique identifier in the system's name space. In order for remote subunit clients to refresh the databases in a synchronized manner, it must first search the name space for the heartbeat. It then connects to the "heartbeat" server and provides it with its Program ID. This also sets up a "Message Passing Conduit" and upon each synchronization event a message is sent to each client notifying it that a 60Hz cycle has begun. At the start of each new cycle, command data is sent to each subunit, and each subunit replies with its status data. For example, cartesian commands are sent to a robot arm, and the robot arm will reply with its encoder positions. This send and reply method guarantees that a connection is established throughout message passing since the reply acts as an acknowledgment.

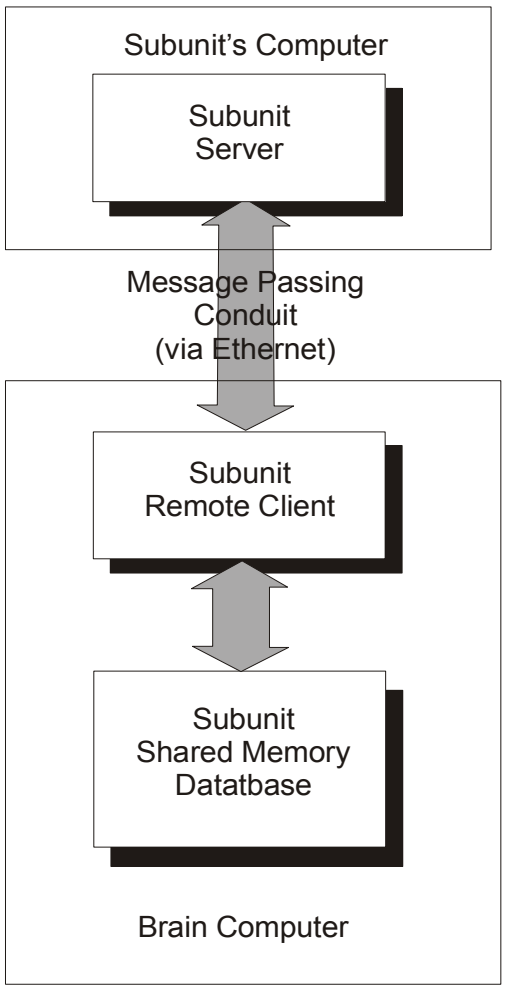


Figure 3.2. The three elements of a subunit

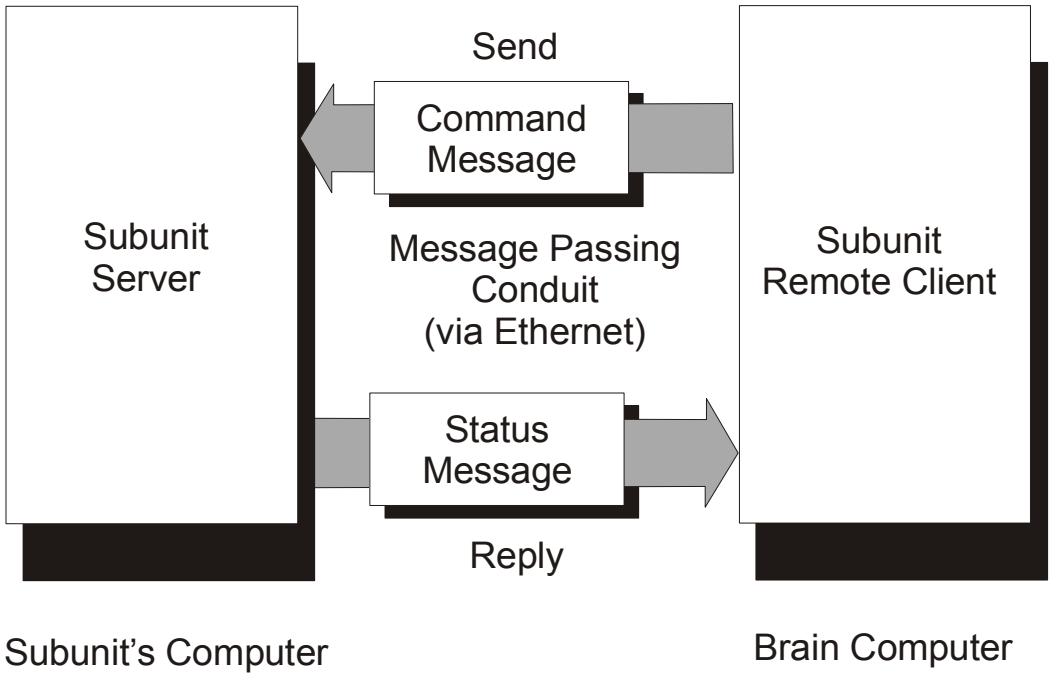


Figure 3.3. Message Passing for a robot subunit

3.3. Subunit Interfaces

Each class of subunit has a unique interface with which to communicate with the Brain Computer through. This section will describe in detail, the standard interface for a robot arm class with six revolute joints and a line laser sensor. Due to our example task program, Autonomous Multiple Robotic Welding, the only database classes available at this time are the standard 6 revolute joint robotic arm manipulator and the line laser tracking sensor.

Most industrial robot arm manipulators follow a common kinematic model. In our application, both the Reis Robotstar V15 and the GM Fanuc S400 share the same kinematic model, however the length of each link is different. Figure 3.4. compares the Reis Robotstar V15 and the GM Fanuc S400; they each have 6 revolute joints and 6 links. In addition the final 3 joints share a coincident origin at the wrist center.

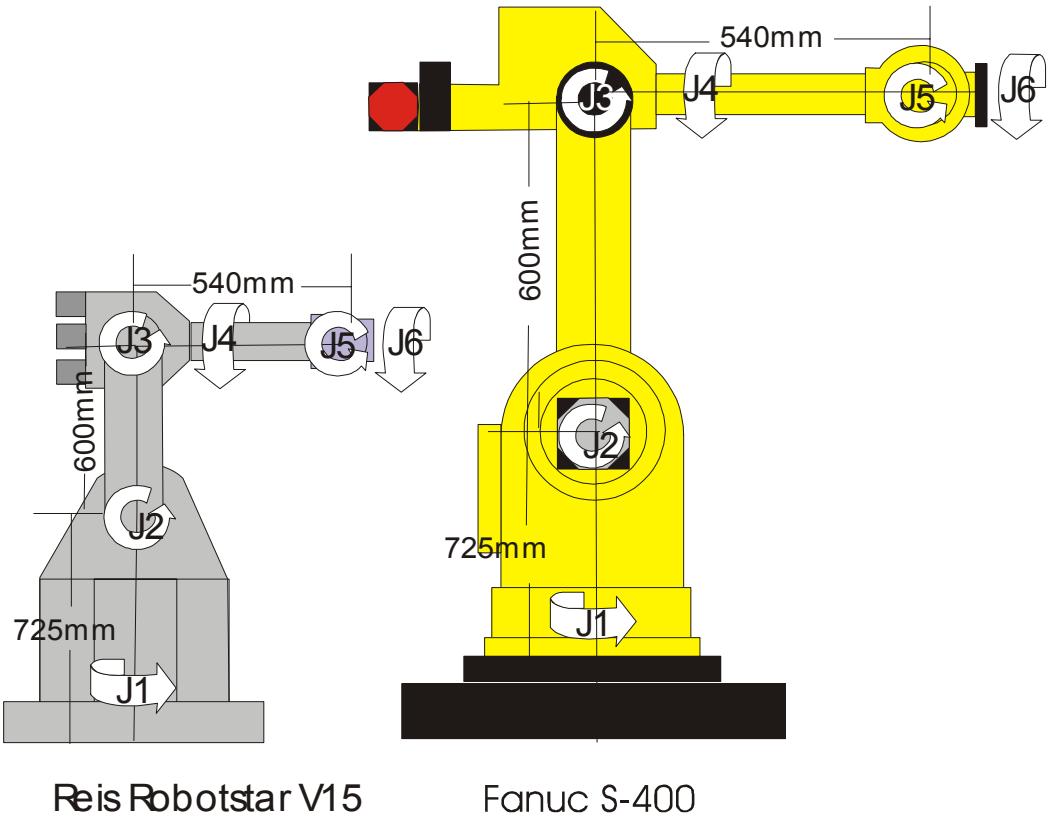


Figure 3.4. Comparison of Reis Robotstar V15 to GM Fanuc S400

For this class of robot manipulator two databases are generated for each subunit. They are known as the "Status shared memory Area" and the "Command shared memory area". Shared memory refers to the POSIX 1003.1 implementation of the Shared Memory Inter-Process Communication method found on most implementations of UNIX including QNX 4, SUN Microsystems Solaris and Linux. In this implementation, a memory area is created and made globally accessible to the operating system through an operating system call. Exclusive access to this is granted to the program which has ownership of the semaphore (signaling event). Programs use an operating system call to place the execution of the program in a sleep state (no CPU usage) until the semaphore is available. The semaphore is incremented upon acquisition and decremented upon release. The arbitration of which program has access to the semaphore is priority based. Therefore, programs using a shared memory resource should have the same

priority to avoid locking of the database due to preemption of the lower priority program.

Table 3.1. lists the fields in the status structure. In general, status information is retained in this database such as current cartesian position of the robot's Wrist-Center-Point (WCP) and the rotation matrix associated with the pose of this point. In addition, the 6 current joint encoder positions and the previous commanded encoder positions are retained in memory. By maintaining the previous commanded encoder positions we can track the target encoder position without accessing another shared memory area which would require waiting for another semaphore. Also, this database contains a busy signal, which tracks how many heartbeat counts are necessary to complete the current encoder position command. Therefore, position commands can be queued until the current command is completed. There is also a home status which indicates whether the robot's encoders have been zerod correctly (homed).

Table 3.2. lists the fields in the command structure. In general, command information is retained in this database. In order for robot arm manipulators to be commanded properly using

the Brain Computer, each robot must maintain their respective database using the following protocol. Command of the robot is governed by two status fields in the database refered to as the COMMAND mode and the COORDINATE mode.

Upon creation of the robot command structure, the COMMAND mode is placed in JOG mode while the COORDINATE mode is placed in JOINT mode. Since the robot has initially not been homed, only relative encoder commands can be made to jog the robot into the approximate home position or into a safe position prior to homing. Once homing is initiated, the robot's command mode is placed in HOME mode until it has completed its homing procedure at which point its command mode becomes NORMAL and the status HOME field becomes HOMED. At this point of operation, either JOINT mode commands or CARTESIAN mode commands can be made. A JOINT mode command comprises 6 encoder position commands along with the number of heartbeat counts needed to complete it. A CARTESIAN mode command comprises of a x, y, z positions in millimeters and the rotation matrix of the wrist-center-point relative to a global reference frame and the number of heartbeat counts required to complete the command.

Table 3.1. The Status shared memory interface

Field Name	Description
x	Cartesian x distance wrt Global Ref. Frame [mm]
y	Cartesian y distance wrt Global Ref. Frame [mm]
z	Cartesian z distance wrt Global Ref. Frame [mm]
R	3X3 Rotation Matrix wrt Global Ref. Frame
encoder_pos	6 joint encoder positions wrt to home [signed encoder counts]
encoder_cmd	current 6 joint encoder cmds wrt to home [signed encoder counts]
busy	"heartbeat" counts left for current command
homed	status flag either HOMED or FALSE

Table 3.2. The Command shared memory interface

Field Name	Description
x	Cartesian x distance wrt Global Ref. Frame [mm]
y	Cartesian y distance wrt Global Ref. Frame [mm]
z	Cartesian z distance wrt Global Ref. Frame [mm]
R	3X3 Rotation Matrix wrt Global Ref. Frame
cart_time_count	"heartbeat" counts for current cartesian command
encoder_pos	6 joint encoder cmds wrt to home [signed encoder counts]
joint_time_count	"heartbeat" counts for current joint command
coord	coordinate system: CARTSPACE or JSPACE
cmd_mode	command mode: OFF, HOME_MODE, JOG_MODE
sent	status flag: TRUE or FALSE

These definitions are kept in a common header file which is included in each accessing program. Also contained in the header file is the shared memory identifier. To access these memory areas, a program uses an operating system call along with the shared memory identifier to obtain the handle to the shared memory. It is then memory mapped into the program's memory space and accessed as a typical data structure; however, the semaphore must be waited on prior to access.

In general, sensor arrays provide targeting information to a system in order to update its position. Ideally they should operate autonomously in a continuous manner, constantly sensing the environment and relating the freshest data when requested; in other words, they should act as a server. In the case of the MVS Line Laser Sensor, the seam feature's location and surface orientation in sensor reference frame coordinates is required . These coordinates and orientation are then passed at the heartbeat rate to the Brain Computer. Therefore, only the transformation matrix is required by the brain to use the sensor. Table 3.3. lists the fields of the sensor's Status shared memory. It is very similar to the Status shared memory database of the robot arm manipulator except that an invalid flag is used to identify erroneous data.

Table 3.3. The Sensor status shared memory interface

Field Name	Description
x	Cartesian x distance wrt Global Ref. Frame [mm]
y	Cartesian y distance wrt Global Ref. Frame [mm]
z	Cartesian z distance wrt Global Ref. Frame [mm]
R	3X3 Rotation Matrix wrt Global Ref. Frame
encoder_pos	not used
encoder_cmd	not used
busy	not used
homed	used as an invalid flag for sensor data

3.4. Task Programs

A "Task program" is simply any program which can fulfill a task using the DROID system. These can take the form of simple move commands or they can be as complex as multiple robotic coordination programs such as the Seam Tracking program. In its essence, a "Task program" does the following: i) sets up shared memory access to the required subunit databases, ii) reads these databases and iii) modifies the command databases of the subunit to be commanded. If synchronization is required, the task program itself can search for the heartbeat and request for a Message Passing Conduit to be set up to achieve global synchronization.

The following utility task programs have been written and can be executed on any robot attached to the Brain Computer. These programs have been written using a UNIX style. This means that each program is very small, self-contained and provides a very specific function which can be controlled using runtime-parameters. Since the programs are written in this manner, they can be used in UNIX-style script files and executed very much like an interpreted language program. Thus a very flexible programming language can be built up from these small task programs.

The Joystick program, is a generic robot program which is used to command robot movement in Joint mode or Cartesian mode using an analog joystick. The robot identifier must be specified as a run-time argument to select which robot can be commanded. Use of the Joystick program is not only limited after the homing routine, but is available prior to homing in the event the robot must be moved to a safe location prior to homing.

The Home program is a generic robot program which initiates the homing routine by changing the Command Mode to the HOME value. Once the home signal has been sent to the robot, the robot takes care of the homing itself with no further control from the Home program. The robot identifier must be specified as a run-time argument to select which robot is to be homed.

The Move program is a generic robot program which commands movement to the robot in a specified time. At run-time, the move command accepts a robot identifier, a command mode, six position commands based on the command mode, and the time in seconds to complete

the move. Once the move command is initiated, it checks to see whether the robot is available for a move command via the Busy field in the status memory database. It then waits until the previous move is completed before writing to the command memory area. Joint mode commands require six encoder count values relative to the zero or home position, while cartesian mode commands require an x, y, and z position as well as the relative roll, pitch, roll angles of the final 3 joints (in degrees) to give the required wrist center point position and pose relative to the global reference frame. The Move program provides a very rich interface with which to build very elaborate move scripts very similar to machining applications.

The Snoop program is a generic program used to view database information continuously. At this point, it accepts a subunit identifier and echos all status information at a rate of 1Hz via the Standard Output port. There is no reason that it must be limited to only the standard console. It is possible to route this output to a Graphical User Interface or through a TCP/IP network port to be visualized on a remote station or a web page.

Chapter 4.

Generic Motion Control

This chapter focuses on one of the main aspects of the DROID system: the ability to control the motion of a robot subunit. The chapter begins with an explanation of the design of the generic motion algorithm with respect to the implementation on the Reis Robotstar V15, the prototype robot subunit. The implementation on a GM Fanuc S400 robot will also be described but is considered a slight variation of the Reis Robotstar V15's motion controller. The objective of this chapter is to give the reader enough background information necessary to implement the algorithm on any given robot subunit. It is in theory possible to implement any robot to be used on the DROID system.

4.1. Prototype Implementation, the Reis Robotstar V15

4.1.1. Robot Mechanics and Kinematics

Before we can begin to design a motion controller for the Reis Robotstar V15, the mechanics of motion must be understood. This involves determining the robot's forward kinematics, the inverse kinematic solution as well as encoder gearings and inter-relationships among joints and calibration values.

4.1.1.1. Robot Forward Kinematics

The Reis Robotstar V15 is a 6 revolute joint robot which provides 6 degrees of freedom, hence any position and wrist orientation can be performed by this robot provided that this point and orientation can be reached physically. Figure 4.1. illustrates the link and joint relations of the Reis Robotstar V15.

Although the motion controller doesn't necessarily require that the current cartesian position and orientation be known, it is useful for monitoring the progress of a cartesian move command. The current cartesian position and orientation of the wrist center can be determined using the forward kinematics of the Reis Robotstar V15. Using the Denavitt-Hartenberg [27], [28] convention, we assign reference frames to the robot in the home configuration and determine the link parameters. Figure 4.2. illustrates the reference frames assigned to the robot while Table 4.1. shows the link table for the robot.

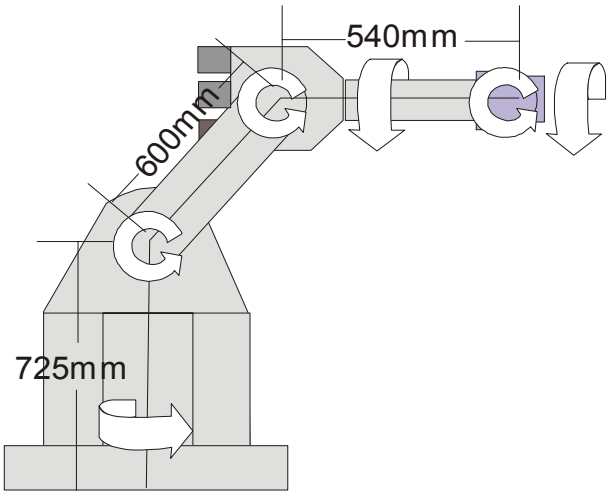


Figure 4.1. Reis Robotstar V15 Geometry

To determine the wrist position and orientation with respect to the world reference frame we perform a series of transformations. Equation 4.1. shows each transformation matrix for each revolute joint. Using the link table, we can determine the 6 local transformation matrices for each joint.

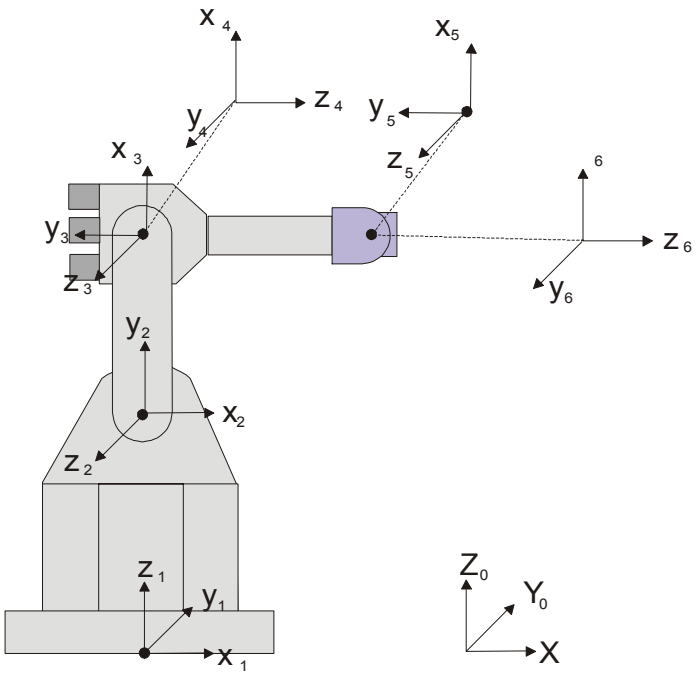


Figure 4.2. Reference Frame Assignment based on D-H convention

Table 4.1. Link Table for D-H convention

Link	Joint Variable	Angle θ_n	Displacement d_n	Length l_n	Twist α_n
1	θ_1	θ_1	725mm	0	+90°
2	θ_2	θ_2+90°	0	600mm	0°
3	θ_3	θ_3	0	0	+90°
4	θ_4	θ_4	540mm	0	-90°
5	θ_5	θ_5	0	0	+90°
6	θ_6	θ_6	0	0	0°

$$ {}^{1}A_2 = \begin{bmatrix} \cos\theta_1 & 0 & \sin\theta_1 & 0 \\ \sin\theta_1 & 0 & -\cos\theta_1 & 0 \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{2}A_3 = \begin{bmatrix} -\sin\theta_2 & -\cos\theta_2 & 0 & -l_2\sin\theta_2 \\ \cos\theta_2 & -\sin\theta_2 & 0 & l_2\cos\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

A) Joint 1 A Matrix B) Joint 2 A Matrix

$$ {}^{3}A_4 = \begin{bmatrix} \cos\theta_3 & 0 & \sin\theta_3 & 0 \\ \sin\theta_3 & 0 & -\cos\theta_3 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{4}A_5 = \begin{bmatrix} \cos\theta_4 & 0 & \sin\theta_4 & 0 \\ \sin\theta_4 & 0 & -\cos\theta_4 & 0 \\ 0 & -1 & 0 & l_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

C) Joint 3 A Matrix D) Joint 4 A Matrix

$$ {}^{5}A_6 = \begin{bmatrix} \cos\theta_5 & 0 & \sin\theta_5 & 0 \\ \sin\theta_5 & 0 & -\cos\theta_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{6}A_{wcp} = \begin{bmatrix} \cos\theta_6 & -\sin\theta_6 & 0 & 0 \\ \sin\theta_6 & \cos\theta_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

E)

Joint 5 A Matrix F) Joint 6 A Matrix

(4.1.)

From Figure 4.2. ${}^{0}A_1$ is an arbitrary matrix that defines the World Frame (Frame 0) with respect to the robot's base frame (Frame 1). Multiplying the above 6 matrices with ${}^{0}A_1$ yields the wrist center position and rotation matrix (WCP) with respect to the world frame.

$$ {}^{0}T_{wcp} = {}^{0}A_1 \, {}^{1}A_2 \, {}^{2}A_3 \, {}^{3}A_4 \, {}^{4}A_5 \, {}^{5}A_6 \, {}^{6}A_{wcp} \qquad (4.2.) $$

4.1.1.2. The Inverse Kinematic Solution

In order to convert cartesian positions and orientations to joint angles, the inverse kinematic solution for the robot must be solved. The robot kinematics of the Reis Robotstar V15 allow for a geometric approach to solving the inverse kinematics of the robot. Since all 6 revolute joints share a common plane and the fact that the last 3 joints are located at the wrist center, we can solve for position by isolating the first 3 joints. Figure 4.3. Illustrates the geometric solution for determining angles 1, 2, 3 from trigonometry. Angle 1 is determined simply by using the arctan function for the x and y position. Using the z position and the projection of links 2 and 3 on the xy plane we can determine the distance from the origin to the wrist center point. We can then determine angles 2 and 3 using the cosine law on the triangle formed by links 2, 3 and the distance from the origin to the wrist-center point.

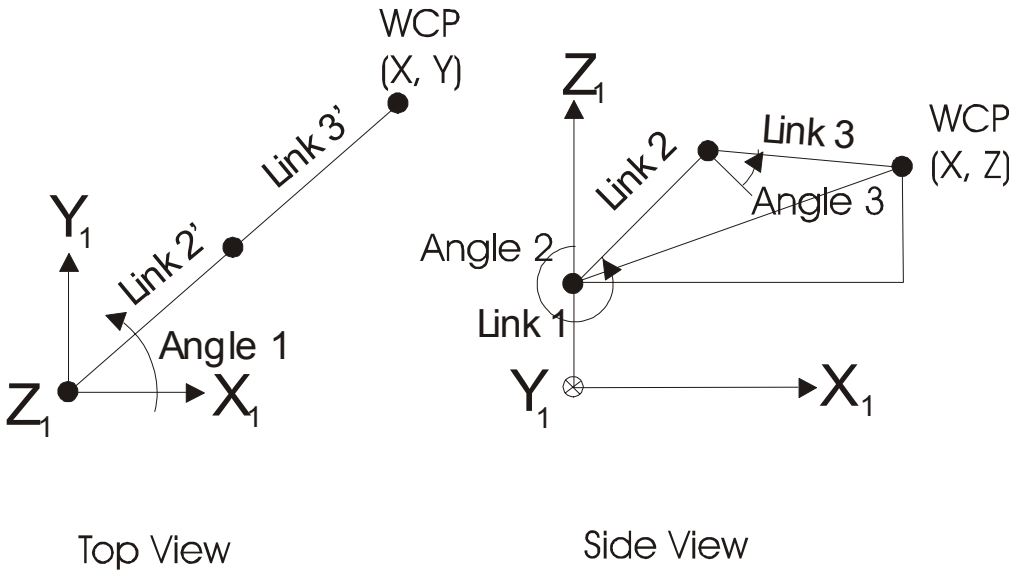


Figure 4.3. Geometric solution for angles 1,2,3 based on position only

Notice that it is possible to have 2 different solutions at this point for angles 2 and 3 depending on the configuration of the elbow (above or below the horizontal through the WCP). In general, all solutions should be found and the closest solution to the current joint configuration will be

selected.

To determine the final 3 joint angles so as to arrive at the desired wrist orientation, matrix methods are used. The wrist configuration of the Reis Robotstar V15 consists of a roll axis (Z) revolute joint followed by a pitch axis (Y) joint followed by a roll-axis joint (Z). This corresponds to the Euler ZYZ matrix shown in Equation 4.2. where alpha, beta and gamma correspond to joint angles 4,5 and 6 in radians respectively [27].

$$R_{Z'Y'Z'}(\alpha,\beta,\gamma) = \begin{bmatrix} c\alpha\, c\beta\, c\gamma - s\alpha\, s\gamma & -c\alpha\, c\beta\, s\gamma - s\alpha\, c\gamma & c\alpha\, s\beta \\ s\alpha\, c\beta\, c\gamma + c\alpha\, s\gamma & -s\alpha\, c\beta\, s\gamma + c\alpha\, c\gamma & s\alpha\, s\beta \\ -s\beta\, c\gamma & s\beta\, s\gamma & c\beta \end{bmatrix} \tag{4.2.}$$

This matrix can be considered as the transformation from the local reference frame situated at joint 4 to the desired orientation of the wrist-center (${}^4T_{wcp}$). Using the forward kinematics developed in section 4.1.1.1., we can substitute angles 1, 2, and 3 into the kinematic chain to determine the transformation from the origin to frame 4 (see Equation 4.3). Hence the wrist-center transformation (${}^0T_{wcp}$) can be determined by multiplying ${}^0T_4$ to ${}^4T_{wcp}$.

$${}^0T_4 = {}^0T_1\ {}^1T_2\ {}^2T_3\ {}^3T_4 \tag{4.3.}$$

$${}^0T_{wcp} = {}^0T_4\ {}^4T_{wcp} \tag{4.4.}$$

$${}^4T_{wcp} = ({}^0T_4)^{-1}\ {}^0T_{wcp} \tag{4.5.}$$

Therefore angles 4,5 and 6 can be determined by comparing the resultant matrix of Equation 4.5. By inspection, alpha, beta, and gamma can be easily solved using inverse trigonometry, however this leads to multiple solutions.

$${}^4T_{wcp} = R_{z'y'z'}(\alpha, \beta, \gamma) \tag{4.6}$$

$$\beta = \text{acos}^{-1}({}^4T_{wcp}(3,3)) \tag{4.7.}$$

$$\alpha = \text{acos}^{-1}({}^{4}T_{wcp}(1,3))/\sin(\beta) \quad (4.8.)$$

$$\gamma = \text{asin}^{-1}({}^{4}T_{wcp}(3,2))/\sin(\beta) \quad (4.9.)$$

For each possible alpha angle, we can determine the corresponding beta and gamma. In total there are 8 possible solutions for the inverse kinematics, however, due to workspace limits, not all are accessible. Therefore, we determine all solutions and eliminate those that are not reachable, we then choose the closest solution in terms of joint space movements. The method for determining the closest solution is explained in detail in Section 4.1.7.

4.1.1.3. Encoder Gearing and Interconnectivity

Once the joint angles for a given cartesian move are determined by the inverse kinematic solution, we must determine the number of equivalent encoder counts. To convert radians into encoder counts we must have specific knowledge of the gear reduction ratios from the harmonic drives to the joint. The Reis Robotstar V15 uses harmonic drive motors which are reduced by a factor of 100 to drive each joint. According to the Reis Robotstar V15's literature [22], the harmonic drives are equipped with differential encoders such that one revolution requires 4000 counts. Hence the final ratio of degrees to encoder counts is calculated using Equation 4.10.

$$\text{radians to encoders} = 1000*4000/360 \quad (4.10.)$$

The Reis Robotstar V15 uses a servo motor placement system which couples certain joints together. Joint 2 and joint 3 are coupled such that for every 100 encoder counts of joint 3, one encoder count must be added to joint 2. As well for every 100 encoder counts of joint 2, one encoder count must be added for joint 3. Joint 4 and joint 5 are coupled such that for every 100 encoder counts of joint 4, one encoder count must be subtracted to joint 5. As well for every 100 encoder counts of joint 5, one encoder count must be subtracted for joint 4.

$$j2 = j2\_angle *(\text{radians to encoders}) + j3/100 \quad (4.11.)$$

j3 = j3_angle *(radians to encoders) + j2/100 (4.12.)

j4 = -j4_angle *(radians to encoders) + j5/100 (4.13.)

j5 = j5_ang *(radians to encoders) - j4/100 (4.14.)

4.1.1.4. Optical Calibration Procedure

In most robot applications, the home position of the robot is determined using a reference fixture. Once this position is determined, the encoder values are stored and referred to each time the robot needs to return to its home position. This is a fairly straight forward system if the robot is relatively small, however, as the robots get larger, so do the fixtures.

An alternative to this fixture method is to use an optical method of calibration which uses a structured light beam to determine the accuracy of a given set of calibration values. The structured light beam, an LED laser, is placed in the center of the tool face of the robot and a sequence of moves and measurements are performed to determine the accuracy of the gearing ratios and the home offsets from the limit switch positions of each joint.

We begin by placing the robot in a pose close to the home position and mark the position of the projected laser point on a screen. To ensure that the laser is centered correctly within the fixture, joint 6 is rotated and the projected point is checked for coincidence. Joint 4 is rotated, if joint 5 is at the correct home position, the laser should remain at the same point. Otherwise joint 5's calibration point is adjusted until it is correct. Following this, joint 3 is rotated and the line it produces on the screen is traced. Joint 5 is then rotated, if joint 4 is at the correct home position, these lines should be coincident. Otherwise joint 4's calibration position is adjusted. Next, the robot is moved such all links point straight up. Rotating joint 1 and joint 4 should produce a single dot, otherwise, joint 2's calibration point is adjusted along with joint 3's 90^{o} position. Our final check involves drawing a circle with a radius of Link 3 on the floor. The robot is moved to its new perceived home position and joint 5 is rotated 90^{o} to the floor. Joint 1 is rotated, if joints 2 and 3 are calibrated correctly, the circle should have a diameter of 1080mm.

From the above procedure, it can be seen that the calibrated values for joint 1 and 6 are arbitrary. In our implementation, joint 6 is calibrated using an inclinometer, while the zero position for joint 1 lies at the center of its workspace. Although the above procedure can be quite tedious, it allows a fixture-less method to be used for calibration requiring only an inexpensive LED laser and a ruler. Since most robots share the same kinematics with the Reis Robotstar V15, this calibration method can be extended to them as well.

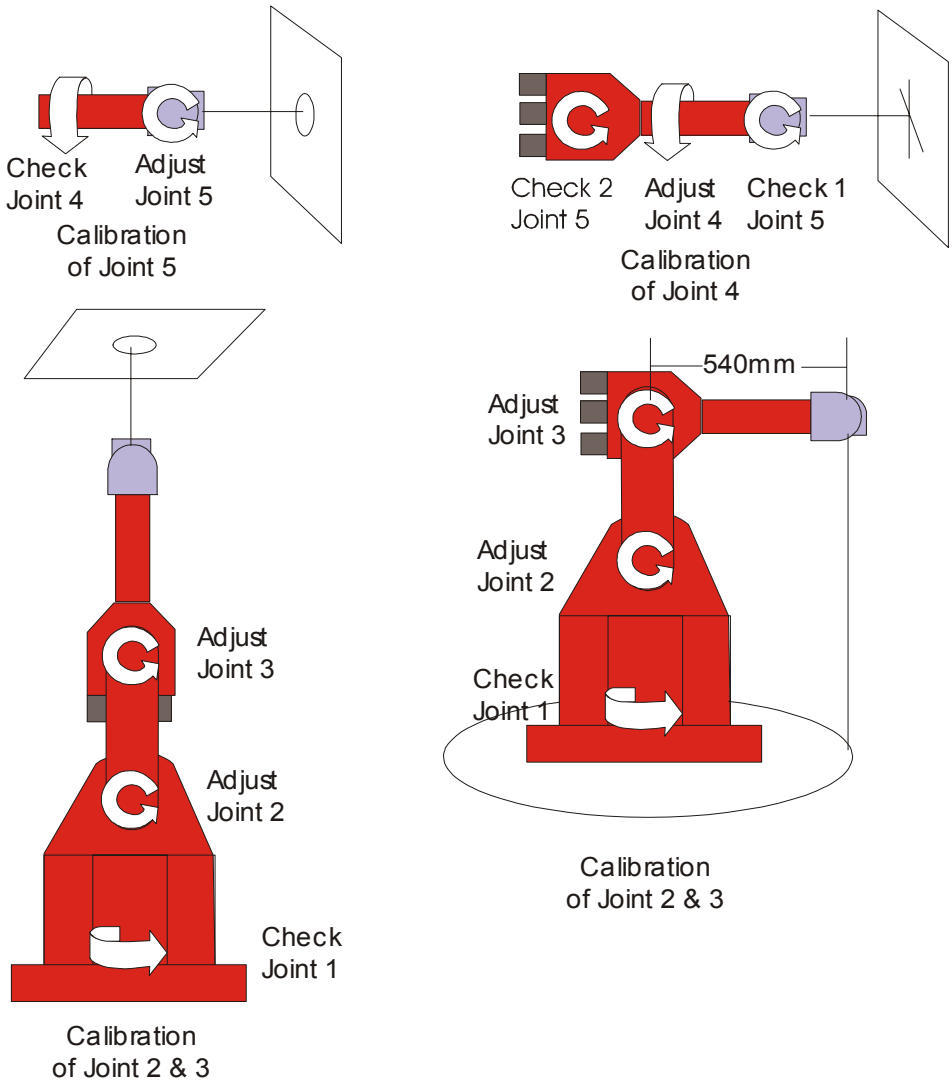


Figure 4.4. Optical Calibration

4.1.2. Robot Hardware Interfacing

The Reis Robotstar V15 is equipped with 2 interface expansion cards. These interface expansion cards allow robot devices to be mapped into the memory of the VMEbus. These robot devices include i) 6 16-bit encoder counters, ii) 6 12-bit digital to analog converters attached to each servo amplifier, iii) 6 joint limit switches, and iv) 2 Watchdog timers.

Each joint is equipped with a differential quadrature encoder which increments or decrements a 16 bit counter upon rotation. Upon power up, these counters contain an arbitrary value, however any write to this memory location will reset the encoder counter to 0. This feature is not always available for other robots, hence it is not exploited in the generic motion controller.

Each joint is driven using a servo motor and a harmonic drive with a ratio of 100:1. This servo is driven with a DC voltage provided by a servo amplifier. The input voltage is supplied to the servo amplifier by a signed 12 bit digital to analog converter (DAC).

For each joint, a limit switch is installed in a known rotational location. Its primary function is to serve as an aid in calibration of the robot's home position. This limit switch is connected to a digital input line.

In the event that the interface expansion cards are disconnected from the VMEbus, a timer will elapse and shutdown the servo-amplifier's power supply. This will latch a system fault which can only be reset by powering down the entire Reis Robotstar V15's controller. These timers must be written to at a constant rate to ensure continuous operation of the controller. Each timer register is connected to a 16-bit digital output port.

The Reis Robotstar V15 is equipped with a VMEbus passive backplane which all expansion cards are attached to. Since the VMEbus adopts the Motorola standard of mapping devices into the physical memory space, all devices connected to the 2 interface expansion cards (such as the devices mentioned earlier) can be directly accessed using conventional memory read and write routines.

4.1.3 Software Design

The main purpose of the Generic Motion Controller is to provide a universal motion control interface to the Brain Computer for any given robot or motion device. It was originally developed on the Reis Robotstar V15 as a high performance replacement motion controller for the original controller which was shipped with the robot in the late 1980's. Due to the Reis Robotstar V15's VMEBus architecture, upgrading the motion controller was possible without extensive hardware intervention. Using modern processors and the QNX 4 realtime operating system, a truly flexible and extremely modular approach to software motion control was taken. To achieve this degree of modularity and flexibility a centralized database is used to store command and status data much like the shared database of the Brain Computer. However, additional information is required to achieve low-level control. This data-centric architecture allows every aspect of motion control to be achieved independently by small program modules with a standard interface to the central database. Hence, kinematic models, device drivers, servo controllers, and subunit servers can be added, updated or removed to suit any robot hardware requirements. The generic motion controller implementation for the Reis Robotstar V15 represents the superset of generic motion control software. This is due to the ability to gain very low-level access to the robot hardware via the VMEBus. Hence, its motion control software is used as a standard implementation for a 6 revolute joint robot arm manipulator.

Similar to the shared memory of the Brain Computer, the motion controller computer "arm1" also retains two shared memory areas corresponding to the status and command information of the robot. In addition, access to this shared memory is achieved in the identical manner as that described in Section 3.3 for the Brain Computer.

Table 4.2. lists the fields in the Status shared memory structure. In comparison to the Status shared memory structure residing on the Brain Computer, the Status shared memory structure residing on the “arm1" computer is identical. Table 4.3. lists the fields in the Command shared memory structure. In comparison to the Command shared memory structure residing on the Brain Computer, there are two sets of encoder command fields called external encoder and internal encoder. These encoder command sets refer to a gross Joint Mode

Command and a Joint-Space interpolated way-point generated from the Planner program module.

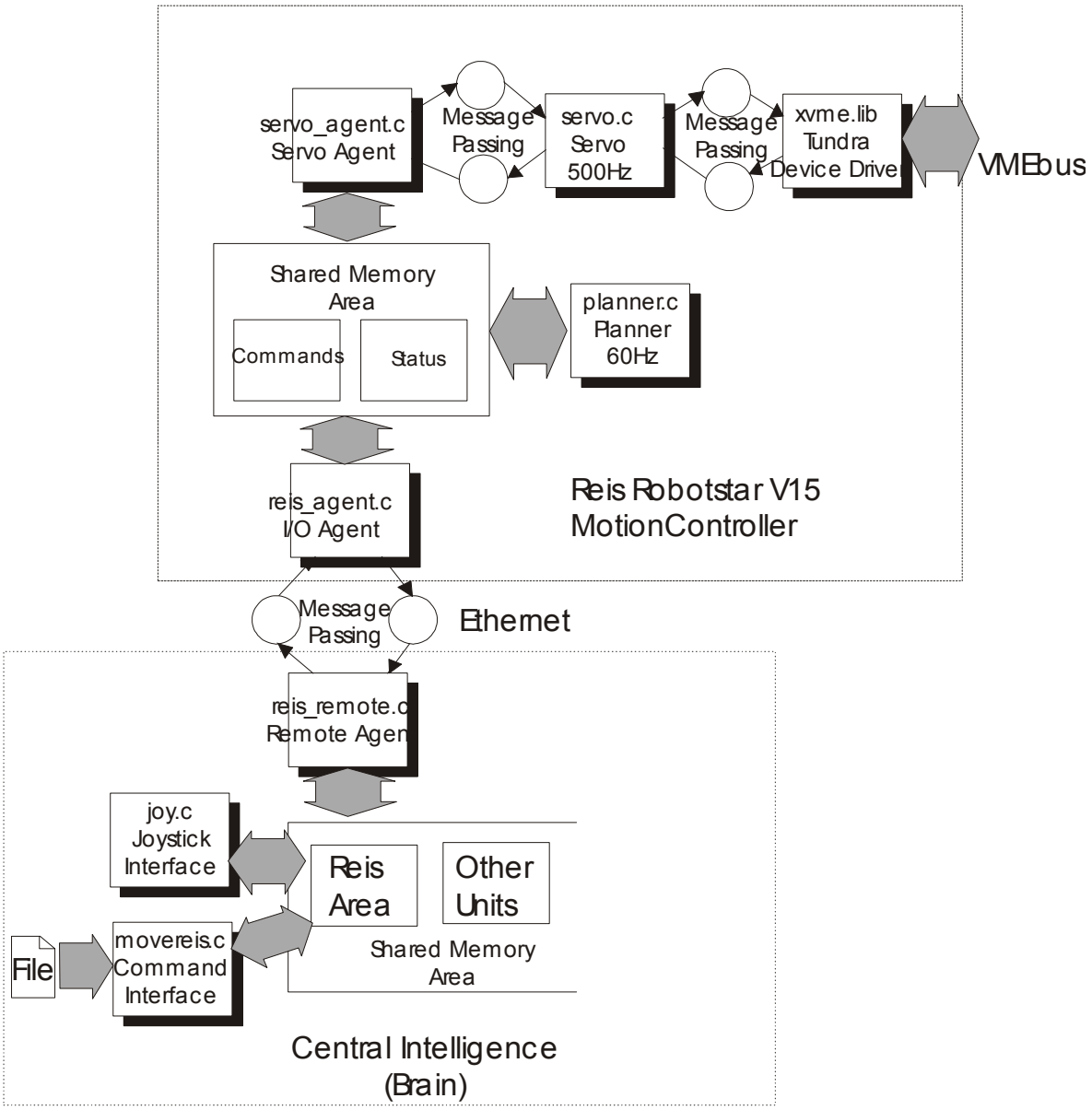


Figure 4.5. Datagram of the Reis motion controller

Table 4.2. The Status shared memory interface of the motion controller

Field Name	Description
x	Cartesian x distance wrt Global Ref. Frame [mm]
y	Cartesian y distance wrt Global Ref. Frame [mm]
z	Cartesian z distance wrt Global Ref. Frame [mm]
R	3X3 Rotation Matrix wrt Global Ref. Frame
encoder_pos	6 joint encoder positions wrt to home [signed encoder counts]
encoder_cmd	current 6 joint encoder cmds wrt to home [signed encoder counts]
busy	"heartbeat" counts left for current command
homed	status flag either HOMED or FALSE

Table 4.3. The Command shared memory interface of the motion controller

Field Name	Description
x	Cartesian x distance wrt Global Ref. Frame [mm]
y	Cartesian y distance wrt Global Ref. Frame [mm]
z	Cartesian z distance wrt Global Ref. Frame [mm]
R	3X3 Rotation Matrix wrt Global Ref. Frame
cart_time_count	"heartbeat" counts for current cartesian command
encoder_cmd_ext	6 joint encoder cmds wrt to home [signed encoder counts], external
joint_time_count	"heartbeat" counts for current joint command
encoder_cmd_int	6 joint encoder cmds wrt to home [signed encoder counts], internal
coord	coordinate system: CARTSPACE or JSPACE
cmd_mode	command mode: OFF, HOME_MODE, JOG_MODE

4.1.4. Functionality

Figure 4.6. illustrates the general operation of the Generic Motion Control software implementation for the Reis Robotstar V15. This software acts as an embedded robot controller which requires very little user intervention. Upon powering on the Reis Robotstar V15's controller chassis, the VMEbus processor board is booted. QNX 4.25 is boot-strapped and begins to load the necessary device drivers which include i) video/console drivers, ii) network adapter drivers, iii) network protocols iv) serial communication drivers v) and VMEBus drivers.

A safety feature of the Reis Robotstar V15 and most conventional robot arm manipulators is the existence of Watchdog timers. Watchdog timers must be written to at a given interval to ensure that the processor board is still attached to the VMEbus. If disconnection of the processor module occurs these timers are not written too, hence control is lost and the control shuts down its servo motors and logs a fault. Hence a Watchdog program is executed following the device driver loading to ensure continuous operation of the motion control software.

Once access to the VMEbus is possible and the Watchdog timers are maintained, the Servo program module is executed. At this point, the operator can safely enable the servo motors without having unwanted motion. The Servo program module executes its Proportional-Integral-Derivative (PID) discrete control law at an independent operating frequency of 500Hz and waits for HOME signal from the Brain Computer.

Following the Servo Program's execution is the creation of the Status and Command shared memory structures. These structures are initialized with safe values so that unwanted sudden motion is eliminated.

Once the shared memory structures are available, the Servo Agent program module is executed. Since the Servo program module maintains constant control of the servo motors at an independent rate of 500Hz, interruptions cannot be tolerated. In addition, direct access to the shared memory area could lead to lock-ups as other programs may preempt the Servo program. Hence the Servo Agent is used to access the shared memory structures on behalf of the Servo program.

The Subunit Server program module is then executed. It refreshes the Command shared memory structure with the Command shared memory structure data on the Brain Computer and simultaneously sends the contents of the Status shared memory structure to the Brain Computer at the "heartbeat" frequency (60Hz). Finally, the Planner program module is executed but it suspends its execution until the robot has been homed.

Once the HOME signal has been received by the Servo program module it begins its Homing procedure which consists of sequentially moving each joint through its range looking for a limit switch at a known encoder position. After all switches have been located, the HOME status is changed to NORMAL.

At this point of operation either Joint Mode commands or Cartesian Mode commands can be sent to the motion controller. Based on the Command Mode, the Planner Program module will read the given command and convert it into a series of interpolated way-points. The next way point is placed in the internal encoder command field of the Command shared memory structure. In addition, the remaining "heartbeat" counts for the current gross move command is updated.

The Servo Agent program module then reads the internal encoder command fields and simultaneously updates the current encoder positions of the Status shared memory structure. The internal encoder commands are passed to the Servo program module.

Once the Servo program module receives the internal encoder commands it executes its Joint Level Planner subroutine. The Joint Level Planner subroutine executes at the "heartbeat" frequency subdividing the command into 8 Joint-Space way-points (500Hz/60Hz . 8) and places these way-points into a position queue. Depending on the circumstances, different position queues or velocity profiles are fed to the Servo Controller routine. For example, in the event of a sudden stop, a "braking" velocity profile is used to ensure that the servo motor gears are not damaged.

The Servo Controller routine uses its discrete PID control law to compute the required control action for each servo motor. This control action is then scaled to a value which can be written into a DAC which is attached to the servo motor hardware. These DACs and encoder counters can be accessed from function calls via the VMEbus device driver.

This entire motion control system acts as a state machine. All that is required to perform a move command is to change the respective command fields in the Command shared memory structure. The rest is taken care of by the Generic Motion Control programs.

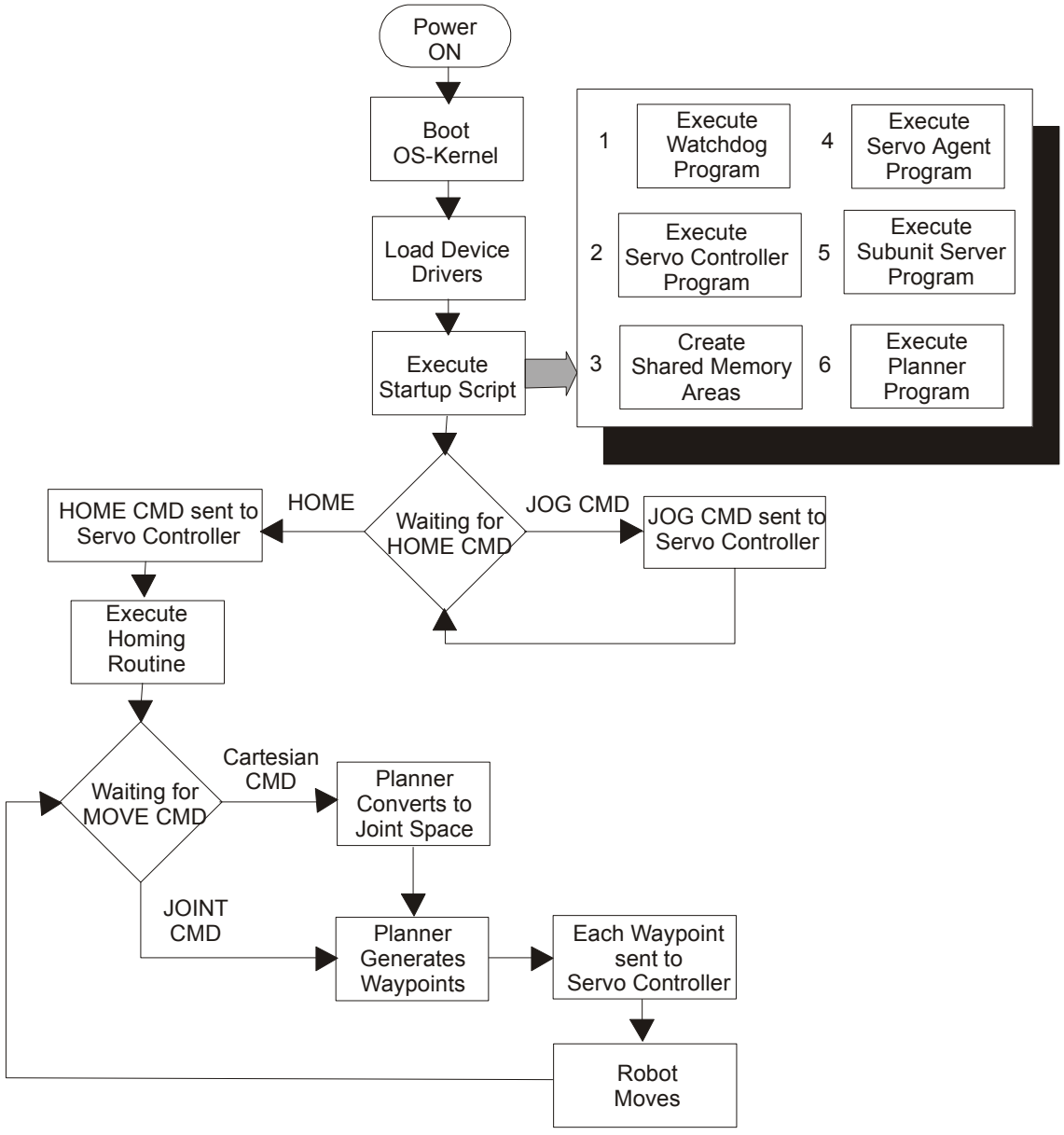


Figure 4.6. Flowchart of Motion Controller Operation

4.1.5. The Servo Program

The Servo program performs the low-level functions required to maintain control of the Reis Robotstar V15's 6 joints. Operating at a rate of 500Hz, it is the primary control unit of our generic motion controller. The Servo program itself is composed of an initialization phase followed by a main control loop in which 3 major functions are executed based on the state of the motion controller. Figure 4.7. illustrates the execution flow of the Servo program. The 3 major functions include i) the Servo function which is a discrete implementation of a 6 channel Proportional-Integral-Derivative Controller (PID), ii) the servo_planner function which feeds encoder commands to the Servo function via a command queue and iii) the Home function which executes a homing routine to determine the zero position of each joint.

The Servo program begins with the registration of the Servo program name into the system's name space. This allows other programs to communicate with the servo program via a Message Passing Conduit. Message Passing is more suited for the servo program so as to avoid a lock up if it were to access a shared memory area instead.

Following name registration, we setup a 500Hz rate generator to trigger a signal every 2ms using the set_timer function. Next we enter the main execution loop which puts the processor in a wait state until a message is received. If the message received is a triggered signal from the rate generator, it executes the Servo function in conjunction with the homing routine based on the home status flag, otherwise, it checks if it is a command packet which will then execute the servo_planner function. The command packet contains high-level joint or cartesian commands which are sent at a rate of 60Hz from the servo_agent program. Since the command and the low-level control rate are different, a position command queue is required to separate the command packet into 8 interpolated movements which will then be executed by the Servo routine at the 500Hz rate. The majority of the code in the Servo program deals mainly with writing appropriate values into this position command queue or in support of it.

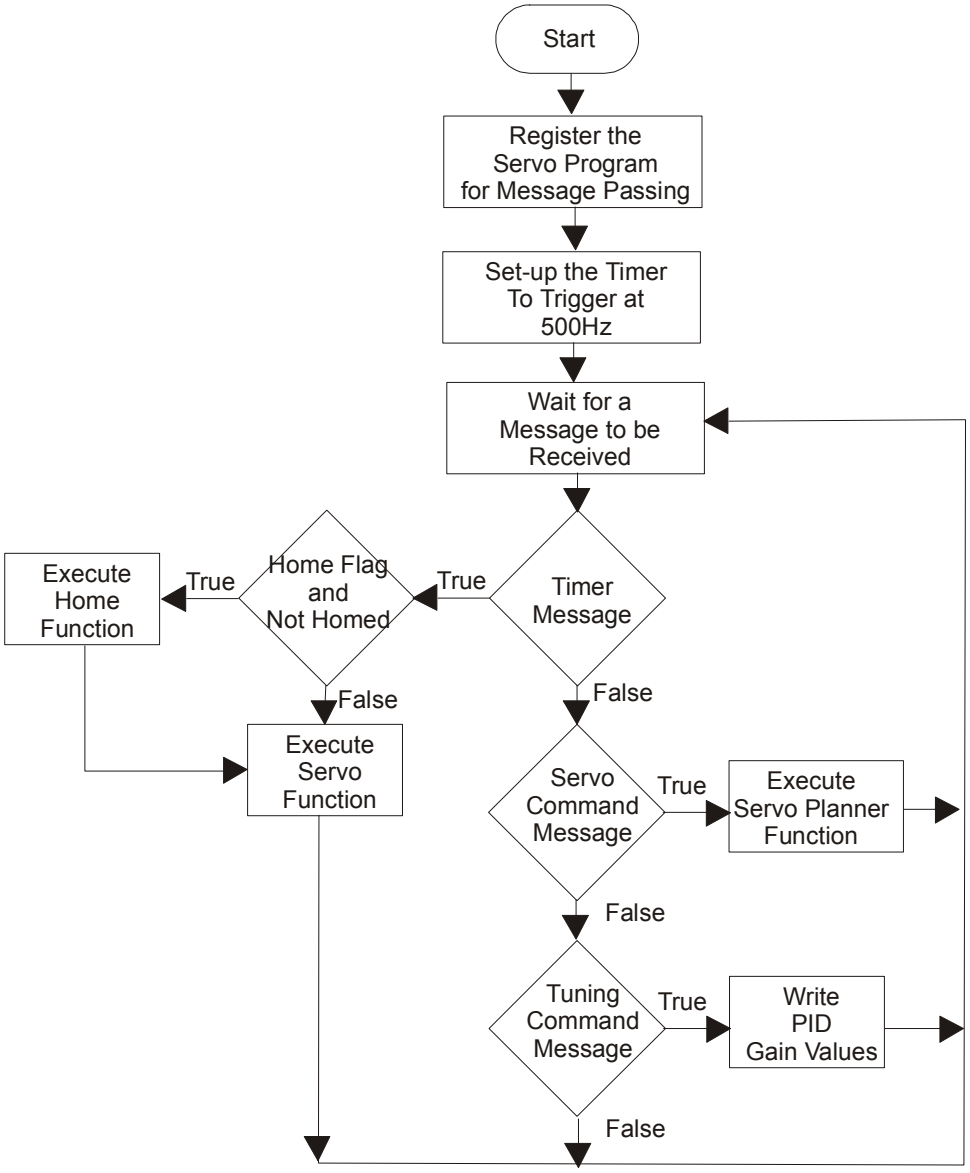


Figure 4.7. Flowchart of the Servo program

4.1.5.1. The Servo Routine

The low-level access to the robot hardware is performed by the Servo function. In order to read the joint encoder values, limit switch status' and to drive the servo motors, read/writes

must be performed on the memory addresses of the VMEbus. These read and writes are performed by calling compiled device driver functions provided by the VMEbus hardware manufacturer and including them with the Servo program.

The Servo function begins with the reading of the joint encoder counters via the device driver functions; these 16-bit values are then checked for encoder roll-over and converted into a 32-bit signed integer. Next, the target encoder positions are computed based on the home flag. The homed situation computes the target values by obtaining the relative-from-home orientation encoder commands from the position command queue and adding it to the calibrated zero encoder position. The non-homed situation copies the relative encoder commands to the target values. These target values are then checked against a set of maximums and minimums before the position command queue counter is decremented.

In fact there are two position command queues, one which is accessed directly by the servo function, while the other holds the next 8 positions to be commanded. This second queue is referred to as the position command queue cache. Upon decrementing the position command queue counter to zero, the servo function copies the contents of the queue cache to the queue and resets the counter to 8. However, if there is no data available in the cache, an emergency braking procedure is loaded into the position command queue instead and executed.

Once the target data is available, a discrete PID controller is implemented using the following algorithm [26].

```
pos_error = target_encoder_pos[i] - encoder_pos_abs[i];
servo_cmd = polarity[i]*K1[i] * (pos_error + K2[i] * integral[i]);
if (servo_cmd > -RANGE12BIT)
  if (servo_cmd < RANGE12BIT)
    integral[i] = integral[i] + TIME_CONSTANT * (pos_error - K3[i] *
                  servo_cmd);
  else
    servo_cmd = MAX12BIT;
else
  servo_cmd = MIN12BIT;
```

The limiting of the servo_cmd to the maximum and minimum control action is required to avoid

the integral error accumulation. Finally these servo command values are written into the DAC registers of the interface cards via the VMEBus device driver functions.

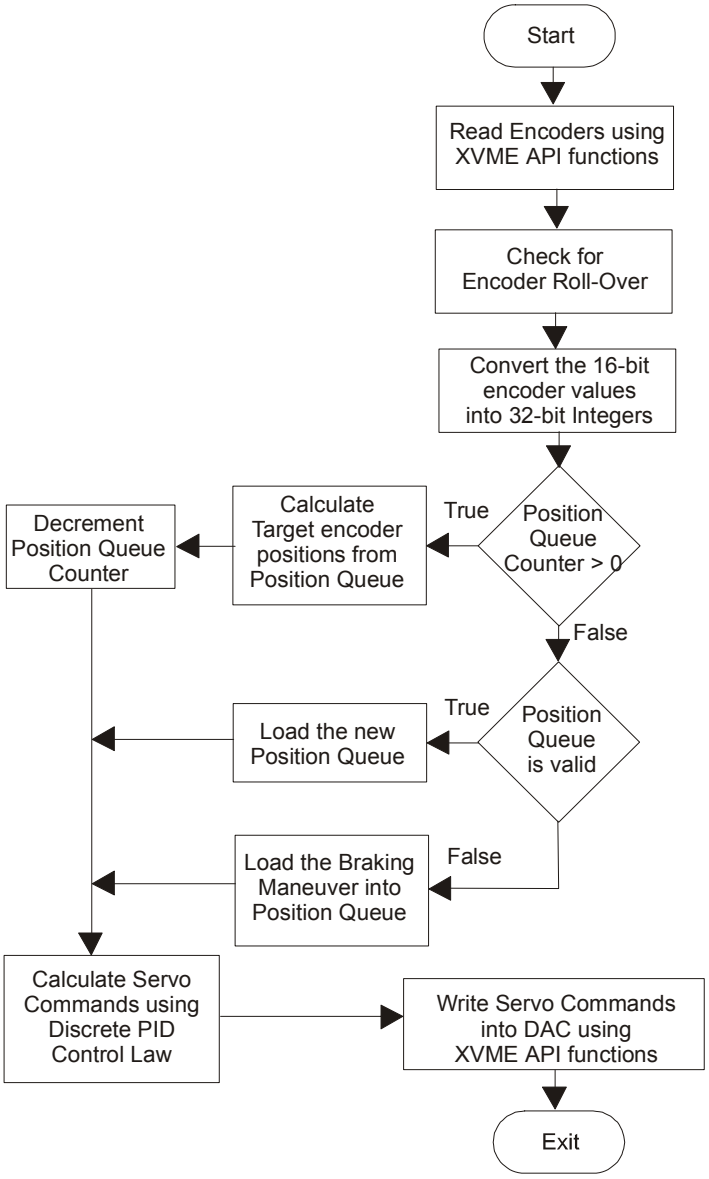


Figure 4.8.
Flowchart of the Servo Function

4.1.5.2. The Servo Planner

The Servo Planner function is responsible for shaping the position command queue cache which is delivered to the Servo function and allows smooth transitions to the target points. The function begins with determining the target encoder position. If JOG Mode is required, the target encoder position is determined by adding the encoder command from the command message to the last encoder command hence a relative command is generated. Otherwise, the target encoder position is copied directly from the command message. The target encoder position is then limited by the maximum allowable speed for the given joint to become the actual target position.

Once, the actual target position is computed, the encoder step value is determined. In this application a linear ramp is desired and is shown in the following code segment. This algorithm permits linear accelerations and braking. Note that the final value in the queue guarantees that the cache remains on target.

```
encoder_step = (target_encoder_act[i] - encoder_cmd[i][QUEUE_DEPTH -1])
                /(QUEUE_DEPTH);
encoder_cmd_cache[i][0] = encoder_cmd[i][QUEUE_DEPTH -1] +
                          encoder_step;
encoder_cmd_cache[i][1] = encoder_cmd[i][QUEUE_DEPTH -1] +
                          2*encoder_step;
encoder_cmd_cache[i][2] = encoder_cmd[i][QUEUE_DEPTH -1] + 3*encoder_step;
encoder_cmd_cache[i][3] = encoder_cmd[i][QUEUE_DEPTH -1] + 4*encoder_step;
encoder_cmd_cache[i][4] = encoder_cmd[i][QUEUE_DEPTH -1] + 5*encoder_step;
encoder_cmd_cache[i][5] = encoder_cmd[i][QUEUE_DEPTH -1] + 6*encoder_step;
encoder_cmd_cache[i][6] = encoder_cmd[i][QUEUE_DEPTH -1] + 7*encoder_step;
encoder_cmd_cache[i][7] = target_encoder_act[i];
```

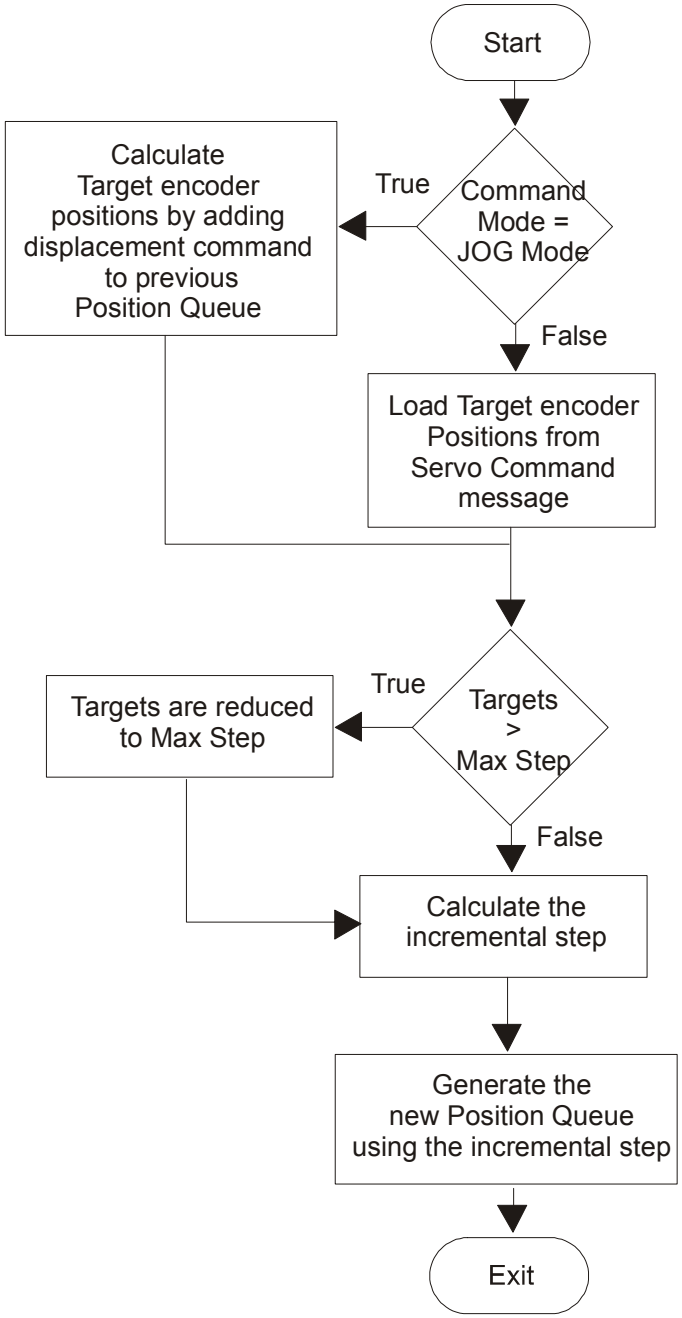


Figure 4.9. Flowchart of the Servo Planner

4.1.5.3. The Homing Routine

The Homing function is responsible for initialization of the robot. Since the robot can be in any configuration at power up, a known position must be found so that all movements can be made relative to it. The zero position corresponds to the known position of the joints where internal limit switches are activated. These internal limit switches are accessible via two digital output registers on the expansion cards. Again, these memory locations are accessible via VMEbus device driver functions compiled with the servo program. A state machine algorithm has been implemented which moves one joint at a time.

The Homing function begins by reading the digital output registers and masking each bit to obtain the state of each joint limit switch. Each joint is searched in series using a set of IF statements. If the limit switched is activated initially, we assume that the position of the joint is past the zero position. A "back-off" latch is set and a single relative joint encoder position command of "moderate speed" (20 encoder counts) in the opposite direction is sent into the encoder command cache to be executed at a rate of 500Hz by the servo function. Once the joint switch is no longer active, the "back-off" latch is off and a "moderate speed" command is sent in the forward direction until the joint switch is active. Then a single encoder count command is sent in the reverse direction until the limit switch is inactive. This procedure establishes the exact location in encoder counts of the limit switch and this value added to the known encoder counts to the home position is stored as the Encoder Position Zero. Thus when a 0 is written to the command cache, the robot will move to the home position. This procedure is executed for each joint until all switches are found, then the Home Flag is finally set true.

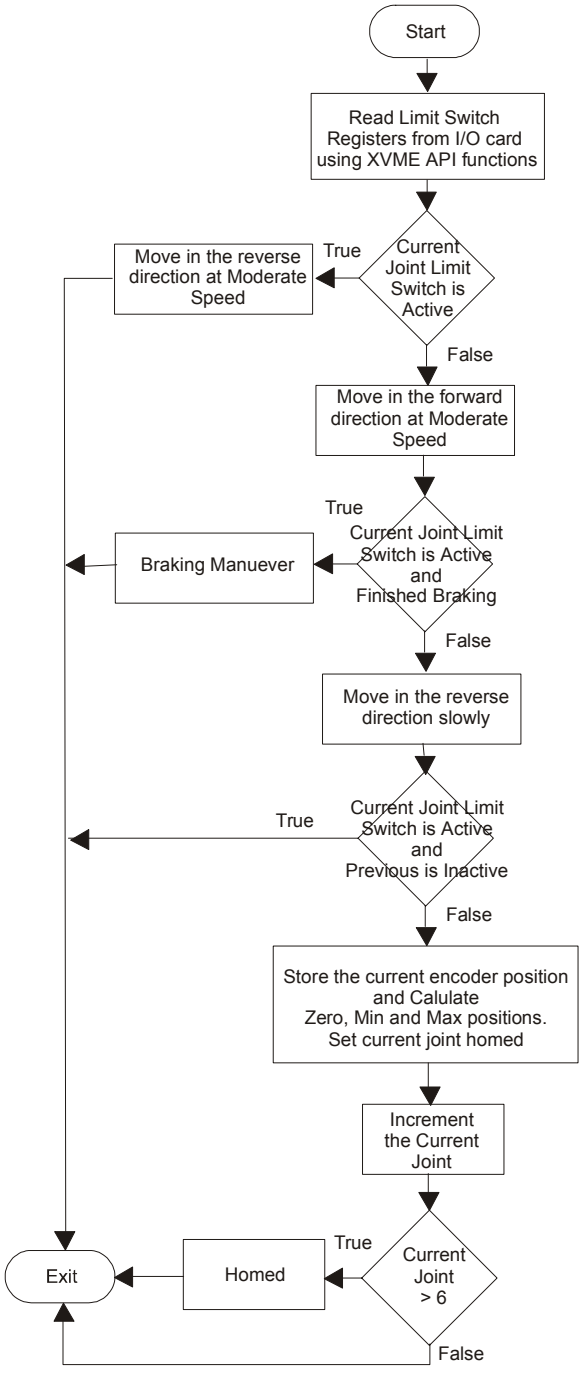


Figure 4.10. Flowchart of the Homing Function

4.1.6. The Servo Agent

The Servo Agent program serves as an isolation mechanism for the generic motion controller. Since high-level interprocess communication is achieved via a shared memory system, all processes must be of the same priority to avoid lockups. This occurs when a higher priority process preempts a lower process who has sole ownership of shared memory semaphore yet still requires access to the shared memory. Hence it will never gain access to the shared memory since the lower process cannot continue to release the semaphore to the higher process.

Since the servo function must remain at a higher priority because it is handling hardware, an equal priority servo agent is executed which copies pertinent data from the shared memory on the servo function's behave and delivers it at a fixed rate of 60Hz to the Servo program in the form of a command message. Figure 4.11. illustrates the functionality of the Servo Agent in the form of a flowchart.

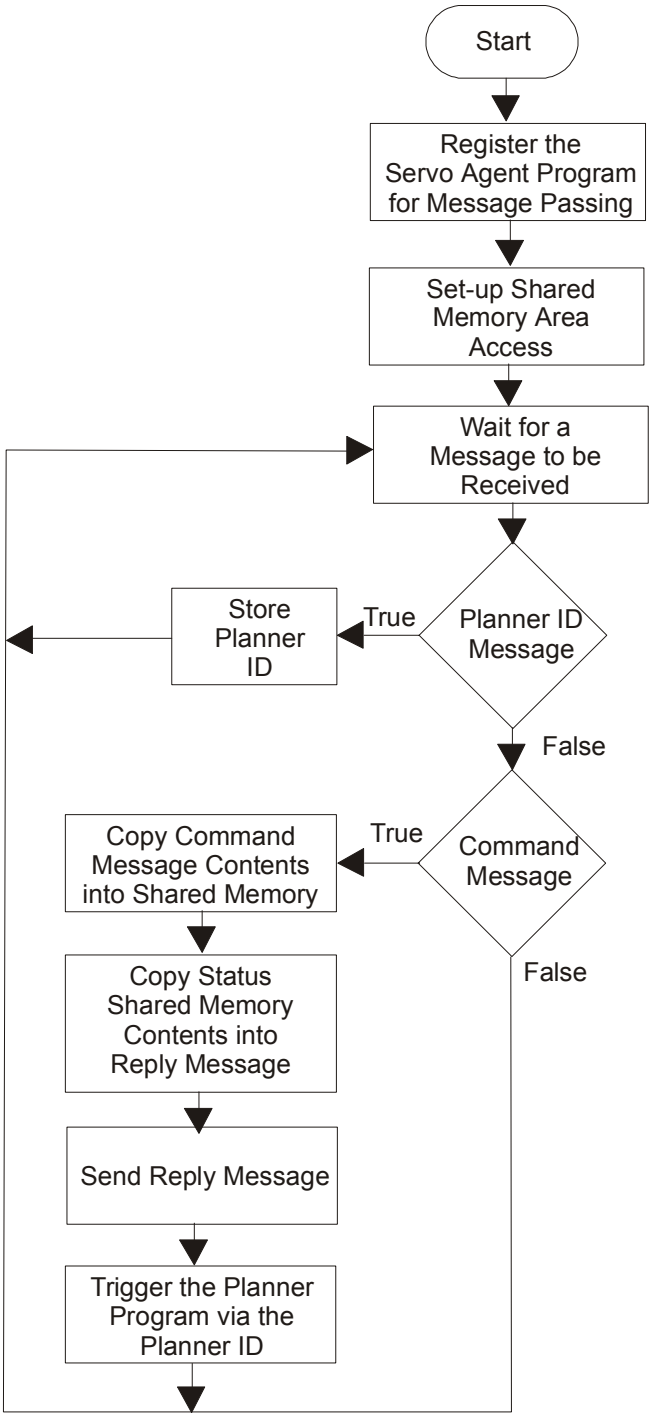


Figure 4.11. Flowchart of the Servo Agent

4.1.7. The Planner

The Planner program is executed only after the robot has been homed. Its main purpose is to convert both cartesian and joint space move commands into smaller joint space moves. The program consists of a main execution loop which calls specific functions based on the command flags in the shared memory areas. Header or include files are used to centralize the Reis Robotstar V15's specific hardware related data. Link lengths as well as calibration data and joint gearing ratios are kept in the reis_defs.h file. The planner programs execution loop is controlled via the heartbeat from the Brain Computer which is propagated to the planner program via the Reis Agent program. Figure 4.12. illustrates the execution flow of the planner program.

The program begins execution by attempting to locate the Reis Remote program. This is followed by the Planner program registering its own unique identifier into the system's namespace for Native Message Passing. Once registration is complete, the Command and Status shared memory areas are opened for access. Before the main execution loop is entered, the connection to the heartbeat is established by sending the Planner's Process Identification Number to the Reis Remote program. Upon each elapsed heartbeat, a message will now be sent to the planner program on behalf of the heartbeat from the Reis Remote.

Once inside the main execution loop, the Command and Status shared memory are accessed via the read_shmem() function. This function copies all pertinent information into a local data structure identical to the shared memory areas. Of particular interest is the command mode and the home flag. If the robot has not been homed, execution for this heartbeat cycle is terminated. Otherwise, the command mode flag is checked for a cartesian command set or a joint-space command set. If a cartesian command set is issued, a loop is entered where the inverse kinematics are solved 8 times, one for each possible robot configuration, using the cart_to_joint() function. As each solution is determined in joint angles, it is converted into joint encoder counts using the joint_to_act() function. These converted joint solutions are placed inside a solution array. As well, these joint solutions are checked to see if they are physically reachable. If they aren't reachable, the solution is given a corresponding invalidation flag.

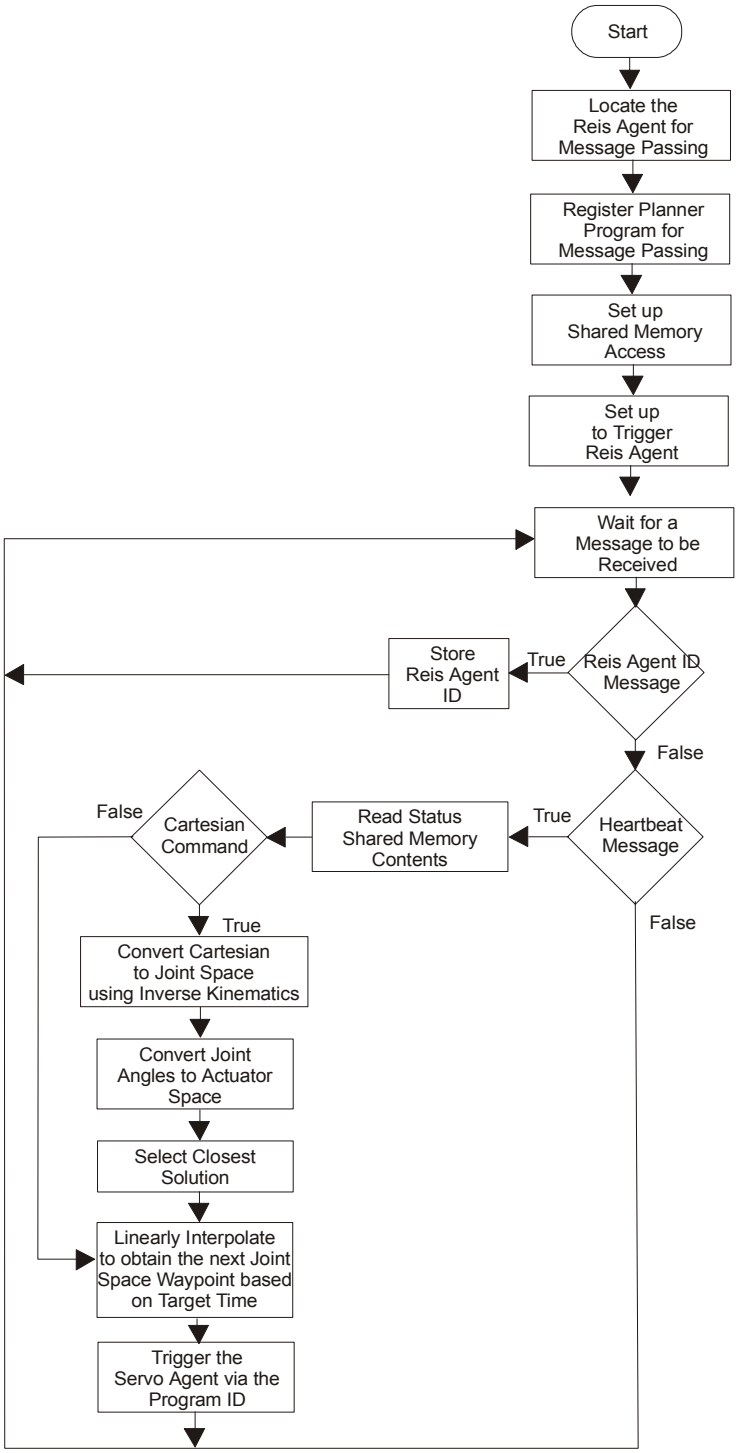


Figure 4.12. Planner Program

```
int cart_to_joint (float x, y, z, *Rot_ptr, int solution, *j1_ang_ptr,
      *j2_ang_ptr, *j3_ang_ptr, *j4_ang_ptr, *j5_ang_ptr, *j6_ang_ptr)
{

/* Creating A-Matrix (ZYZ-Orientation Matrix)*/
  for(i=0; i<4;i++)
    for(i=0; i<4;i++)
      if (j == 3)
     {
        if (i == 0) A[0][3] = x;
        else if (i == 1) A[0][3] = y;
        else if (i == 1) A[0][3] = z;
     }
      else
        A[i][j] = *Rot_ptr; Rot_ptr++;

/* Inverse Kinematic Solution */
  theta1 = atan2(A[1][3], A[0][3]);
  d1     = A[2][3] - Link0;

if ((solution == 0)||(solution == 1)||(solution ==2)
     ||(solution == 3))
 d2 = sqrt(A[0][3]*A[0][3] + A[1][3]*A[1][3]);
else
 d2 = -1*sqrt(A[0][3]*A[0][3] + A[1][3]*A[1][3]);

D = (d1*d1 + d2*d2 - L1*L1 - L2*L2)/(2*L1*L2);

if ((solution == 0)||(solution == 1)||(solution == 4)
    ||(solution == 5))
 Ds = sqrt(1-D*D);
else
 Ds = -1*sqrt(1-D*D);

theta3 = atan2(D, Ds);
L2c    = Link2*cos(theta3);
L2s    = Link1 + Link2*sin(theta3);
Alpha  = atan2(L2c, L2s);
Beta   = atan2(d1, d2);
theta2 = Alpha + Beta - HALF_PI;

/* Compute Inverted end-of-arm frame ZYZ-orientation matrix*/
Ctheta1 = cos(theta1); Stheta1 = sin(theta1);
Ctheta2 = cos(theta2); Stheta2 = sin(theta2);
Ctheta3 = cos(theta3); Stheta3 = sin(theta3);

R[0][0] = -Ctheta1*(Stheta2*Ctheta3 + Ctheta2*Stheta3);
R[0][1] = -Stheta1*(Stheta2*Ctheta3 + Ctheta2*Stheta3);
R[0][2] =  Ctheta2*Ctheta3 - Stheta2*Stheta3;

R[1][0] =  Stheta1;
R[1][1] = -Ctheta1;

R[2][0] = -Ctheta1*(Stheta2*Stheta3 - Ctheta2*Ctheta3);
R[2][1] = -Stheta1*(Stheta2*Stheta3 - Ctheta2*Ctheta3);
R[2][2] =  Ctheta2*Stheta3 + Stheta2*Ctheta3;
```

```
/* Compute the wrist orientation Matrix */
O[0][0] = R[0][0]*A[0][0] + R[0][1]*A[1][0] + R[0][2]*A[2][0];
O[0][1] = R[0][0]*A[0][1] + R[0][1]*A[1][1] + R[0][2]*A[2][1];
O[0][2] = R[0][0]*A[0][2] + R[0][1]*A[1][2] + R[0][2]*A[2][2];

O[1][0] = R[1][0]*A[0][0] + R[1][1]*A[1][0] + R[1][2]*A[2][0];
O[1][1] = R[1][0]*A[0][1] + R[1][1]*A[1][1] + R[1][2]*A[2][1];
O[1][2]  =  R[1][0]*A[0][2]  +  R[1][1]*A[1][2]  +
R[1][2]*A[2][2];

O[2][0]  =  R[2][0]*A[0][0]  +  R[2][1]*A[1][0]  +
R[2][2]*A[2][0];
O[2][1]  =  R[2][0]*A[0][1]  +  R[2][1]*A[1][1]  +
R[2][2]*A[2][1];
O[2][2]  =  R[2][0]*A[0][2]  +  R[2][1]*A[1][2]  +
R[2][2]*A[2][2];

/* Compute Z-Y-Z angles for wrist orientation */
if ((solution == 0)||(solution == 2)||(solution == 4)
     ||(solution == 6))
  Rr = sqrt(O[2][0]*O[2][0] + O[2][1]*O[2][1]);
else
  Rr = -1*sqrt(O[2][0]*O[2][0] + O[2][1]*O[2][1]);

Beta = atan2(Rr, O[2][2]);
if (Beta != 0)
{
  Sb    = sin(Beta);
  Alpha = atan2(O[1][2]/Sb,  O[0][2]/Sb);
  Gamma = atan2(O[2][1]/Sb, -O[2][0]/Sb);
}
else
{
  Alpha = 0.0;
  Gamma = atan2(-O[0][1], O[0][0]);
}
/* Translating Final Outputs */
  *j1_ang_ptr = theta1*RAD_TO_DEG;
  *j2_ang_ptr = theta2*RAD_TO_DEG;
  *j3_ang_ptr = theta3*RAD_TO_DEG;
  *j4_ang_ptr = Alpha*RAD_TO_DEG;
  *j5_ang_ptr = Beta*RAD_TO_DEG;
 *j6_ang_ptr = Gamma*RAD_TO_DEG;
return 0;
}

int joint_to_act ( float j1_ang, j2_ang, j3_ang, j4_ang,
                   j5_ang, j6_ang,
                   int *j1_ptr, *j2_ptr, *j3_ptr,
*j4_ptr, *j5_ptr, *j6_ptr)
{
  j1 = -j1_ang * J_GEAR;
  if ((j1 < J1_MIN) || (j1 > J1_MAX)) return 1;

  j3 = j3_ang * J_GEAR;
  j2 = j2_ang * J_GEAR + j3/100;
  if ((j2 < J2_MIN) || (j2  > J2_MAX)) return 2;
```

```
  j3 = j3_ang * J_GEAR + j2/100;
  if ((j3 < J3_MIN) || (j3 > J3_MAX)) return 3;

  j5 = j5_ang * J_GEAR;
  j4 = -j4_ang * J_GEAR + j5/100;
  if ((j4 < J4_MIN) || (j4 > J4_MAX)) return 4;

  j5 = j5_ang * J_GEAR - j4/100;
  if ((j5 < J5_MIN) || (j5 > J5_MAX)) return 5;

  j6 = j6_ang * J_GEAR;
  if ((j6 < J6_MIN) || (j6 > J6_MAX))
  return 6;

  *j1_ptr = j1; *j2_ptr = j2; *j3_ptr = j3; *j4_ptr = j4;
  *j5_ptr = j5; *j6_ptr = j6;
  return 0;
}
```

After all solutions have been found and checked, a single solution is selected using the pick_soln() function. The pick_soln() function assigns each valid solution a score based on the overall number of encoder counts needed to reach the solution. It is possible to weight each joint so as to favor a certain configuration. In our implementation we added weights to joints 1, 2 and 3 so that movements about these joints would not be favored. Equation 4.15. shows the score function where the k values represent the joint weights..

$$Score_n = \sum_{i=1}^{6} k_i \left| jpos_{ni} - jpos_{previ} \right| \qquad (4.15.)$$

```
int pick_soln(int *soln_set, int *invalid,
              int *prev_j_ang, float *j_gain)
{
  int i,j, first, solution;
  int soln[8][6], invalid_idx[8]; int prev[6];
  float K[6]; float move_score, min_move_score;

/* Loading into local cache */
  for (i=0; i<8; i++)
  {
    for (j=0; j<6; j++)
    {
      soln[i][j] = *soln_set;
      soln_set++;
    }
    invalid_idx[i] = *invalid;
```

```
      invalid++;
    }
    for (i=0; i<6; i++)
    {
      K[i] = *j_gain;
      prev[i] = *prev_j_ang;
      j_gain++; prev_j_ang++;
    }

/* Decision Function */
  solution = -1;
  first = 1;
  for (i=0; i<8; i++)
  {
    if (!invalid_idx[i])
    {
      move_score = fabs(soln[i][0]-prev[0])*K[0] +
                   fabs(soln[i][1]-prev[1])*K[1] +
                   fabs(soln[i][2]-prev[2])*K[2] +
                   fabs(soln[i][3]-prev[3])*K[3] +
                   fabs(soln[i][4]-prev[4])*K[4] +
                   fabs(soln[i][5]-prev[5])*K[5];
      if ((move_score < min_move_score) || (first))
      {
        first = 0; solution = i;
        min_move_score = move_score;
      }
    }
  }
  return solution;
}
```

Once the joint space solution has been determined, the joint-space interpolation routine or plan_joint_cmd() function is executed. The plan_joint_cmd() function takes the joint-space command and interpolates the next joint-space move which will be accomplished in 16.7ms (60Hz). The time count is then decremented so that the number of steps required to finish the move can be monitored. These encoder commands are then copied into the internal encoder command registers of the shared memory area where the Servo Agent program will pick up the commands and send them to the Servo program to initiate movement of the Reis Robotstar V15.

```
int plan_joint_cmd (void)
{
  int step, i;

/* Calculate the Step */
  for (i=0; i < NUM_JOINTS; i++)
    if (local_cmd.joint_time_count > 1)
    {
```

```
        step = (local_cmd.encoder_cmd_ext[i] -
                local_stat.encoder_cmd[i])/local_cmd.joint_time_count;

       local_cmd.encoder_cmd_int[i] = local_stat.encoder_cmd[i] + step;
      }
      else
        local_cmd.encoder_cmd_int[i] = local_cmd.encoder_cmd_ext[i];

    if (local_cmd.joint_time_count > 0)
      local_cmd.joint_time_count--;

  /* Sending Command Information */
    sem_wait(&cmd_ptr->semaphore);
    for (i=0; i < NUM_JOINTS; i++)
    {
      cmd_ptr->encoder_cmd_ext[i] = local_cmd.encoder_cmd_ext[i];
      cmd_ptr->encoder_cmd_int[i] = local_cmd.encoder_cmd_int[i];
    }
    cmd_ptr->joint_time_count = local_cmd.joint_time_count;
    cmd_ptr->coord = JSPACE;
    sem_post(&cmd_ptr->semaphore);

  /* Updating the Status Information */
    sem_wait(&stat_ptr->semaphore);
    stat_ptr->busy = local_cmd.joint_time_count;
    sem_post(&stat_ptr->semaphore);

    return 0;
  }
```

4.1.8. The Reis Agent and the Remote Reis Programs

The Reis Agent program resides on the motion control computer while the Remote Reis program resides on the Brain Computer. These two programs form the network link necessary to deliver data from the Brain Computer to the Reis motion controller and vice-versa.

The Remote Reis program begins by attaching to the heartbeat program. Each time a heartbeat message is sent to the Remote Reis program it copies its contents from the Reis's shared memory area of the Brain Computer into a Native Message Passing packet. This packet contains command data and is then sent to the Reis Agent program across the Ethernet link.

Once the package is received by the Reis Agent program, the command data is copied into the command shared memory area of the Reis motion controller. Status data from the motion controller is also copied and packed into a Native Message Passing packet. This packet is sent back to the Remote Reis program to form a reply. In addition, a heartbeat relay message

is sent to the Planner program to indicate that a new command package has been copied into the shared memory area.

Once the Remote Reis program has received the status data from the motion controller, it uses the forward kinematic model for the Reis Robot to calculate the current cartesian position and orientation matrix of the wrist-center point. This new status data is then copied into the Status shared memory area of the Brain Computer.

4.2. GM Fanuc S400 Motion Controller

4.2.1. Overview

The GM Fanuc S400 motion controller represents a partial implementation of the generic motion controller described in Section 4.1. Due to the proprietary nature of the bus architecture, only access to the encoders on the Fanuc controller has been attempted. Gross movement commands are achieved using the Fanuc controller's built-in RS-232 serial interface.

Designed as an automotive welding/lifting robot for General Motors in the late 1980's, its controller design was meant for the harsh environment of a factory. Its programming language came in the form of an interpreted high level language similar in syntax to BASIC; this programming language is called KAREL. The KAREL programs can be written on the controller directly or written offline and sent to the controller via the RS-232 serial interface. Although the serial interface was meant to be used primarily for file transfer, it also allows data to be sent to and from the controller. By writing a KAREL serial server program on the controller which sends its cartesian location, receives cartesian commands and translates these data points into KAREL commands, a simple off-board controller can be built. The main disadvantage to this approach is that the Karel interpreted language is computationally expensive and its processor is not powerful enough to handle such a processor intensive task. Thus, a bi-directional server program could not be used to build an off-board controller.

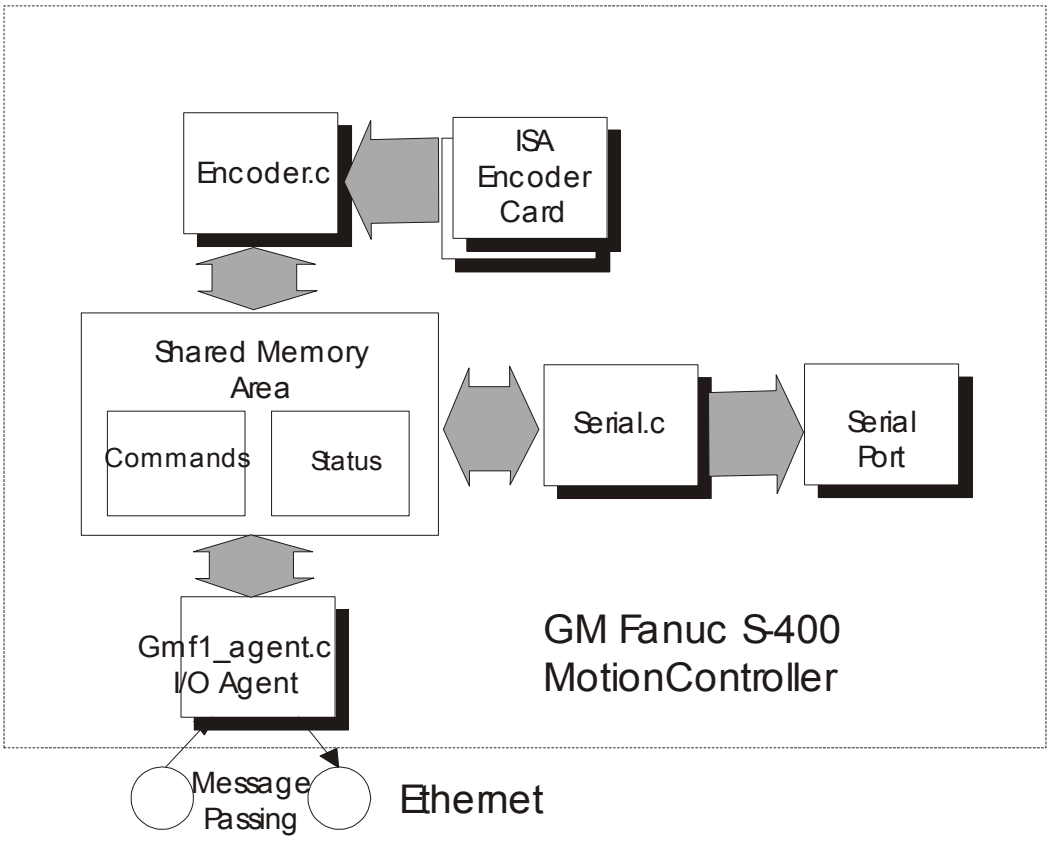


Figure 4.13. Anatomy of the GM Fanuc S400 motion control software

The design approach taken in building a generic motion controller interface for the GM Fanuc S400 to the DROID system was to build a highly optimized KAREL server which receives command data via the serial interface and translates the data into cartesian move commands. It is fortunate that the Fanuc S400 Controller is modular in terms of its internal circuit design and allows encoder signals to be read from a port located on each of its axis control circuitry. These single-end encoder signals are sent to two ISA encoder card located on the DROID generic motion control interface computer. Figure 4.13. illustrates a detailed schematic of the anatomy of the GM Fanuc S-400's motion control software which resides on the interface computer.

This implementation of generic motion control for the GM Fanuc S400 was built by

Crymble [30] and is referred to as GMF1. The following discussion is limited to the operational characteristics and data flow of the software modules which make up the GMF1 Motion Controller.

4.2.2. Shared Memory

The shared memory area residing on the GMF1 is similar to the Reis Robotstar V15's motion controller in that they contain both a Command and Status shared memory area (see Table 4.2. and Table 4.3). The Command shared memory area holds both cartesian and joint space commands as well as homing commands. The Status shared memory area contains current cartesian location and orientation and joint encoder counts as well as status flags. Access is granted by first obtaining a semaphore signal and relinquishing it upon terminating the access. Since the database can be monopolized during accessing it is important that every accessing module must have equal priority so that another module cannot interrupt during its access thus locking the shared memory area.

4.2.3. GMF1 Agent and Remote GMF1 programs

To interface to the DROID system's Brain Computer, a network link is necessary to deliver data from the Brain Computer to the GMF1 motion controller and vice-versa. This is provided by the GMF1 Agent and Remote GMF1 programs. These programs are similar to those of the Reis Robotstar V15's motion controller. The GMF1 Agent program resides on the motion control computer while the Remote GMF1 program resides on the Brain Computer.

The Remote GMF1 program begins with attaching to the heartbeat program. Each time a heartbeat message is sent to the Remote GMF1 program, it copies its contents from the GMF1's shared memory area of the Brain into a native message passing packet. This packet contains command data and is then sent to the GMF1 Agent program across the Ethernet link.

Once the package is received by the GMF1 Agent program, the command data is copied into the command shared memory area of the Motion Controller Computer. Status data from the motion control computer is also copied and packed into a Native Message Passing packet. This packet is sent back to the Remote GMF1 program to form a reply. In addition, a heartbeat relay message is sent to the Serial program to indicate that a new command package has been copied into the shared memory area.

4.2.4. The Serial Program

Communication between the GMF1 motion control computer and the Fanuc S400 controller is achieved using a serial link. The Fanuc S400 controller is equipped with a RS-232 25-pin serial port which has a bandwidth of 9600 bps.

The Serial program begins execution by initializing the serial port. It then waits for a relay message from the GMF1Agent program indicating that a new command has been placed in the Command shared memory area. It then obtains the semaphore to gain access to the Command shared memory area and copies the contents into a local cache and finally relinquishes control of the shared memory area. Since the command data is in the form of a transformation matrix, it must be converted into global translations and Euler rotation angles so that the KAREL Serial Server can generate the appropriate move commands. This command set is sent to the KAREL Serial Server as ASCII characters separated by whitespaces.

The incoming messages from the Brain Computer are sent at a rate of 60Hz which corresponds to the DROID system heartbeat. Due to the underpowered processor resident on the Fanuc Controller (Motorola 68000 at 16MHz), a 60Hz cycle rate for the KAREL Serial Server program is not possible. The strain of the KAREL interpreted language limits the cycle rate of the KAREL Serial program to 6Hz even under optimized conditions. Hence, the Serial program must ignore the other 9 commands and only send the most recent command set to the Fanuc controller once it has completed its previous move.

4.2.5. The Encoder Program

The main function of the GM Fanuc S400 is to provide workpiece positioning for the DROID Seam Tracking application. Its payload capabilities make it a good candidate for such labor. Furthermore, due to the control rate limitations of its controller, it must perform gross movements at a rate of 6Hz while the Reis Robotstar V15 provides tool positioning at 60Hz. Although there is a large discrepancy in control rates, the 2 robots can cooperate and work at the 60Hz as long as the slower robot can provide status information concerning its position and orientation at 60Hz.

To provide positional and orientation data of the Wrist-Center-Point (WCP) at 60Hz or greater, each joint encoder signals was sampled using an ISA encoder board, as shown in Section 2. The Encoder program is device driver/server which initializes the 2 ISA encoder boards and reads the contents of each encoder register at a rate of 120Hz. Each joint encoder count is then mapped into a joint angle which follows the Denavit-Hartenberg convention. Using a forward kinematic model, the 6 joint angles are converted into a transformation matrix which is placed into the status memory area. This data is then sent to the DROID Brain Computer by the GMF1 Agent on the next heartbeat pulse.

The Encoder program operates at 120Hz using an independent timer so as to isolate it from the rest of the DROID system. It samples at least twice the heartbeat cycle to ensure that the transformation matrix is the most recent.

Chapter 5.

Seam Tracking Implementation

This chapter focuses on the Seam Tracking application program where a single line laser profile sensor is used to provide seam data to the multiple robotic system at a rate of 60Hz. The line laser responsible for profiling the seam is created using a laser diode and cylindrical lens to form a line. This line is cast across the joint or seam and the reflected profile is acquired by a gray-scale charged-coupled device camera (CCD). This image is acquired and digitally filtered using a digital signal processing board which resides on a host computer. This host computer interfaces the digital signal processing board with a given operating system thus allowing an application program to process and deliver the seam's positional data to the multiple robotic system via a serial link. The chapter begins with an explanation of the kinematics of seam tracking. Then we discuss the system software used to operate the MVS sensor and interface it to the DROID system. This chapter concludes with an explanation of the seam tracking task program.

5.1. Kinematics of Seam Tracking

One of the key objectives in Dynamic Seam Tracking is to map out a portion of the welding seam with respect to the workpiece coordinate system. Figure 5.1. illustrates an enlarged view of the seam area. In our application of Autonomous Multiple Robotic Welding, the seam feature that we are trying to identify takes the form of a "lap-joint" or a "stepped profile". The workpiece reference frame origin is coincident with the Wrist-Center-Point (WCP) of the workpiece holder robot (GM Fanuc S400) since it is a fixed point with respect to the workpiece and its pose can be computed from the forward kinematics of the workpiece holder robot. From Figure 5.1. the Sensor reference frame is fixed with respect to the Reis WCP frame and is related by the transformation ${}^{(Reis)6}T_{sensor}$. The seam feature is determined with respect to the sensor frame and can therefore be defined with respect to the Reis WCP frame using the following transformation equation.

$$ {}^{(Reis)6}T_{seam} = {}^{(Reis)6}T_{sensor}\ {}^{sensor}T_{seam} \tag{5.1.} $$

However, the seam location with respect to the weld robot is not adequate for mapping since the weld robot is moving with respect to the workpiece, hence its origin would be constantly changing. Therefore, the WCP of the workpiece holder robot is used since the workpiece is rigidly held in place by the workpiece holder robot.

To determine the seam location with respect to the WCP of the workpiece holder robot we refer to the kinematic diagram illustrated in Figure 5.2. Note the direction of the arrows dictate the relativity of each reference frame. Since each robot's WCP transformation is known, we can determine the seam position with respect to the workpiece holder wrist center with the following Transformation Equations.

$$ {}^{(Reis)0}T_{global}\ {}^{global}T_{(GMF1)0}\ {}^{(GMF1)0}T_{6}\ {}^{(GMF1)6}T_{seam} = {}^{(Reis)0}T_{6}\ {}^{(Reis)6}T_{seam} \tag{5.2.} $$

Or

$$ {}^{(GMF1)6}T_{seam} = ({}^{(Reis)0}T_{global}\ {}^{global}T_{(GMF1)0}\ {}^{(GMF1)0}T_{6})^{-1}\ {}^{(Reis)0}T_{6}\ {}^{(Reis)6}T_{seam} \tag{5.3.} $$

Similarly we can determine the current Tool-Tip transformation with respect to the workpiece holder WCP with the following transformation equations.

$$ {}^{(Reis)0}T_{global}\ {}^{global}T_{(GMF1)0}\ {}^{(GMF1)0}T_{6}\ {}^{(GMF1)6}T_{tool} = {}^{(Reis)0}T_{6}\ {}^{(Reis)6}T_{tool} \quad (5.4.) $$

Or

$$ {}^{(GMF1)6}T_{tool} = ({}^{(Reis)0}T_{global}\ {}^{global}T_{(GMF1)0}\ {}^{(GMF1)0}T_{6})^{-1}\ {}^{(Reis)0}T_{6}\ {}^{(Reis)6}T_{tool} \quad (5.5.) $$

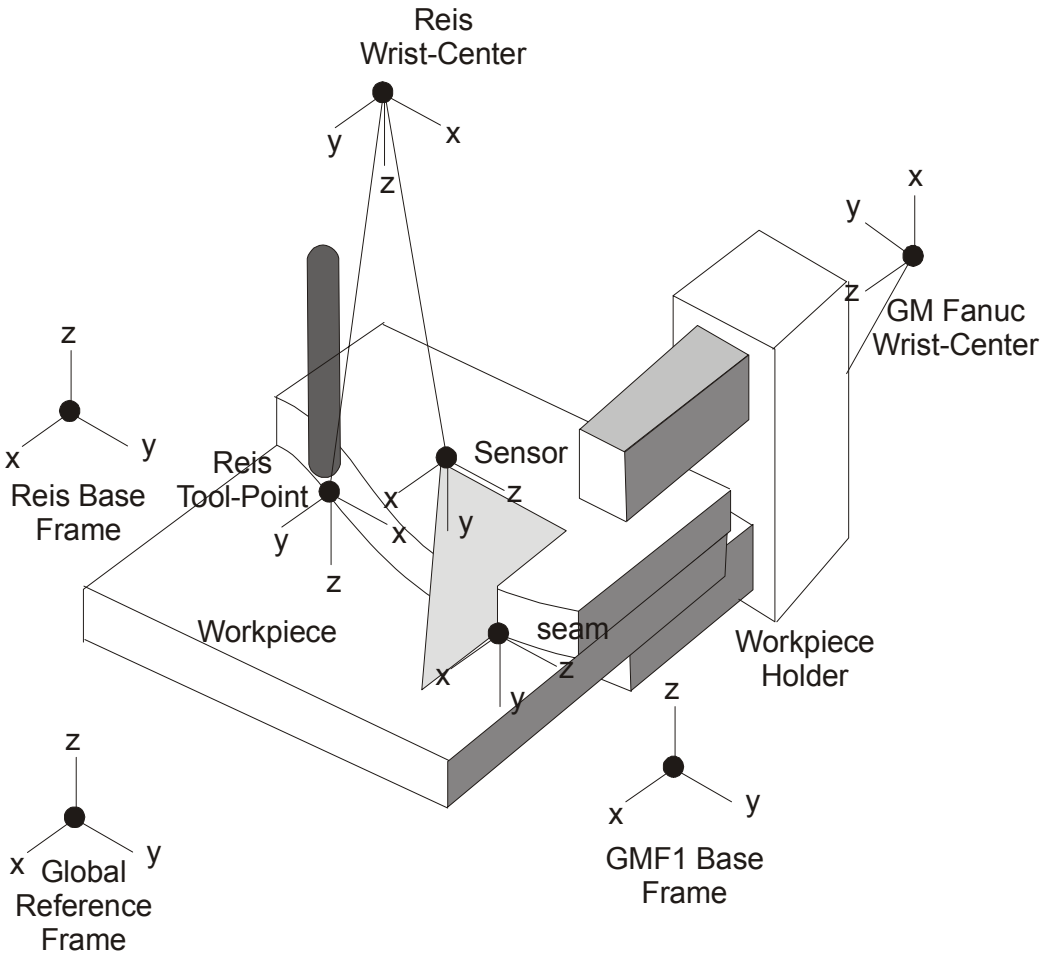


Figure 5.1. Reference frames used in Seam Tracking software

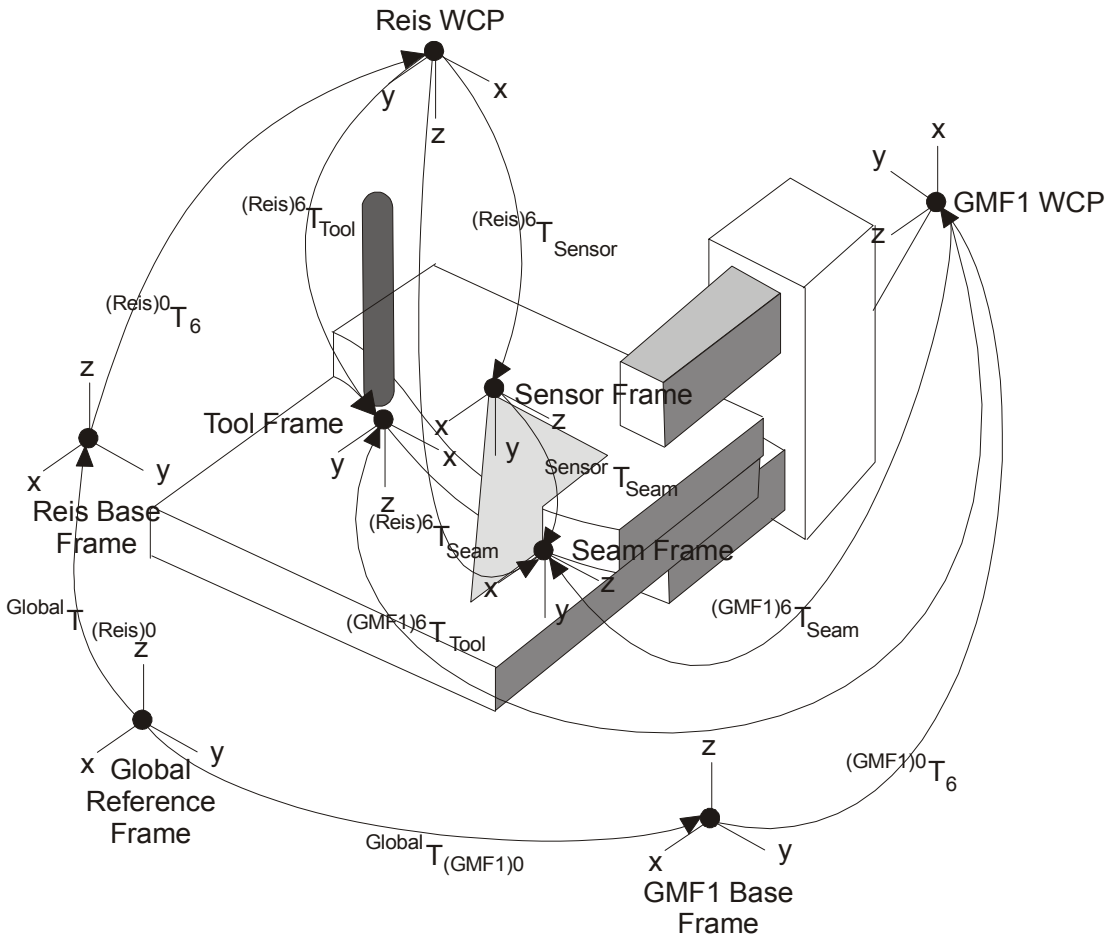


Figure 5.2. Kinematic Diagram

The seam position and orientation with respect to the workpiece holder WCP is captured at the "heartbeat" frequency and stored in a dynamic queue or dynamic seam patch. This patch of the workpiece represents the seam from the camera to the tool point as shown in the enlarged view of Figure 5.1. The next section describes the determination of the move commands for both the welding robot and the workpiece holder robot.

5.2. Coordinated Motion Control

The objective of any tracking system is to control motion such that a desired point is coincident with the target. Dynamic Seam Tracking is no different, however as more degrees of freedom are involved in the system, more choices are available as to the approach to the target. In this implementation of Autonomous Multiple Robotic Welding, two 6 degrees of freedom robot arm manipulators are used to give a total of 12 degrees of freedom. Hence, there are a very large number of possible orientations that both the welding robot and workpiece holder robot can take to make the welding tip coincide with the seam position and orientation. This is a very desirable situation, especially for welding since it is always possible to reach the optimal orientation for welding. However, rules must be put into place such that the optimal orientation for welding is always selected from the large number of possible solutions. This section describes the process of determining the target location from the captured seam data resident in the dynamic seam patch and the determination of the motion commands for both the welding robot as well as the workpiece holder robot.

The dynamic seam patch contains the position and orientation of the seam with respect to the workpiece holder wrist center from the point directly below the current camera position to the current tool tip location. Therefore, as the welding robot moves past a seam position it is discarded and as new seam positions are acquired they are added to the patch. In addition, the number of captured positions in the patch or resolution is dynamic. Since we are acquiring seam data at the"heartbeat" rate (60Hz), lower travel velocity will yield higher resolutions. Hence, during the initial acceleration phase of welding, the resolution will be high as the inertia of motion leads to low velocities (large patch), however, as the velocity reaches the target velocity the resolution will decrease or the patch will shorten.

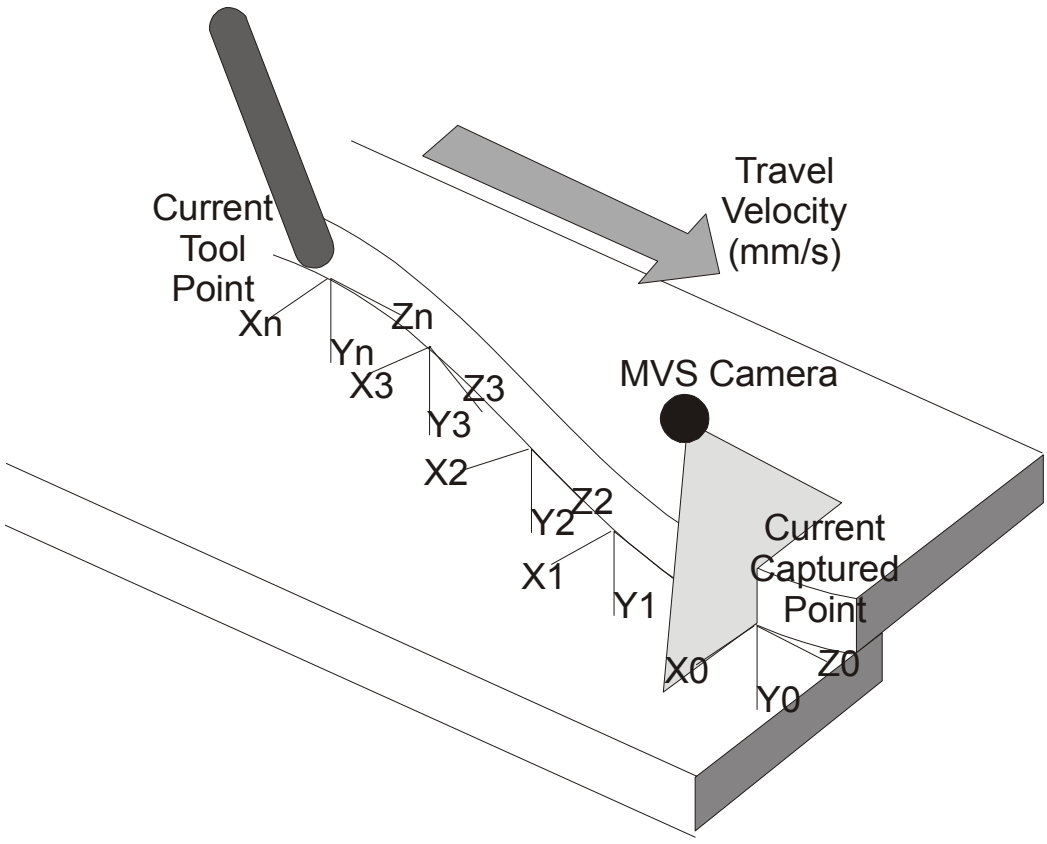


Figure 5.3. The Dynamic Seam Patch

Due to the dynamic nature of the seam patch, the seam must be searched in order to determine the target seam position and orientation for the current control cycle. The search begins at beginning of the patch which represents the current camera position on the seam. The patch is then traversed sequentially towards the current tool tip position with respect to the workpiece holder robot. The target seam position is actually a linearly interpolated position based on the linear distance required to meet the travel velocity specified by the user in mm/s. Therefore, each seam position in the patch is evaluated to determine its linear distance from the current tool tip position. Once the two bounding seam positions are determined, the target seam position is linearly interpolated. In order to compute the orientation portion of the transformation matrix, the transformation matrix is first rotated at the roll angle determined from the MVS

sensor. The yaw axis rotation angle is computed from Equation 5.6. while the pitch angle is calculated using Equation 5.7. By multiplying each angle by a gain prior to rotating the transformation matrix, the yaw and pitch compensation behavior of our system can be controlled so that the seam feature can be maintained within the camera throughout seam tracking.

$$\text{Yaw Angle} = \text{atan}^{-1}\left((x_0 - x_n)/\text{dist\_sensor\_to\_tool}\right) \quad (5.6.)$$

$$\text{Pitch Angle} = \text{atan}^{-1}\left((y_0 - y_n)/\text{dist\_sensor\_to\_tool}\right) \quad (5.7.)$$

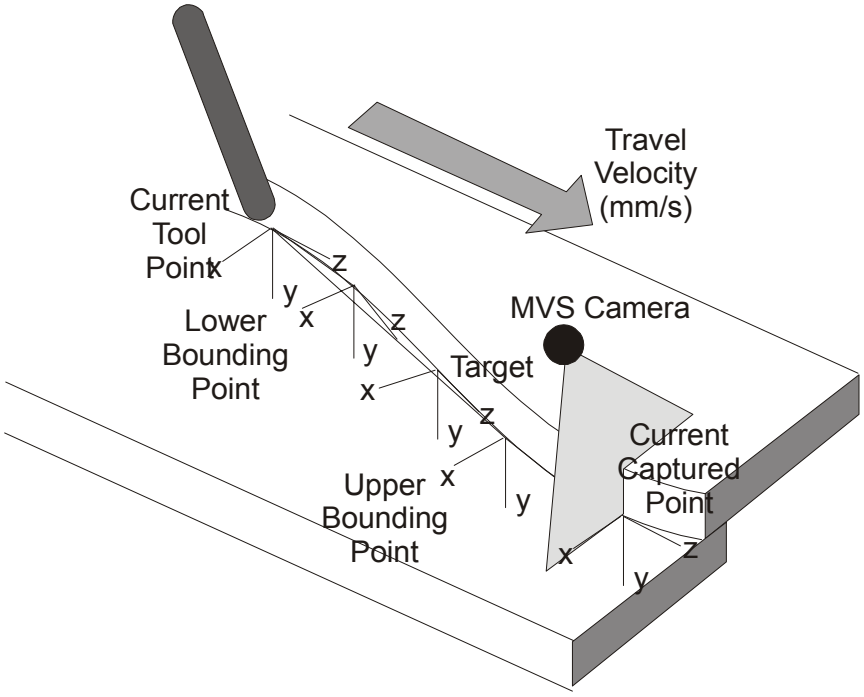


Figure 5.4. Interpolation scheme for determining target seam position and orientation

Once the Target Tool Tip transformation matrix has been determined it must be translated into a wrist center transformation matrix for the welding robot. Using the kinematic diagram shown in Figure 5.4., the wrist center transformation equation for the welding robot can be determined using the tool transformation with respect to the global reference frame.

$$ {}^{global}T_{(Reis)\,0l}\,{}^{(Reis)0}T_{6}\,{}^{(Reis)6}T_{tool} = {}^{global}T_{(GMF1)0}\,{}^{(GMF1)0}T_{6}\,{}^{(GMF1)6}T_{tool} \qquad \text{Equation 5.8.} $$

$$ {}^{(Reis)6}T_{tool} = ({}^{global}T_{(Reis)\,0l}\,{}^{(Reis)0}T_{6})^{-1}\,{}^{global}T_{(GMF1)0}\,{}^{(GMF1)0}T_{6}\,{}^{(GMF1)6}T_{tool} \qquad \text{Equation 5.9.} $$

Ideally this motion control action should be accomplished in one “heartbeat” count.

5.3. MVS System Software

The digital signal processing (DSP) board which captures and filters the video image of the seam profile is a proprietary product built by MVS Inc. (St. Laurent, Quebec). Its design with respect to hardware and software dates back to the mid 1990's and because of this, the host operating system is Microsoft DOS. Due to the proprietary nature of this DSP board, initialization is difficult to achieve without the optimized libraries compiled by MVS Inc. Hence, MVS’s compiled Control Program is used to initialize the video processor board prior to the execution of the application program. Figure 5.5. displays the Control Program’s graphical user interface to the MVS DSP board. It performs the proper initialization of the DSP board as well as providing read and write access to the DSP’s memory and configuration registers. This allows a variety of cameras and digital video filters to be used with the board. The configuration data required for each digital filter are stored in HEX files which reside in the executable’s directory.

The image from the RS-170 camera is captured at a resolution of 512X240 with a frame rate of 60 frames per second and a grayscale resolution of 256 levels. Since the incoming image contains a significant amount of information and noise, digital filtering must be applied to reduce the amount of extraneous data. A Finite-Impulse-Response (FIR) filter is used to identify and locate the seam feature, the result of this filtering is a 512X240 1-bit bitmap with a single pixel for each row representing the best approximation of the seam feature. This bitmap is placed in the memory banks of the DSP card and can be accessed at a rate of 60Hz after each frame has been captured.

Figure 5.5. The P1C30 Control Program

5.4. Application Software

The application software, called sensor.exe processes the filtered video data into a seam position and orientation and sends this data to the Brain Computer at a rate of 60Hz via the serial port. Figure 5.6. shows a screen capture of the sensor.exe program which graphically monitors the position and attitude of the surface with respect to the camera. It also displays the targeted feature which is a discontinuity across two lines where one side is higher than the other (a lap-joint). It also displays the frames processed/second upon exiting the program. The key components in sensor.exe includes the power2 assembly language routine which reads the video data from the MVS DSP board, the line fitting and discontinuity detection code which determines the seam's positional data, the graphics portion and the serial output which transmits the data to the Brain Computer.

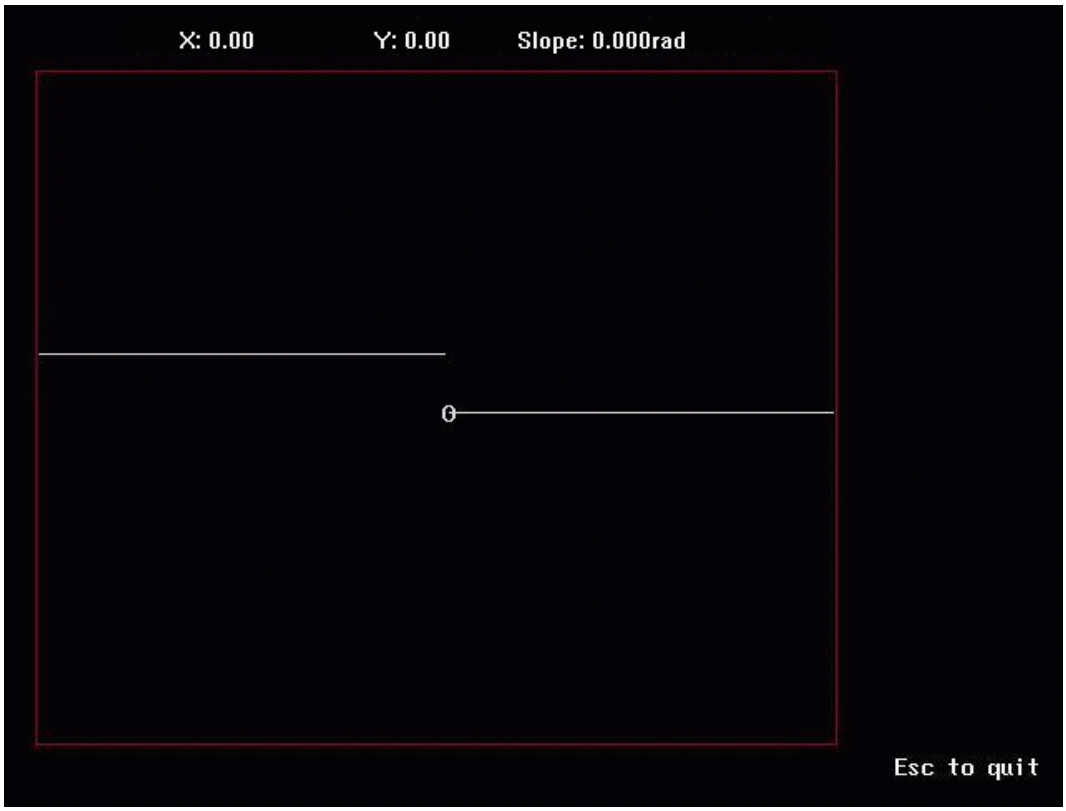


Figure 5.6. Screen Capture of the sensor.exe

The power2( ) function is an assembly routine located within the ml4.asm file, its object file must be linked in order to gain access to the MVS DSP board. The reason for this assembly routine is that board's memory contents are refreshed upon each video vertical retrace. This vertical retrace occurs every 16.7ms (60Hz) and memory must be accessed during this time. Therefore, accesses to the memory must take place within 1 ms (approximately) to ensure data integrity. Since conventional C-code cannot assure this, high performance assembly code is used to perform the memory copying.

The sensor.exe program begins by initializing the calibration coefficients for the particular sensor used. These coefficients allow the pixel data to be converted into real-world coordinates (mm and radians). The graphics system is then initialized and the user interface is

drawn. Next, the serial port (COM 1) is opened and is configured as an output device.

The main execution loop is where the data analysis is done. It begins by calling the power2( ) assembly routine which waits for the processor board memory to be free before accessing it. Therefore the execution loop's control rate is synchronized with the vertical retrace at 60Hz. The data read in is an array of 240 integers whose contents are the processor board's estimate to where the profile center lies along the screen row. There are 240 screen rows and 512 columns per access. This image corresponds to the interlaced image from the camera to give a combined resolution of 512X480 pixels.

The following section describes briefly the data analysis procedure developed by Strauss [5]which is implemented in sensor.exe. It begins by generating a histogram to determine the concentration of data points along a given column. This histogram indicates the spread of the pixel data and defines the limits of our line fitting algorithm. Next, the first derivative along a fixed interval of points is calculated. The largest first derivative should indicate the area where the discontinuity lies.

Once the feature is detected, two straight lines are fitted to the data on either side of the discontinuity point. These two straight lines approximate the surface and the slope of this line defines the roll angle relative to the camera frame. The program then converts this pixel position data into real-world coordinates using calibrated pixel-to-mm data. The data values are sent to the serial port as ASCII data which are delimited by commas. Finally the lines which represent the surface, the discontinuity target and the position and slope values are written to the user interface. The execution loop ends when an escape key is pressed and the frames processed/second is calculated.

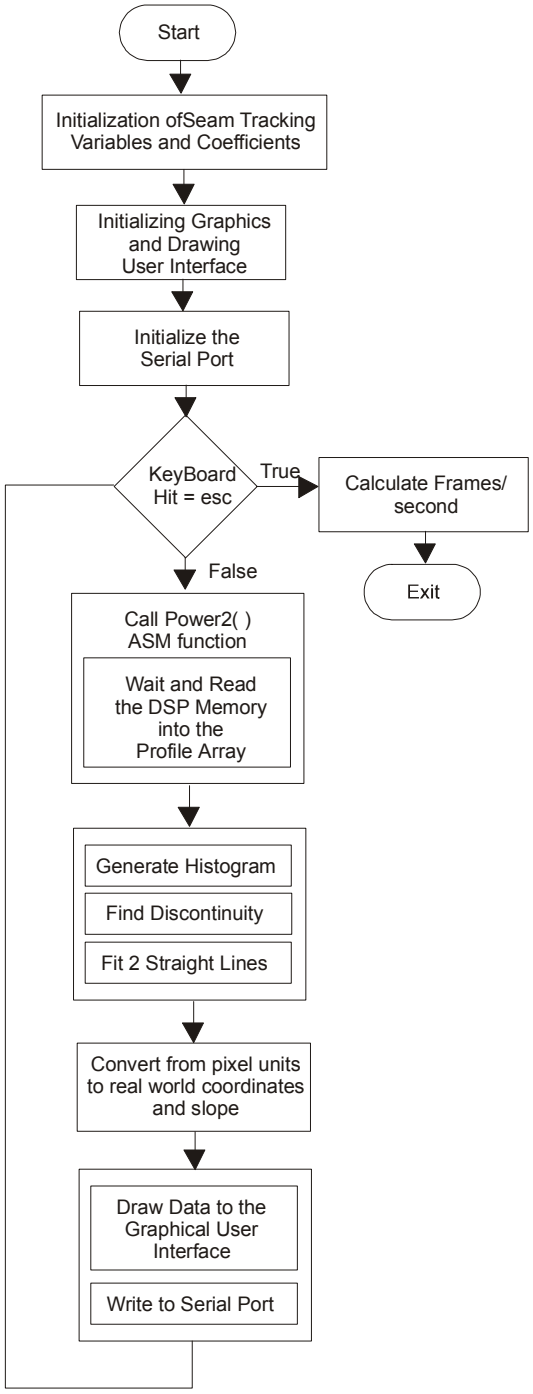


Figure 5.7. Flowchart of application software operation

5.5. Dynamic Seam Tracking Task Program

5.5.1. Overview

For our multiple robotic system to accomplish an autonomous coordinated weld, a Task program is needed to coordinate each robotic arm manipulator and process the incoming seam data from the sensor. This Task program is called "seamtrack" and resides on the Brain Computer.

Dynamic Seam Tracking is a technique used to accomplish a weld autonomously without preplanning the path for a robot to travel through. Using only sensor data and kinematic relationships, the seam tracking software will determine the optimal path dynamically as it receives new sensor data. The sensor need only be placed in the general viewable vicinity of the weld seam. Once the Dynamic Seam Tracking program is started it continues along the seam without further user intervention. However, some kind of user intervention is used to stop the seam tracking program.

Figure 5.8. illustrates the role of each subunit. The Reis Robotstar V15 acts as the welding robot. The GM Fanuc S400 acts as the workpiece holder and the MVS Line Laser Sensor acts as the seam sensor input element in this system. In theory, either robot manipulator can act as welding robot or workpiece holder. However, due to the finer degree of control attained on the Reis Robotstar V15 through the implementation of the generic motion control software, it is the preferred welding robot while the payload capabilities of the GM Fanuc S400 favor it as the workpiece holder.

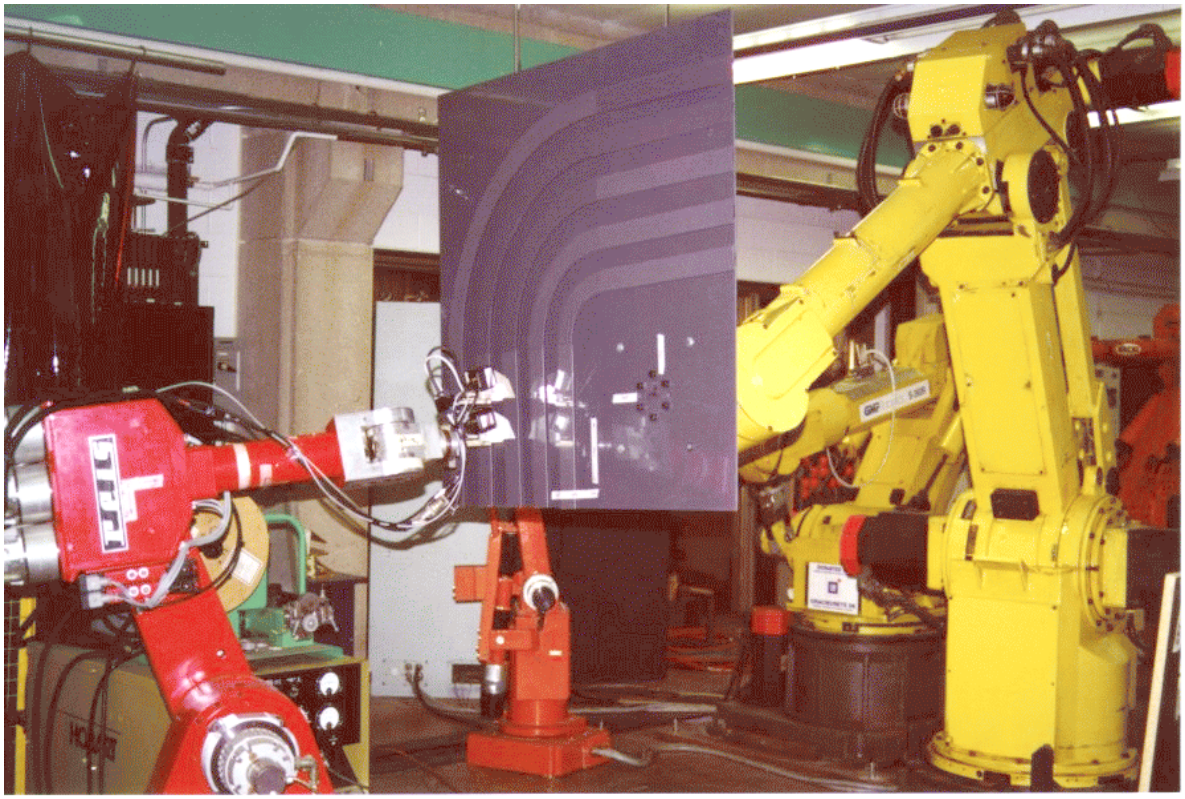


Figure 5.8. Hardware Setup for an Multiple Robotic Autonomous Welding

5.5.2. Software Implementation of Dynamic Seam Tracking

The Dynamic Seam Tracking Program is called “seamtrack” and resides on the Brain Computer. It is an independent compiled executable which can be run at the console with a single run-time argument. In this implementation, the runtime argument represents the desired travel velocity along the seam in mm/s. In order to execute properly, the shared memory areas of the subunits belonging to the i) Reis Robotstar V15 motion controller ii) GM Fanuc S400 motion controller and iii) the MVS Line Laser Sensor must be created and initiated. The “heartbeat” synchronization server must be operational. As well, the MVS Line Laser sensor must be brought into the viewable vicinity of the weld seam.

Figure 5.9. illustrates the major software components which make up the “seamtrack” program. The include header files allow access to robot specific geometry definitions and access to the shared memory areas. This allows robot geometry to be centralized in a common header file for ease of access and upgrading. From the diagram, it can be seen that each step in the Dynamic Seam Tracking technique has been coded into its own function. This allows every step to be upgraded with relative ease and facilitates debugging. This allows the “seamtrack” program to have the same simple and modular paradigm found throughout the DROID system.

The “seamtrack” program begins its execution by initializing key variables and flags as well as all transformation matrices using the robot geometry definitions found in the header files. This is followed by the decoding of the run-time arguments which in this case is a single argument representing the travel velocity of the seam tracking in mm/s. Next the locating of the “heartbeat” synchronization event server module is attempted every second. The program will wait indefinitely if the “heartbeat” is not present. Once the “heartbeat” is located, a triggering event known as a “proxy” is establish within the “seamtrack” program. This proxy’s identifier is then sent to the “heartbeat”. Each “heartbeat” expiration will then trigger the “seamtrack” proxy, this is how synchronization will be accomplished.

The next step is to map all required shared memory areas locally for read and write privileges using the unique identifiers found in the header files. In this program, this refers to the Reis Robotstar V15's shared memory area (welding robot), the GM Fanuc S400's shared memory area(workpiece holder), and the MVS Line Laser Sensor shared memory area (sensor input).

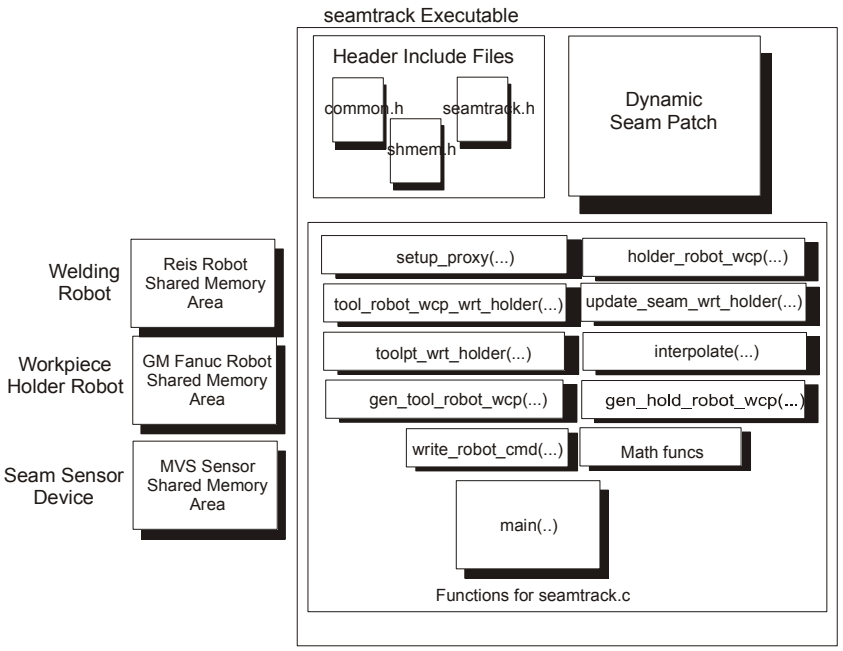


Figure 5.9. Anatomy of the "Seamtrack" Task program

Once synchronization and shared memory areas are established, we enter the main execution loop. At the beginning of this loop, the processor is put to sleep waiting for the "proxy" to be triggered from the " heartbeat". After being triggered by the "proxy" the GMF1 shared memory area is read using the holder_robot_wcp() function. This function waits for the shared memory semaphore to become available before downloading the database into its corresponding transformation matrix. The Reis shared memory area is read in the same manner using the tool_robot_wcp_wrt_holder() function, however, this particular function also uses kinematic relationships (Equation 5.10.) to determine the Reis robot's WCP with respect to the GMF1's WCP.

$$ {}^{(GMF1)6}T_{(Reis)6} = ({}^{global}T_{(Reis)\,0l}\,{}^{(Reis)0}T_{6})^{-1}\,{}^{global}T_{(GMF1)0}\,{}^{(GMF1)0}T_{6} \qquad (5.10.) $$

Next, the tool point of the Reis robot is determined with respect to the GMF1's WCP using the toolpt_wrt_holder() function using Equation 5.5.

New seam data is read from the MVS shared memory area and incorporated into the seam patch using the update_seam_wrt_holder() function. This function transforms the MVS seam data with respect to the GMF1's WCP using Equation 5.3. It then determines the yaw and pitch angles using Equations 5.6. and 5.8. The yaw and pitch rotation angles are then multiplied by a yaw gain and a pitch gain respectively prior to being applied to the transformation matrix so as to maintain the seam feature within the camera during seam tracking. The seam patch is then rotated and the new seam transformation matrix is placed at the front of the queue.

The target position and pose within the seam patch of the tool point is calculated using the interpolate() function. This function searches the queue looking for the two bounding points which the target tool point will lie between. It then linearly interpolates to satisfy the seam travel velocity. The orientation is taken from the seam transformation matrix which lies at the front of the seam patch queue.

Using the kinematic relationship of Equation 5.11 , the WCP with respect to the Reis robot's base is calculated using gen_tool_robot_wcp(). As well, the gen_hold_robot_wcp() function determines the new position and pose of the workpiece holder robot's WCP with respect to the GMF1 robot's base. Finally, these two robot commands are written to their respective shared memory areas using the write_robot_cmd() function.

$$ {}^{(Reis)0}T_6 = {}^{global}T_{(GMF1)0}\ {}^{(GMF1)0}T_6\ {}^{(GMF1)6}T_{tool}\ ({}^{(Reis)6}T_{tool})^{-1} \qquad (5.11.) $$

Figure 5.10. illustrates the execution flow of the Seam Tracking Task program. The source code for select functions is included at the end of this chapter.

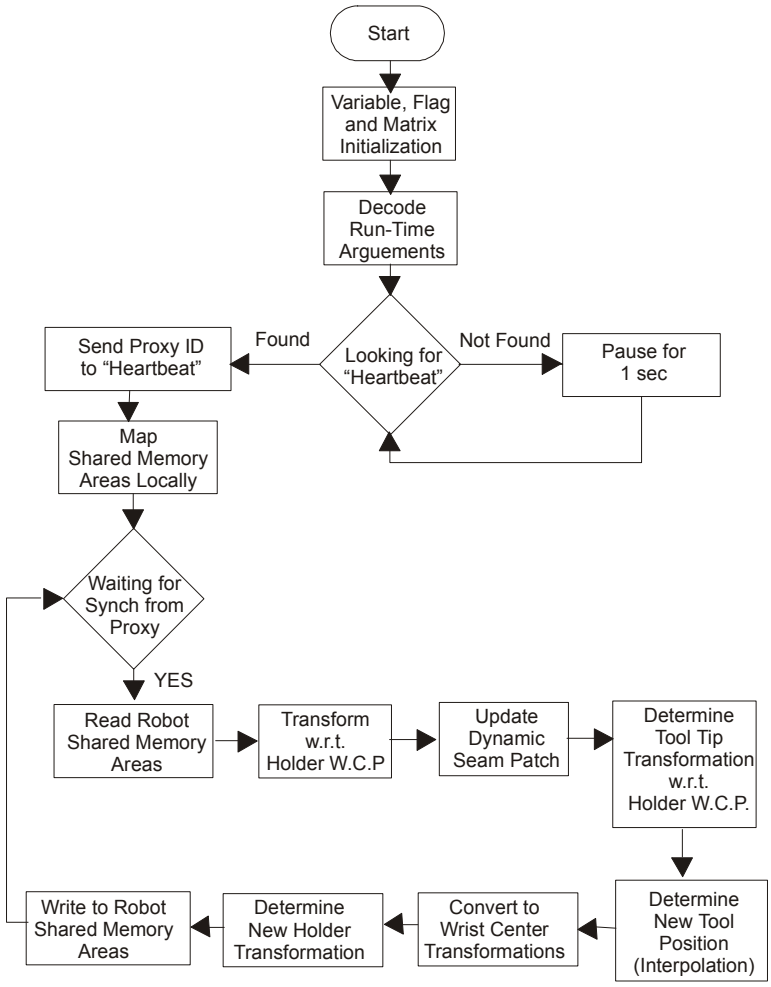


Figure 5.10. Flowchart of Seam Tracking software operation

```
int tool_robot_wcp_wrt_holder(shmem_cmd_t *cmd_ptr, float
                   *T06hold_ptr, *Ttool_hold_ptr,
                    *T06tool_ptr,*T6tool_hold_ptr)
{
   float T06hold_inv[4][4], Ttool_hold_inv[4][4], temp[4][4],
*T06_ptr;
 int i,j, error;

 T06_ptr = T06tool_ptr;

/* Command Information */
  sem_wait(&cmd_ptr->semaphore);
  for (i=0;i<3;i++)
    for(j=0;j<4;j++)
    {
      if (j == 3)
```

```
      {
        if (i==0) *T06_ptr = cmd_ptr->x;
        else if (i==1) *T06_ptr = cmd_ptr->y;
        else if (i==2) *T06_ptr = cmd_ptr->z;
      }
      else
        *T06_ptr = cmd_ptr->R[i][j];
      T06_ptr++;
    }
  sem_post(&cmd_ptr->semaphore);

/* Generating Tool WCP wrt to Holder WCP */
  error = mat_inv_homo(T06hold_ptr, &T06hold_inv[0][0]);
  error = mat_inv_homo(Ttool_hold_ptr,
                       &Ttool_hold_inv[0][0]);
   error = mult44(&T06hold_inv[0][0], &Ttool_hold_inv[0][0],
                &temp[0][0]);
  error = mult44(&temp[0][0], T06tool_ptr, T6tool_hold_ptr);

 return 0;
}

int update_seam_wrt_holder(float *T6tool_hold_ptr,
                           *Tcamera_6_ptr, int *tcp_idx)
{
  static int dead_zone = 30, first = TRUE;
  static float temp[4][4], temp2[4][4];
  static float Tyaw[4][4]; Tpitch[4][4];
  int i,j,k,error;
  float yaw=0, pitch=0;

/* Initialize the Seam Information */
  if (first)
  {
    first = FALSE;
    for (i=0; i<3; i++)
      for (j=0; j<4; j++)
        seam[0][i][j] = Treis7_gmf1[i][j];
  }

/* Reading the MVS Data for Roll and Translation Information
 */
  if (dead_zone > 0)
      dead_zone--;
  else
  {
    dead_zone = 0;
    sem_wait(&mvs_stat_ptr->semaphore);
    Tmvs[0][3] = mvs_stat_ptr->x;
    Tmvs[1][3] = mvs_stat_ptr->y;
    Tmvs[2][3] = mvs_stat_ptr->z;
    for(i=0;i<3;i++)
      for(j=0;j<3;j++)
        Tmvs[i][j] = mvs_stat_ptr->R[i][j];
    sem_post(&mvs_stat_ptr->semaphore);
  }
```

```
/* Determine the Pitch and Yaw from Previous Seam
   Transformation */
  yaw   = -(Tmvs[1][3]/CAMERA_TO_TCP)*kpyaw;
  pitch =  (Tmvs[0][3]/CAMERA_TO_TCP)*kppit;

  Tyaw[1][1] = cos(yaw); Tyaw[1][2] = -sin(yaw);
  Tyaw[2][1] = sin(yaw); Tyaw[2][2] =  cos(yaw);

  Tpitch[0][0] = cos(pitch);  Tpitch[0][2] = sin(pitch);
  Tpitch[2][0] = -sin(pitch); Tpitch[2][2] = cos(pitch);

  error = mult44(T6tool_hold_ptr, Tcamera_6_ptr,
                 &temp[0][0]);
  error = mult44(&temp[0][0], &Tmvs[0][0], &temp2[0][0]);
  error = mult44(&temp2[0][0], &Tpitch[0][0], &temp[0][0]);
  error = mult44(&temp[0][0], &Tyaw[0][0], &temp2[0][0]);

/* Rotating the Seam Queue */
  for (i=*tcp_idx; i >= 0;i--)
    for (j=0; j < 3; j++)
      for (k=0; k < 4; k++)
        seam[i+1][j][k] = seam[i][j][k];
  *tcp_idx = *tcp_idx + 1;
/* Update with the latest Data */
  for (i=0; i<3; i++)
    for (j=0; j<4; j++)
      seam[0][i][j] = temp2[i][j];

  return 0;
}

int toolpt_wrt_holder(float *T6tool_hold_ptr,
                      *T67_tool_ptr, *Ttool7_hold_ptr)
{
  int error;
  error = mult44(T6tool_hold_ptr, T67_tool_ptr,
                  Ttool7_hold_ptr);
 return 0;
}

int interpolate (float vel, int *tcp_idx_ptr, float
                 *Ttool7_hold_ptr, *Ttool7_hold_new_ptr)
{
  int i,j, found;
  float prev_dx, prev_dy, prev_dz, prev_dist;
  float next_dx, next_dy, next_dz, next_dist;
  float btwn_dx, btwn_dy, btwn_dz, btwn_dist;
  float Ttool7_hold[4][4], *ptr;

  ptr = Ttool7_hold_ptr;
  for (i=0;i<4;i++)
    for(j=0;j<4;j++)
    {
      Ttool7_hold[i][j] = *ptr;
      ptr++;
    }
```

```
/* Determine the Interpolated value using the further of the
   two points */
  if (*tcp_idx_ptr > MAX_QUEUE) *tcp_idx_ptr = MAX_QUEUE;
  else if (*tcp_idx_ptr < 1) *tcp_idx_ptr = 1;

/* Find the Index which is closest to the TCP */
  prev_dx = seam[*tcp_idx_ptr][0][3] - Ttool7_hold[0][3];
  prev_dy = seam[*tcp_idx_ptr][1][3] - Ttool7_hold[1][3];
  prev_dz = seam[*tcp_idx_ptr][2][3] - Ttool7_hold[2][3];
  prev_dist = sqrt((prev_dx*prev_dx + prev_dy*prev_dy +
                    prev_dz*prev_dz));

  next_dx = seam[*tcp_idx_ptr-1][0][3] - Ttool7_hold[0][3];
  next_dy = seam[*tcp_idx_ptr-1][1][3] - Ttool7_hold[1][3];
  next_dz = seam[*tcp_idx_ptr-1][2][3] - Ttool7_hold[2][3];
  next_dist = sqrt((next_dx*next_dx + next_dy*next_dy +
                      next_dz*next_dz));

  btwn_dx = seam[*tcp_idx_ptr-1][0][3] -
            seam[*tcp_idx_ptr][0][3];
  btwn_dy = seam[*tcp_idx_ptr-1][1][3] -
            seam[*tcp_idx_ptr][1][3];
  btwn_dz = seam[*tcp_idx_ptr-1][2][3] -
            seam[*tcp_idx_ptr][2][3];
  btwn_dist = sqrt((btwn_dx*btwn_dx + btwn_dy*btwn_dy +
                    btwn_dz*btwn_dz));
  found = FALSE;
  while ((!found) && (*tcp_idx_ptr > 1))
  {
    if((prev_dist < btwn_dist) && (next_dist < btwn_dist))
      found = TRUE;
    else if (next_dist < vel)
    {
      found = TRUE;
      *tcp_idx_ptr = *tcp_idx_ptr -1;
    }
    else
      *tcp_idx_ptr = *tcp_idx_ptr - 1;
    prev_dx = seam[*tcp_idx_ptr][0][3] - Ttool7_hold[0][3];
    prev_dy = seam[*tcp_idx_ptr][1][3] - Ttool7_hold[1][3];
    prev_dz = seam[*tcp_idx_ptr][2][3] - Ttool7_hold[2][3];
   prev_dist = sqrt((prev_dx*prev_dx + prev_dy*prev_dy +
                     prev_dz*prev_dz));

    next_dx = seam[*tcp_idx_ptr-1][0][3] -
                    Ttool7_hold[0][3];
    next_dy = seam[*tcp_idx_ptr-1][1][3] -
                    Ttool7_hold[1][3];
    next_dz = seam[*tcp_idx_ptr-1][2][3] -
                    Ttool7_hold[2][3];
    next_dist = sqrt((next_dx*next_dx + next_dy*next_dy +
                      next_dz*next_dz));

    btwn_dx = seam[*tcp_idx_ptr-1][0][3] -
              seam[*tcp_idx_ptr][0][3];
    btwn_dy = seam[*tcp_idx_ptr-1][1][3] -
              seam[*tcp_idx_ptr][1][3];
```

```
      btwn_dz = seam[*tcp_idx_ptr-1][2][3] -
                seam[*tcp_idx_ptr][2][3];
     btwn_dist = sqrt((btwn_dx*btwn_dx + btwn_dy*btwn_dy +
                      btwn_dz*btwn_dz));
    }

  /* Determine the Interpolated value using the further of the
     two points */
    if (*tcp_idx_ptr > MAX_QUEUE)
      *tcp_idx_ptr = MAX_QUEUE - 1;
    else if (*tcp_idx_ptr < 1)
      *tcp_idx_ptr = 1;

    ptr = Ttool7_hold_new_ptr;
    for (i=0;i<3;i++)
      for(j=0;j<4;j++)
      {
        if (j == 3)
        {
          if (i==0)
            *ptr = Ttool7_hold[0][3] + vel/next_dist*next_dx;
          else if (i==1)
            *ptr = Ttool7_hold[1][3] + vel/next_dist*next_dy;
          else if (i==2)
            *ptr = Ttool7_hold[2][3] + vel/next_dist*next_dz;   }
        else
          *ptr = seam[0][i][j];
        ptr++;
      }
    return 0;
  }

  int gen_tool_robot_wcp(float *T0tool_0hold_ptr, *T06hold_ptr,
                         *Ttool7_hold_new_ptr,*T67tool_ptr, *tool_cmd_ptr)
  {
    static float temp[4][4], T06[4][4], T07[4][4];
    int error;

  /* Calculate the Tool Point Transformation WRT to the World
   */
    error = mult44(T0tool_0hold_ptr, T06hold_ptr,
                   &temp[0][0]);
    error = mult44(&temp[0][0], Ttool7_hold_new_ptr,
                   &T07[0][0]);
    error = mat_inv_homo(T67tool_ptr, &temp[0][0]);
    error = mult44(&T07[0][0], &temp[0][0], &T06[0][0]);

  /* Calculate the Tool Point Transformation WRT to the World
   */
    error = mult44(&T06[0][0], T67tool_ptr, &T07[0][0]);
    error = mat_inv_homo(T67tool_ptr, &temp[0][0]);
  /* Calculate the New Robot WCP Transformation */
    error = mult44(&T07[0][0], &temp[0][0], tool_cmd_ptr);

    return 0;
  }
```

Chapter 6.

Experimentation and Performance

This chapter focuses on exploring the performance envelope of the seam-tracking task program as well as identifying key parameters which affect its overall performance. The chapter begins by discussing the objectives of the seam-tracking task program. The methodology of the data collection will be discussed. As well, the rationale for the test conditions of each trial collected will be explained. The chapter concludes with a presentation of the data collected as well as observations concerning the performance of the seam-tracking program with respect to key variables. Although a large number of experiments were conducted, only selected experiments are presented to highlight the key observations and conclusions for the sake of brevity.

6.1. Experimental Methodology

The objective in Dynamic Seam Tracking is to follow an arbitrary path in 3 dimensional space within a given accuracy. While the tool-point is traveling the path, it must maintain its feed-velocity as well as its relative pose to the workpiece.

The key indicator of performance for the seam-tracking system is the deviation from the seam at the tool-point. Using a secondary MVS laser profiling sensor positioned virtually at the Tool-Tip position, the following values are collected: i) the lateral deviation from the seam in millimeters ii) the height deviation from the seam in millimeters iii) the roll angle of the surface in radians. These 3 values indicate the error in the Seam Tracking system, because if no error were present, these 3 values would be zero. These error values are available to the DROID system at a rate of 60Hz which corresponds to twice the control rate of the Seam Tracking system. The collection of data occurs at 10Hz to facilitate analysis and charting. Since the data is made available to the DROID system via shared memory, data collection can be synchronized to other events within the system.

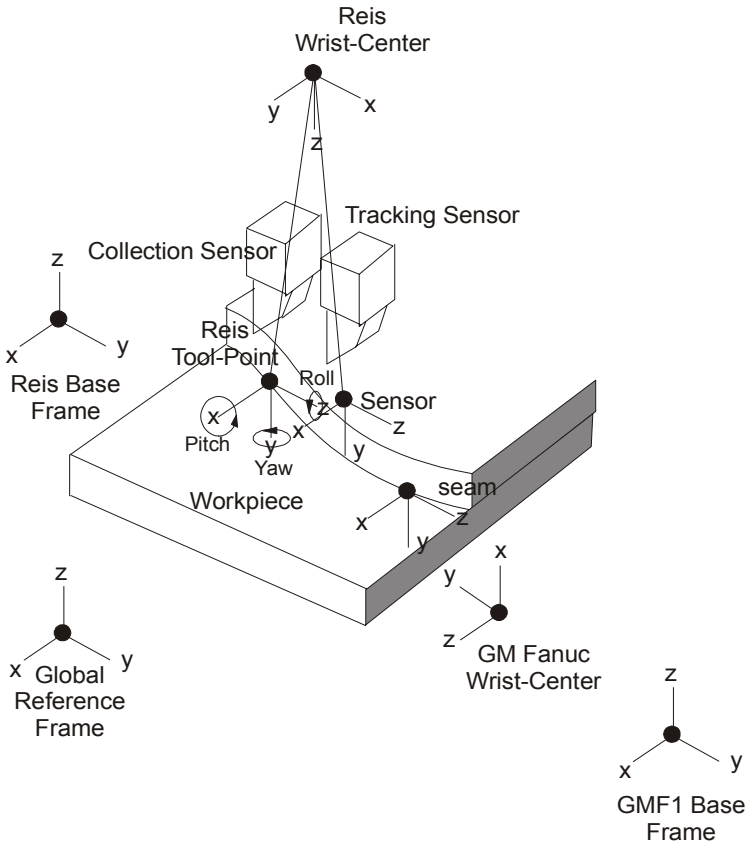


Figure 6.1. Seam reference frame

As each trial is executed and the data collected, a single system parameter will be varied to qualitatively illustrate the effects of this parameter on seam tracking performance. These key parameters include: i) the seam travel speed, ii) the angular orientation of the workpiece iii) the position of the workpiece, iv) the yaw angle gain, v) the pitch angle gain, vi) the sensory transport delay, vi) wrist setup configuration..

6.1.1. Experimental Workpieces

Due to the nature of the Seam Tracking program, an initial section is required to start building the seam segment between the sensor and the tool center point needed for tracking. From its predefined starting position, the robot is programmed to move the sensor in a straight line along the seam. Once this initial section has been traveled and the tool center point has reached the point on the seam at which the sensor started, any path can be tracked. In the first set of experiments, a step change in seam direction was used to investigate the response of the system. This step input is provided by a track with a 100mm initial straight section followed by an instantaneous change in direction of 5 degrees, the path remains straight for 600mm. This track is fixed to the GM Fanuc S-400 workpiece holder robot's tool plate.

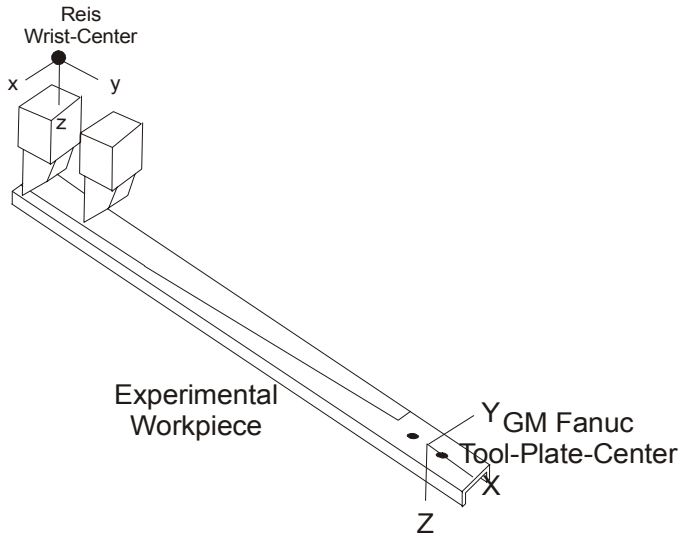


Figure 6.2. Step- Input Experimental Workpiece

The track itself is constructed using a 50mm plastic C- channel. Using a small width channel allows the track to be positioned deep inside the Reis Robotstar V15's work envelope without interfering with any of its links. The track is white in color and is a specular surface which provides excellent image data to the MVS sensors. The reason for using an ABS plastic channel is that its flexibility allows very aggressive tracking algorithms to be attempted without fear of colliding with a rigid workpiece. Figure 6.2 illustrates the Step Input Experimental Workpiece and its approximate orientation during the experiments.

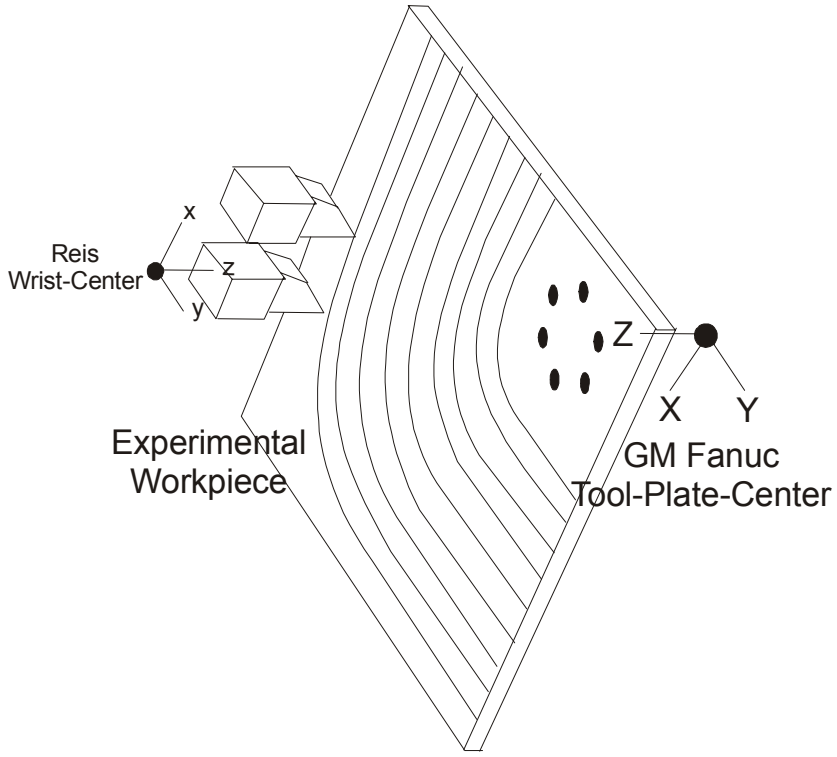


Figure 6.3. Ramp Input Experimental Workpiece

In addition, to the step input, a pseudo ramp input can be provided to the seam tracking system by having an initial straight section followed by a fixed radius arc. A second experimental workpiece was constructed using a flat 1000mmx1000mm sheet of PVC plastic. The surface of the PVC sheet is uniform and reflects the laser light used by the profiling sensor. Precision

mounting holes are located at one corner of the sheet to allow rigid placement of the workpiece to the robot tool plate. 5 tracks of 50mm are spaced 50mm apart. Each track comprises of a 300mm straight track, the track then turns 90 degrees with a fixed radius and concludes with a 300mm straight track. The radii vary from 200mm to 500mm with values at 200mm, 300mm, 400mm and 500mm. The tracks exhibit a uniform edge height of approximately 2mm. Figure 6.3. illustrates the Ramp Input Experimental Workpiece and its approximate orientation during the experiments.

6.1.2. Variable Descriptions and Ranges

The previous section illustrates that each track varies in terms of cartesian position, plotting the acquired sensor data with respect to each x,y ,z axis would not provide a very meaningful comparison. Therefore, data was plotted with respect to the path length traveled so that data from any path can be compared. This actual path length is the independent variable. Therefore, the seam travel speed is the linear distance traveled along the seam in millimeters per second. The speed was varied from 4mm/s to 64mm/s with the following travel speeds: 4mm/s, 8mm/s, 16mm/s, 32mm/s, and 64mm/s.

Due to the kinematics of the Reis Robotstar V15, the different positions/orientations place the robot in different joint configurations which may affect the ability of the robot to reach its target position/orientation in a seam-tracking cycle thereby affecting the accuracy/performance of the Seam Tracking system. Another situation that arises due to joint configurations is the crossing over from a workspace partition to another. The work space of the robot can be considered a sphere centered about joint 2. This sphere is partitioned into 8 using three orthogonal planes which intersect at the origin of the workspace sphere. Crossing these planes results in the robot changing its joint configuration instantaneously. Depending on the location of this cross-over, the results vary from slight jitter to momentary lose of camera data. To avoid this situation, we limit the Seam Tracking to the positive x, positive y, positive z spherical partition.

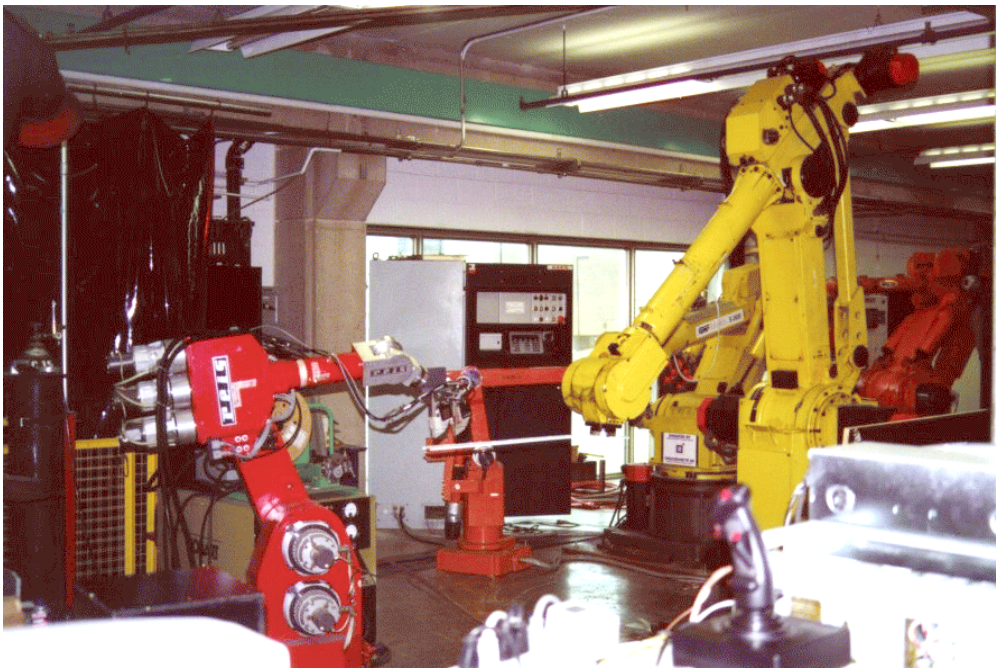


Figure 6.4. (i) Position within positive workspace and 0 degree Roll and Pitch

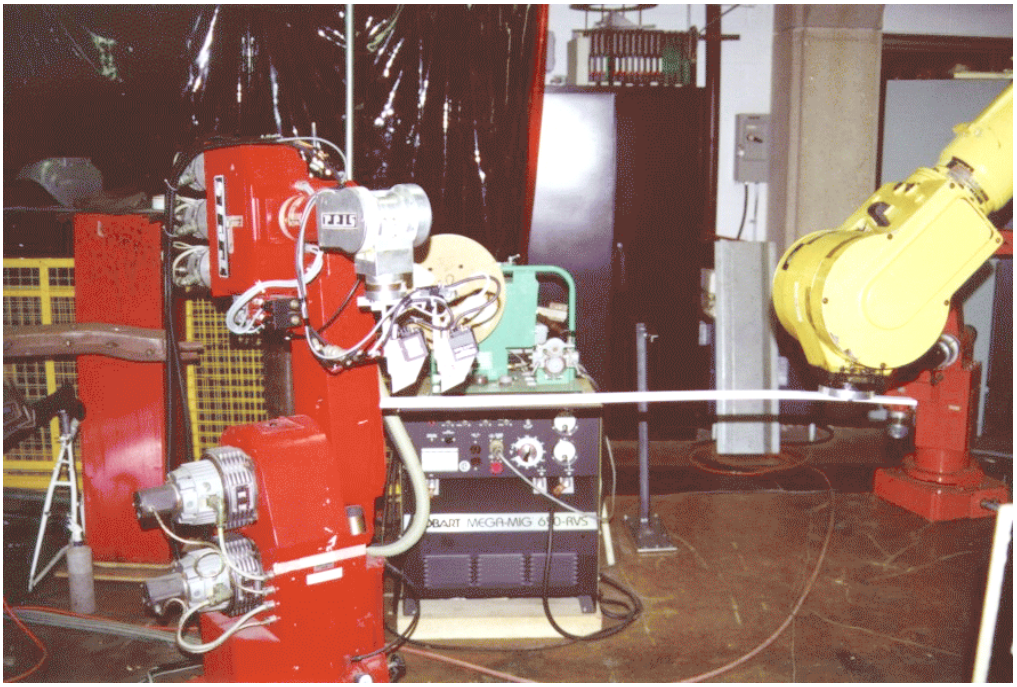


Figure 6.4. (ii) 300mm lateral offset of the track to position (i)

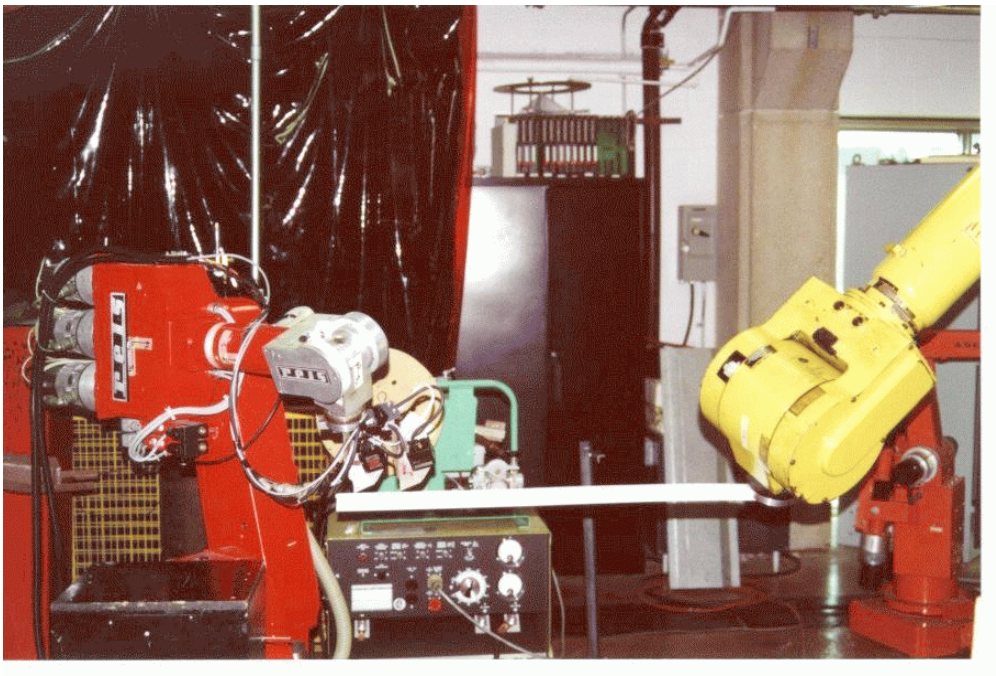


Figure 6.4.(iii) Same as position (i) but with a 30 degree Roll of the track

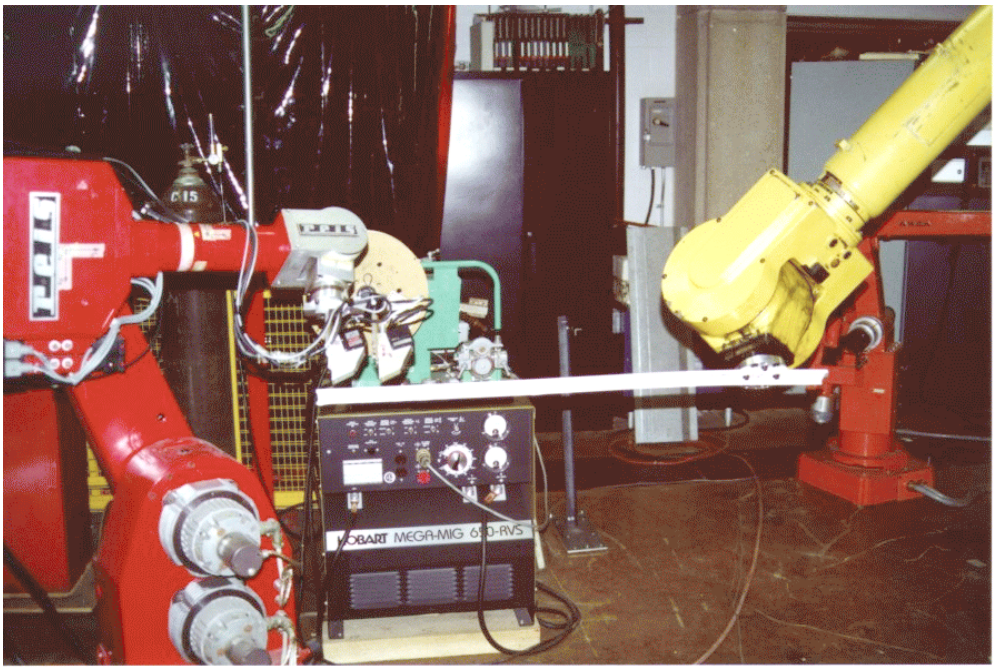


Figure 6.4. (iv) Same as position (i) but 45 degree Roll of the track

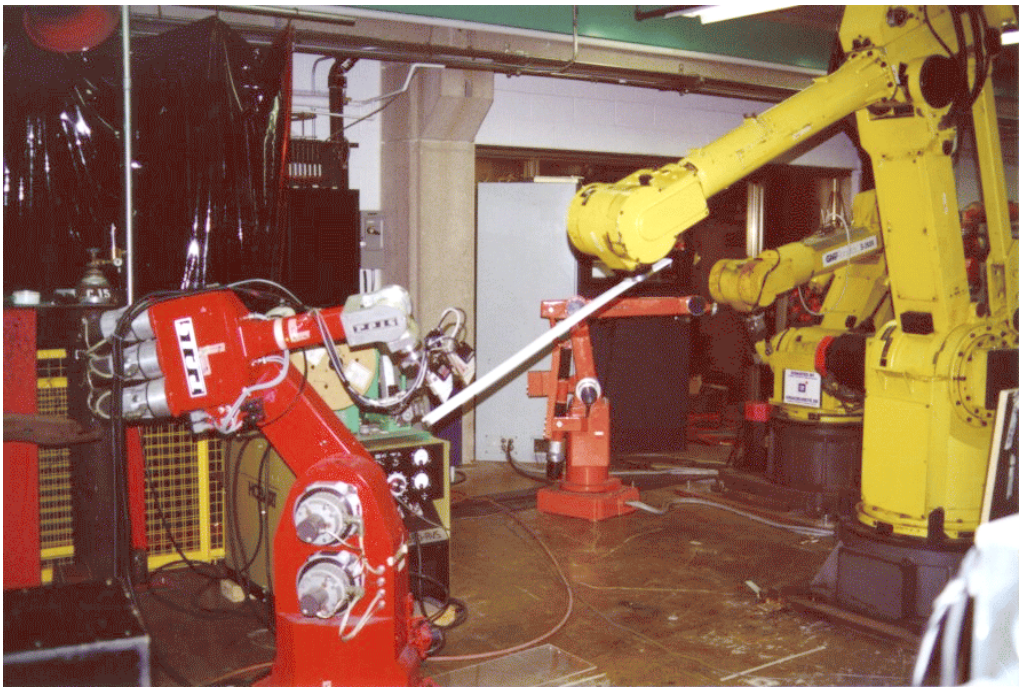


Figure 6.4. (v) Same as position (i) but 30 degree incline

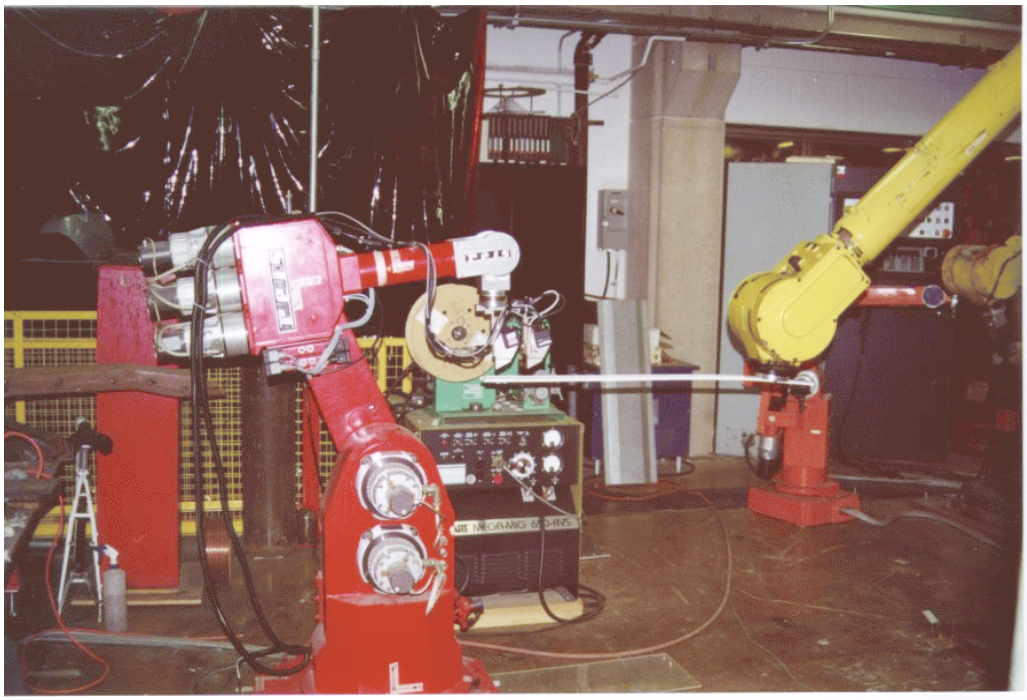


Figure 6.4. (vi) Position and orientation along radial

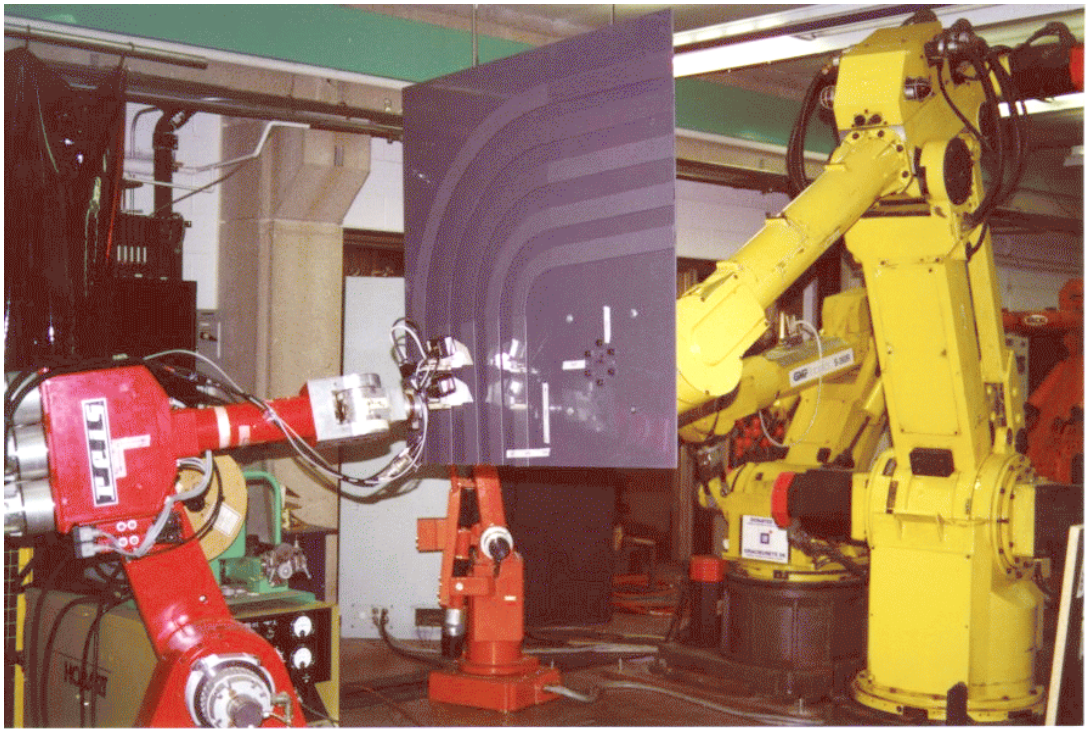


Figure 6.4. (vi) Same as position (i) but vertical.

Seven different starting positions and orientations were used to explore the seam tracking performance with respect to position and orientation. These joint configurations include i) position within positive workspace (x = 300mm, y = 100mm, z = 1340mm w.r.t. Reis Base) and 0 degree Roll and Pitch (i.e. the track is flat) ii) 300mm lateral offset of the track to position (i), iii) same as position (i) but with a 30 degree Roll of the track, iv) same as position (i) but 45 degree Roll of the track, v) same as position (i) but 30 degree incline vi) position and orientation along radial, vii) same as position (i) but vertical. Figures 6.4. (i) through (vii) illustrate the different joint configurations used to reach each start position/orientation.

In order to have the MVS Sensor positioned on the seam throughout seam-tracking, a proportional controller is implemented so as to rotate the camera about the tool-tip so that the seam feature remains within the camera view. These proportional gains are defined as the Yaw Angle Gain for the Yaw plane and the Pitch Angle Gain for the Pitch plane. The Yaw Angle

Gain was varied from 0.05 to 0.75 with values at 0.05, 0.15, 0.25, 0.35, 0.55, and 0.75. The Pitch Angle Gain was varied from 0.05 to 0.45 with values at 0.05, 0.15, 0.25, 0.35, and 0.45.

The DROID architecture is a distributed network of robots and sensors, therefore it is crucial that data transfer occur in predictable and timely fashion. To explore the possibility of adverse network connections, delays were inserted into the sensor stream. This experiment consisted of running the Seam Tracking program for 150mm (which is approximately when steady-state tracking is reached), then a single delay between 1 to 16 seam tracking cycles is inserted and the tracking monitored. The inserted delays are 1 cycle, 2 cycles, 4 cycles, 5cycles, 6cycles, 7cycles, 8 cycles and 16 cycles. Since the seam tracking frequency is 30Hz, these delays correspond to 33.3ms, 66.7ms, 133.3ms, 166.7 ms, 200.0 ms, 233.3ms, 266.7ms, and 533.3ms delays.

6.2. Experimental Results and Data Analysis

6.2.1. Experiment Set 1: Position, Orientation and Seam Travel Speed

The first experiments conducted were those which varied position, orientation and seam travel speed. The purpose of these experiments was to identify the optimal position of the workpiece, the optimal orientation and the maximum travel speed for this given seam tracking configuration. Certain key variables were needed to be fixed, although at this point we could not determine their optimal values. One of these non optimized variables is the workspace partition, which we set as the positive x, positive y positive z partition. Other non optimized variables include the proportional gains used for the control of the seam tracking. Different gain values can cause the Seam Tracking program to respond differently which is not what this initial experiment wishes to focus on, thus the gains were kept constant at values of 0.15 for the yaw gain and 0.25 for the pitch gain. Table 6.1. is an experimental matrix which shows the variable setting for each experiment. An experiment consists of a single Seam Tracking program execution on a given experimental workpiece, at a given position, at one of the 5 seam velocities

dictated in Section 6.1. For the step-input workpiece, all 5 seam travel speeds (4, 8, 16, 32, 64 mm/s) are run for 5 of the 7 position/orientation positions mentioned in section 6.1. For the Ramp Input Experimental Workpiece, 4 of the 5 seam travel speeds (4, 8, 16, 32mm/s) are run for the vertical position due to workspace interference.

Table 6.1. Initial Exploration: Travel Speed and Position/Orientation

ID	Track	Travel Speed	Orientation	Position
1.1	Straight-Line (step input)	4 mm/s	0º roll, 0º pitch	position (i)
1.2	Straight-Line (step input)	8 mm/s	0º roll, 0º pitch	position (i)
1.3	Straight-Line (step input)	16 mm/s	0º roll, 0º pitch	position (i)
1.4	Straight-Line (step input)	32 mm/s	0º roll, 0º pitch	position (i)
1.5	Straight-Line (step input)	64 mm/s	0º roll, 0º pitch	position (i)
1.6	Straight-Line (step input)	4 mm/s	0º roll, 0º pitch	position (i), +300mm lateral offset
1.7	Straight-Line (step input)	8 mm/s	0º roll, 0º pitch	position (i), +300mm lateral offset
1.8	Straight-Line (step input)	16 mm/s	0º roll, 0º pitch	position (i), +300mm lateral offset
1.9	Straight-Line (step input)	32 mm/s	0º roll, 0º pitch	position (i), +300mm lateral offset
1.10	Straight-Line (step input)	64 mm/s	0º roll, 0º pitch	position (i), +300mm lateral offset

ID	Track	Travel Speed	Orientation	Position
1.11	Straight-Line (step input)	4 mm/s	30º roll, 0º pitch	position (i)
1.12	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch	position (i)
1.13	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch	position (i)
1.14	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch	position (i)
1.15	Straight-Line (step input)	64 mm/s	30º roll, 0º pitch	position (i)
1.16	Straight-Line (step input)	4 mm/s	45º roll, 0º pitch	position (i)
1.17	Straight-Line (step input)	8 mm/s	45º roll, 0º pitch	position (i)
1.18	Straight-Line (step input)	16 mm/s	45º roll, 0º pitch	position (i)
1.19	Straight-Line (step input)	32 mm/s	45º roll, 0º pitch	position (i)
1.20	Straight-Line (step input)	64 mm/s	45º roll, 0º pitch	position (i)
1.21	Straight-Line (step input)	4 mm/s	0º roll, 30º pitch	position (i)
1.22	Straight-Line (step input)	8 mm/s	0º roll, 30º pitch	position (i)
1.23	Straight-Line (step input)	16 mm/s	0º roll, 30º pitch	position (i)
1.24	Straight-Line (step input)	32 mm/s	0º roll, 30º pitch	position (i)
1.25	Straight-Line (step input)	64 mm/s	0º roll, 30º pitch	position (i)
1.26	500 mm radial (ramp input)	4 mm/s	0º roll, 90º pitch	position (i)
1.27	500 mm radial (ramp input)	8 mm/s	0º roll, 90º pitch	position (i)
1.28	500 mm radial (ramp input)	16 mm/s	0º roll, 90º pitch	position (i)
1.29	500 mm radial (ramp input)	32 mm/s	0º roll, 90º pitch	position (i)
1.30	400 mm radial (ramp input)	4 mm/s	0º roll, 90º pitch	position (i)
1.31	400 mm radial (ramp input)	8 mm/s	0º roll, 90º pitch	position (i)
1.32	400 mm radial (ramp input)	16 mm/s	0º roll, 90º pitch	position (i)
1.33	400 mm radial (ramp input)	32 mm/s	0º roll, 90º pitch	position (i)
1.34	300 mm radial (ramp input)	4 mm/s	0º roll, 90º pitch	position (i)
1.35	300 mm radial (ramp input)	8 mm/s	0º roll, 90º pitch	position (i)
1.36	300 mm radial (ramp input)	16 mm/s	0º roll, 90º pitch	position (i)

ID	Track	Travel Speed	Orientation	Position
1.37	300 mm radial (ramp input)	32 mm/s	0º roll, 90º pitch	position (i)
1.38	200 mm radial (ramp input)	4 mm/s	0º roll, 90º pitch	position (i)
1.39	200 mm radial (ramp input)	8 mm/s	0º roll, 90º pitch	position (i)
1.40	200 mm radial (ramp input)	16 mm/s	0º roll, 90º pitch	position (i)
1.41	200 mm radial (ramp input)	32 mm/s	0º roll, 90º pitch	position (i)

6.2.2. Experiment 1: Observations

Figure 6.5. shows the data collected from a trial where the step input workpiece was rotated 30 degrees about the Roll axis and the trial conducted at 8mm/s. The response is a typical seam tracking response to a step input. The seam tracking begins with an initial error. Due to the Seam Tracking program's initial conditions routine, the error is only corrected after it has traveled for more than 100mm. An overshoot is seen in the Y axis as the seam-tracking begins and a slight steady-state error is observed. The X-axis appears to exhibit minimal overshoot with a slight steady-state error. The roll angle typically undergoes little change, hovering about zero slightly.

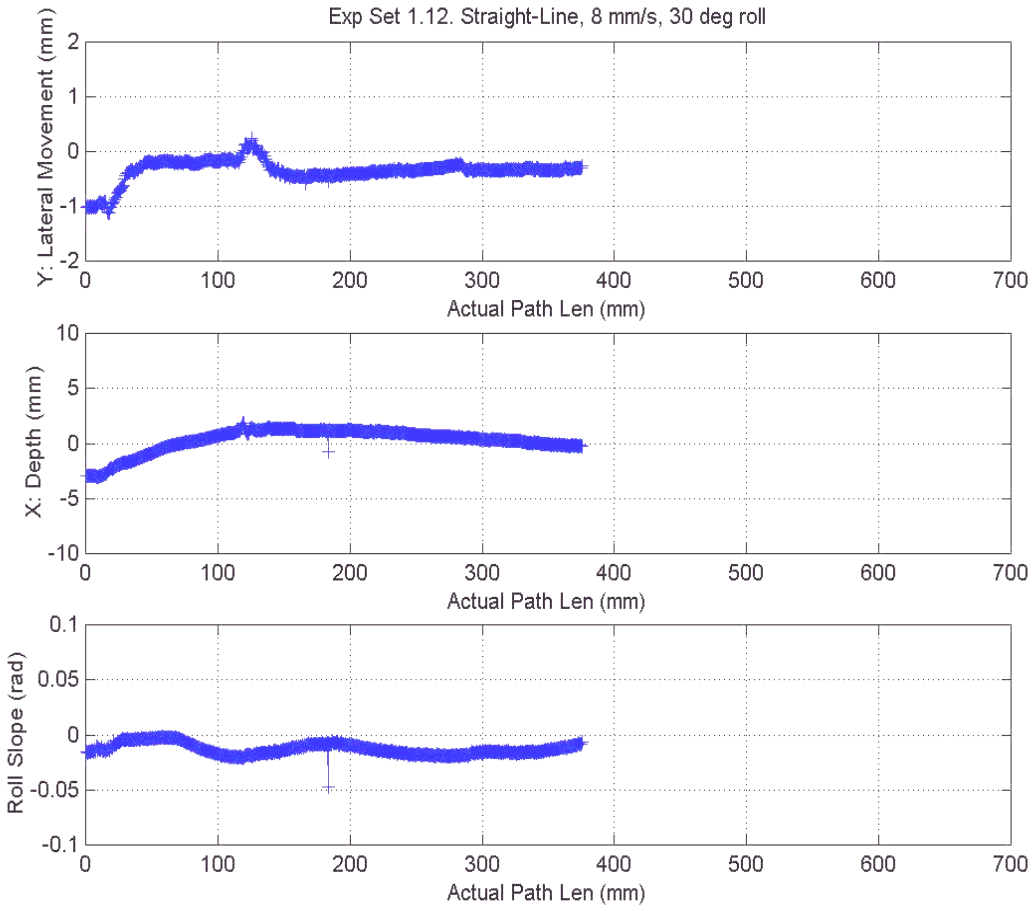


Figure 6.5. Step input response for all axis' with 30 degree roll at 8mm/s

In general, as speed increases, the overshoot on the X axis increases slightly. The Y axis overshoot is reduced and produces a smoother response initially. However, oscillations result late in seam tracking at 16 mm/s. At 32 mm/s the seam tracking trial is cut short as the lead sensor loses the seam feature. At 64mm/s the oscillations amplitudes are much larger. Figure 6.6. shows the step input response of a workpiece rolled at 30 degrees at 4mm/s, 8mm/s, 16mm/s, 32mm/s and 64 mm/s.

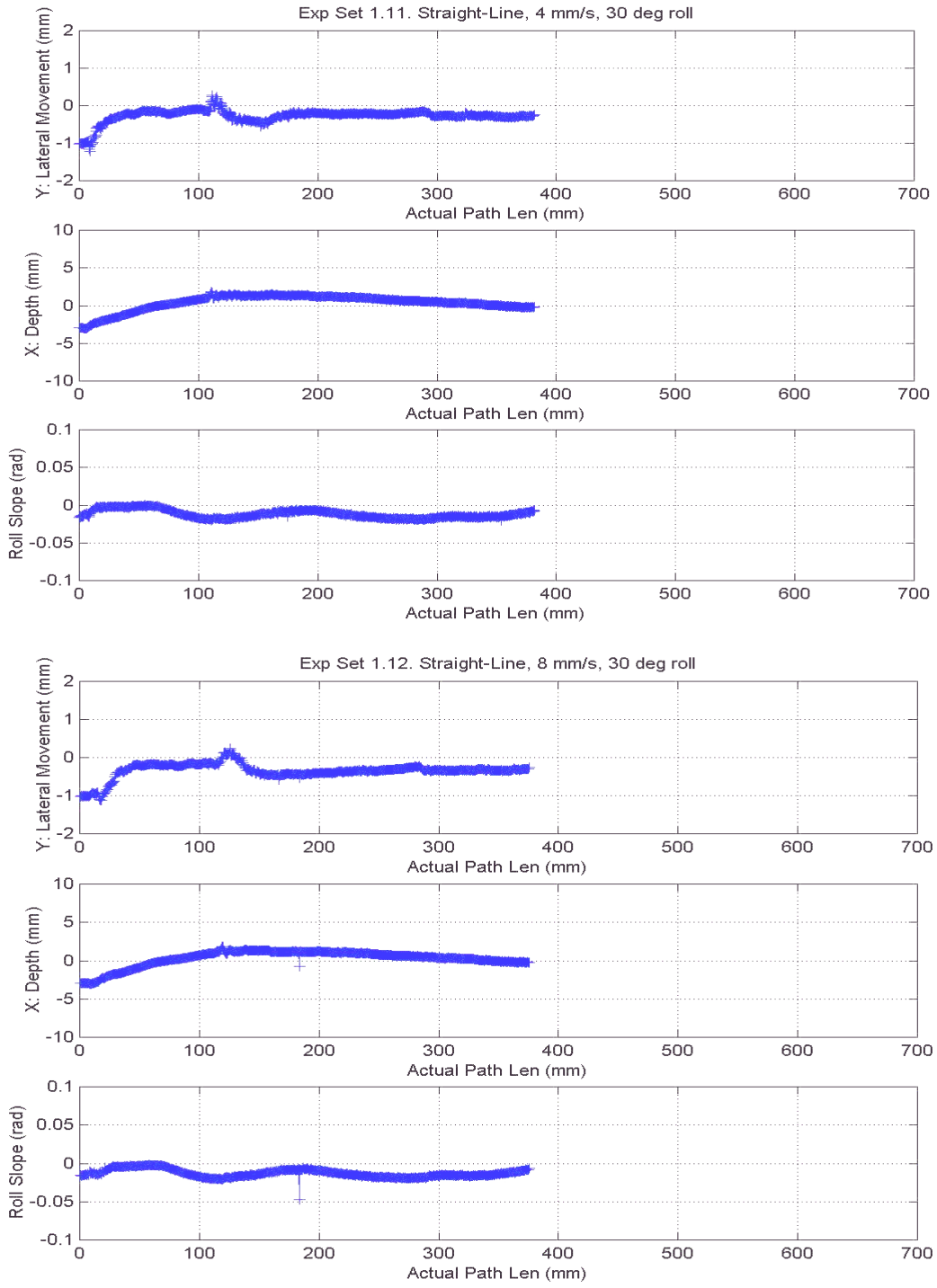


Figure 6.6. Step input response for all axis' at 30 degree Roll angle

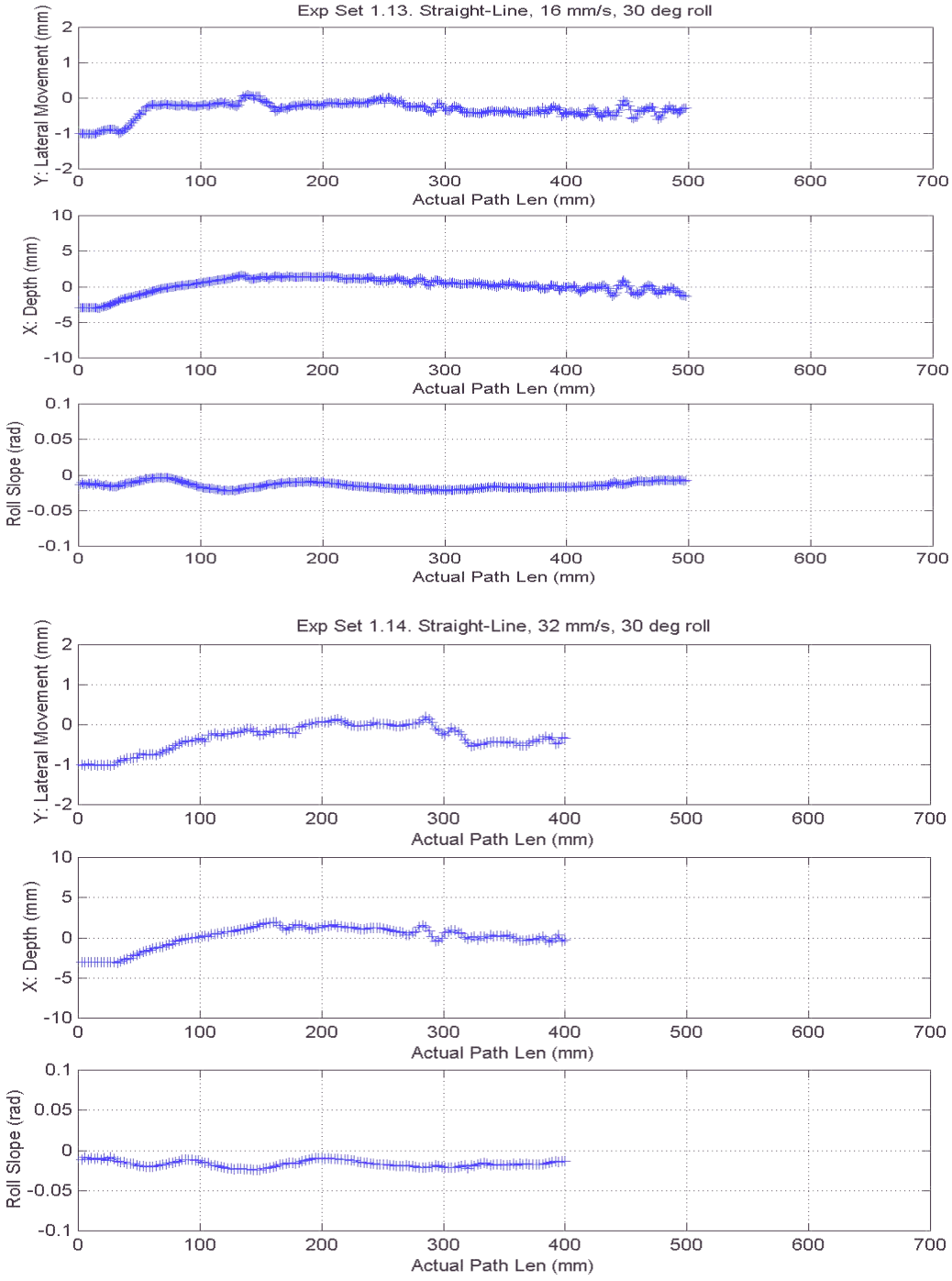


Figure 6.6. Step input response for all axis' at 30 Degree Roll Angle (cont'd)

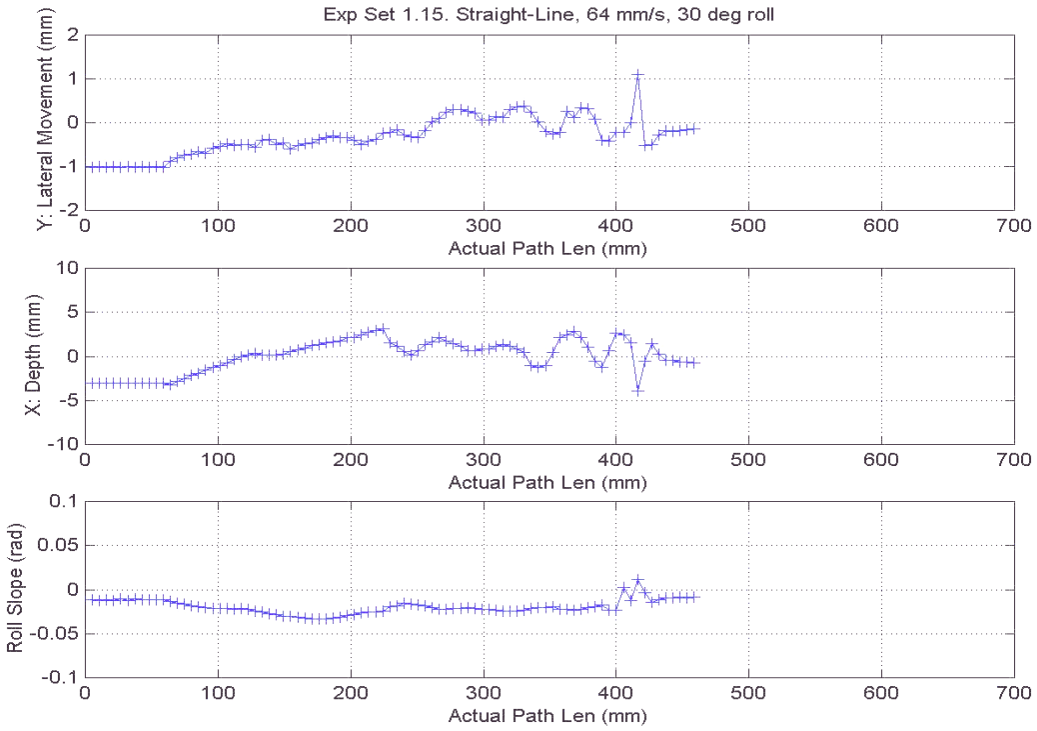


Figure 6.6. Step input response for all axis's at 30 degree Roll angle (cont'd)

From Figure 6.7. Experiment 1.7 shows that as the workpiece is placed farther away from the robot it causes stronger oscillations throughout the seam tracking trial. Rotating the workpiece about the roll axis causes the response to be smoother and reduces the oscillations. Experiment 1.12 shows that at 30 degree roll angle, the system is most stable with the given proportional gains. Experiment 1.17 shows that at 45 degree roll the response is also stable, but not as stable as the 30 degree case. Rotating the workpiece about its pitch angle such as in Experiment 1.22 causes an amplified overshoot at the 100mm mark as well as causing higher amplitude oscillations at the latter part of the seam.

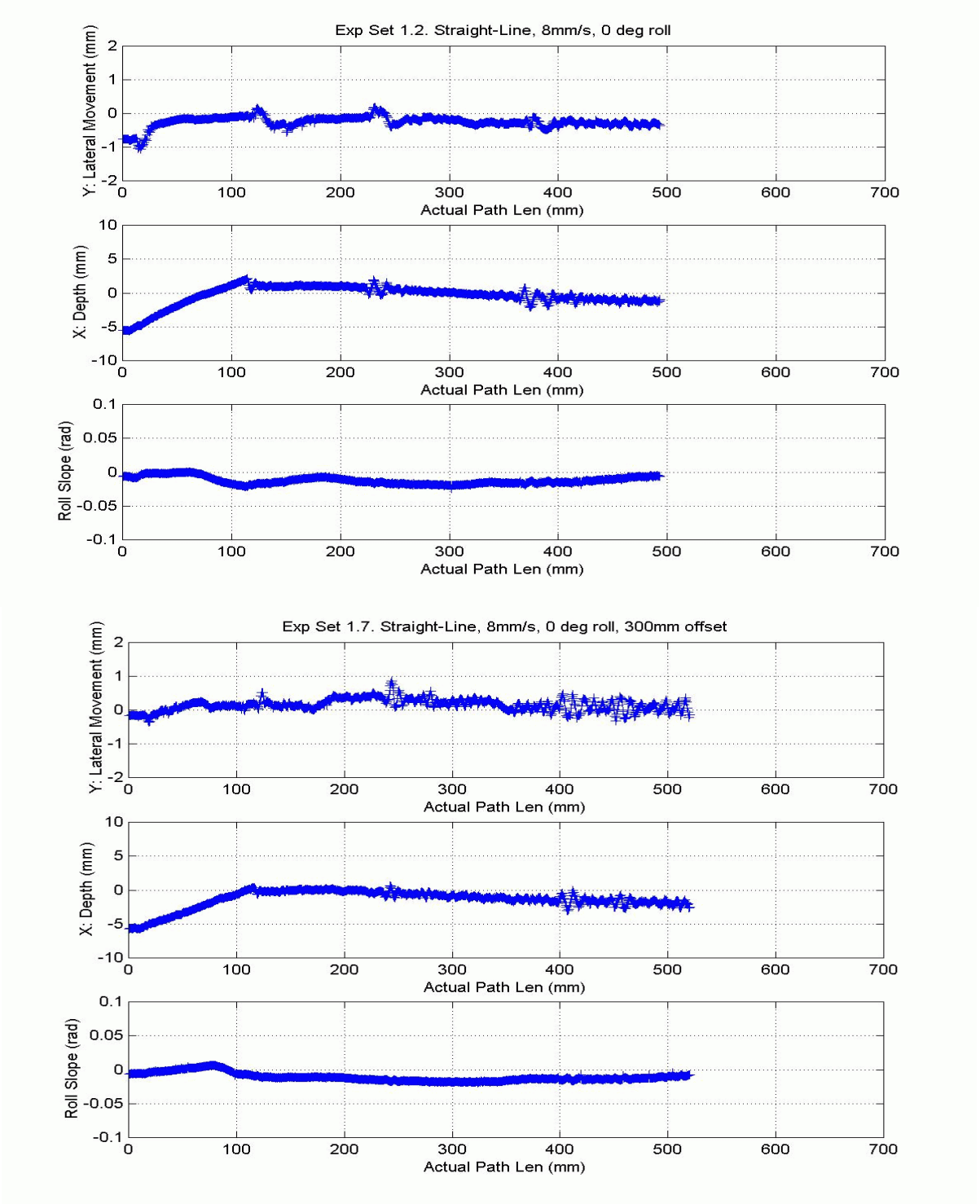
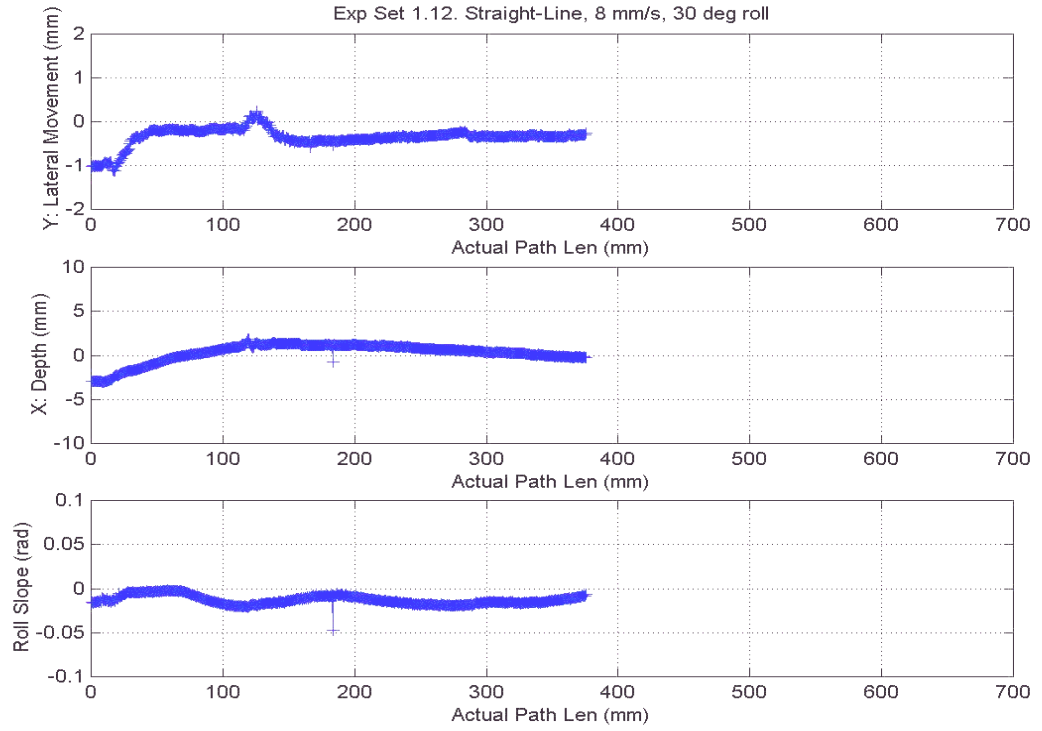


Figure 6.7. Step input response for all axis' with changing Position/Orientation


Exp Set 1.12. Straight-Line, 8 mm/s, 30 deg roll
Y: Lateral Movement (mm)
X: Depth (mm)
Roll Slope (rad)
Actual Path Len (mm)

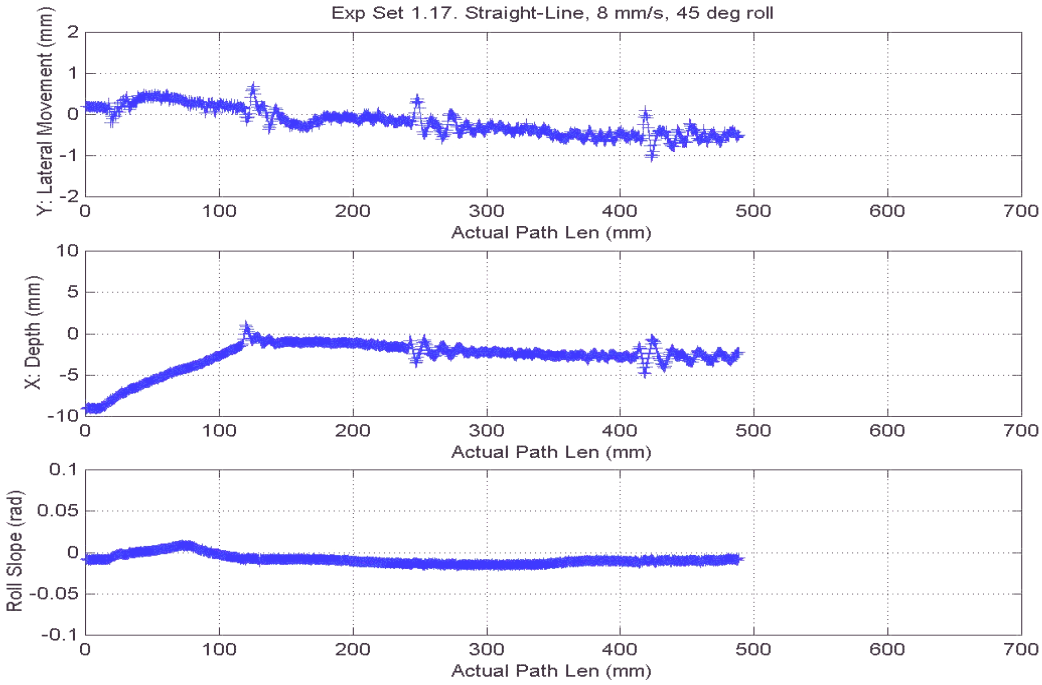

Exp Set 1.17. Straight-Line, 8 mm/s, 45 deg roll
Y: Lateral Movement (mm)
X: Depth (mm)
Roll Slope (rad)
Actual Path Len (mm)

Figure 6.7. Step input response for all axis' with changing Position/Orientation (cont'd)

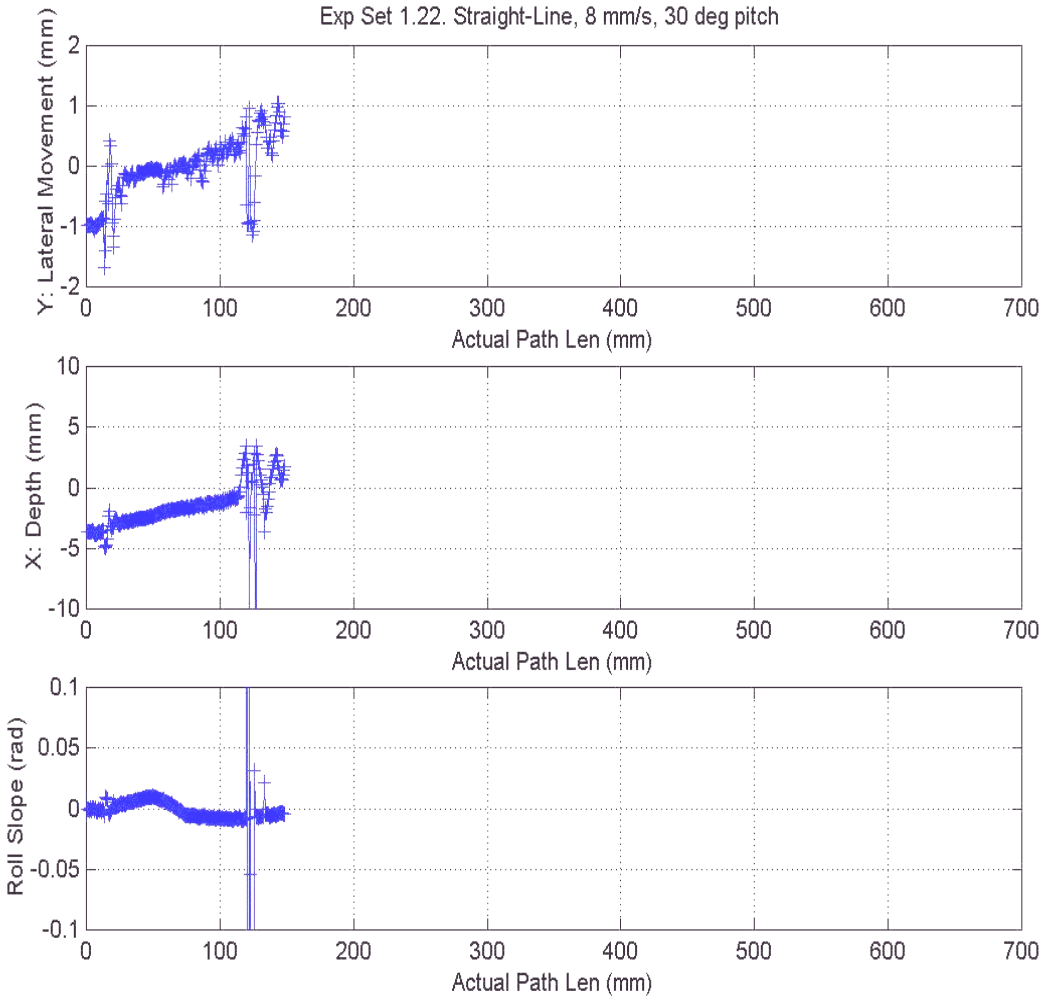


Figure 6.7. Step input response for all axis' with changing Position/Orientation (cont'd)

The data collected from the experiments conducted with ramp inputs show better tracking performance in the Y axis (lateral) compared X axis. The roll axis remains unaffected by the ramp input regardless of the speed or radius. Figure 6.8. shows a set of data collected for 500mm radius at 4 mm/s.

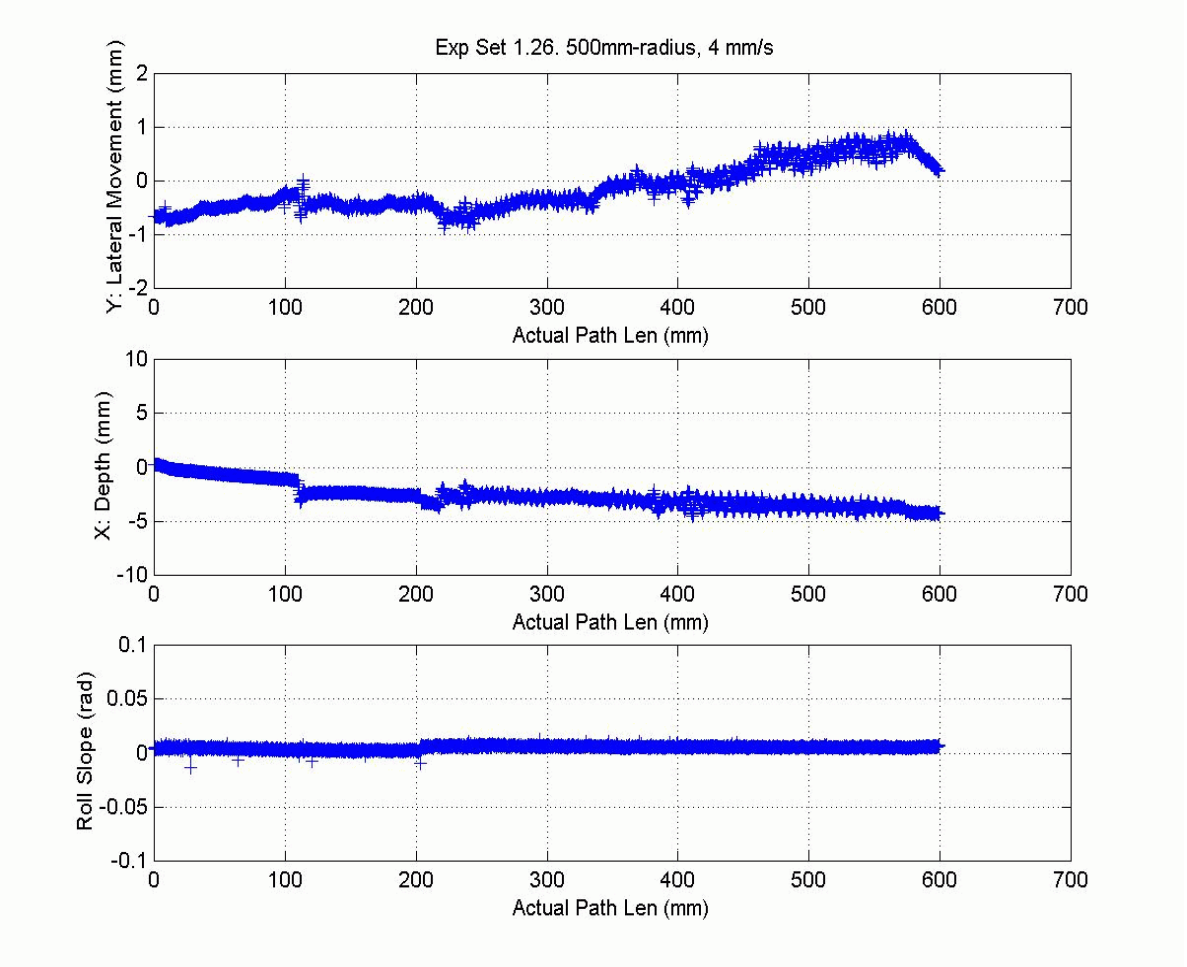
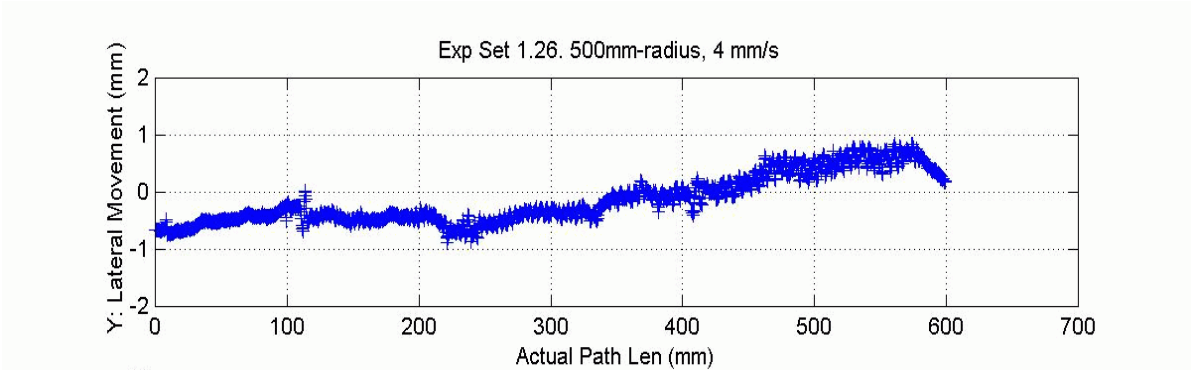
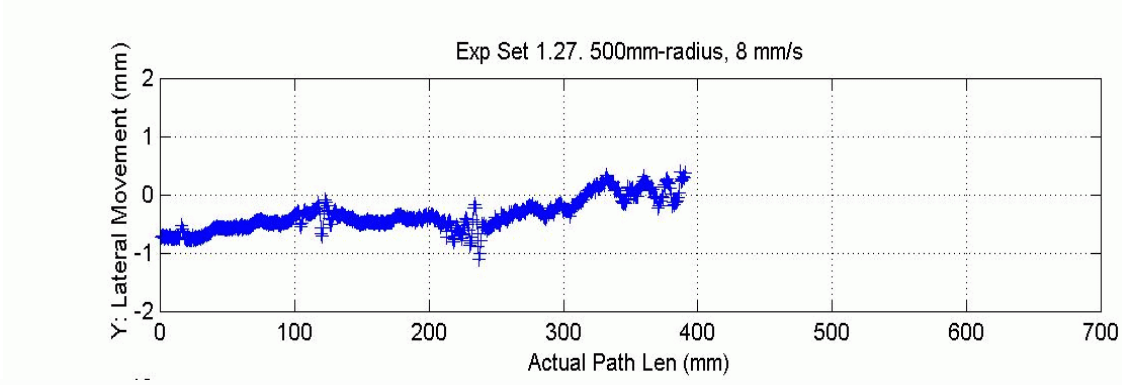
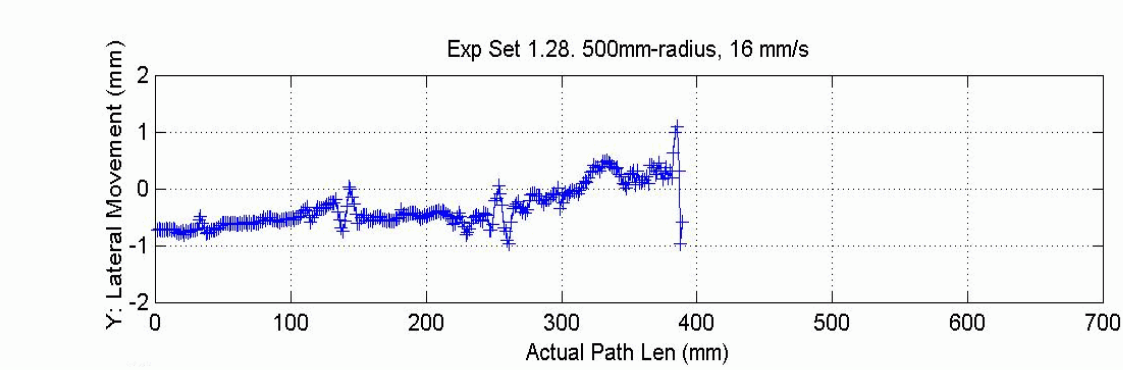


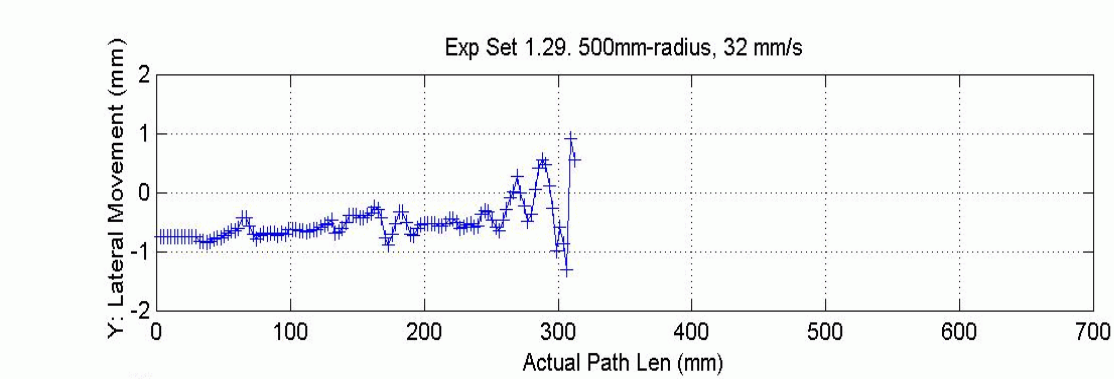
Figure 6.8: Typical Seam Tracking system response to ramp input

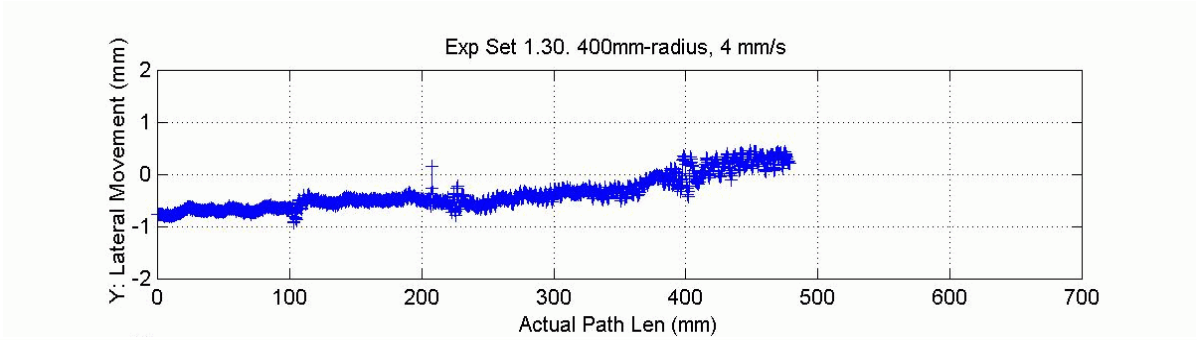
For the Y axis, the system tracks the seam, maintaining a slight offset in error until at approximately 300 mm, it attempts to compensate and overshoots the target. Figure 6.9 to 6.12 shows a series of Y axis plots as the seam travel speed increases and are grouped according to radius. From Figures 6.9 and 6.12. it can be seen that the trends remain the same as the speed increases As the radius decreases, which corresponds to a larger ramp slopes, the lateral error remains at approximately 1mm but compensation occurs late in the seam tracking trial indicating a slower response to the ramp input.

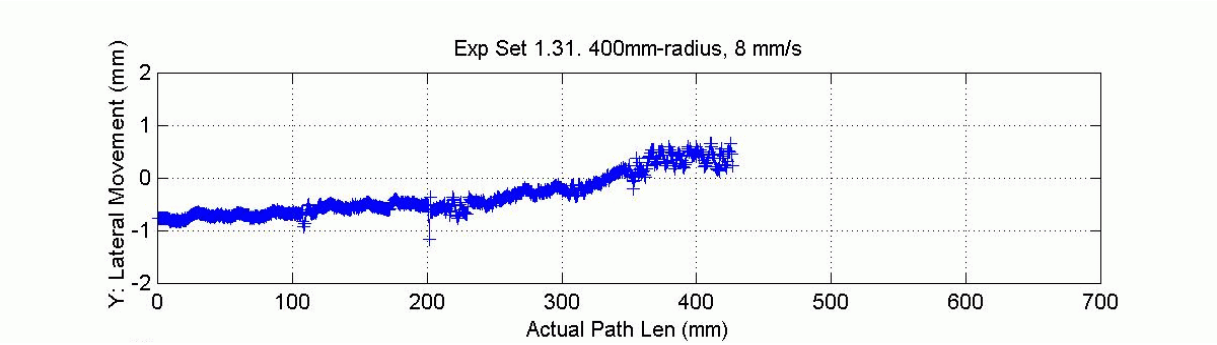

Exp Set 1.26. 500mm-radius, 4 mm/s
Y: Lateral Movement (mm)
Actual Path Len (mm)

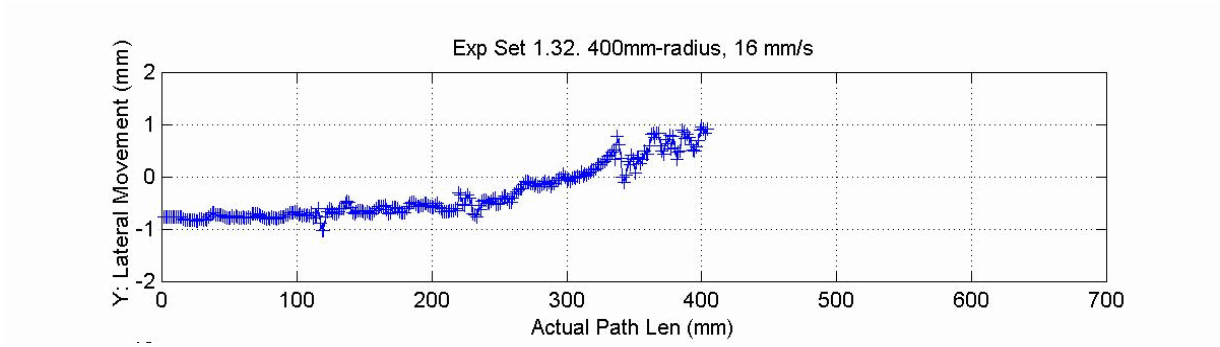

Exp Set 1.27. 500mm-radius, 8 mm/s
Y: Lateral Movement (mm)
Actual Path Len (mm)


Exp Set 1.28. 500mm-radius, 16 mm/s
Y: Lateral Movement (mm)
Actual Path Len (mm)


Exp Set 1.29. 500mm-radius, 32 mm/s
Y: Lateral Movement (mm)
Actual Path Len (mm)







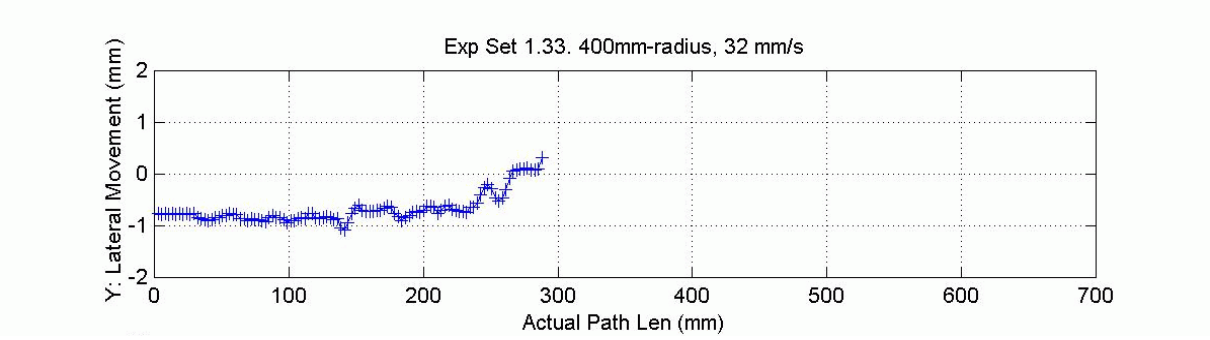


Figure 6.9: Lateral Seam Tracking system response to Ramp Input Radius of 500mm

Figure 6.10: Lateral Seam Tracking system response to Ramp Input Radius of 400mm

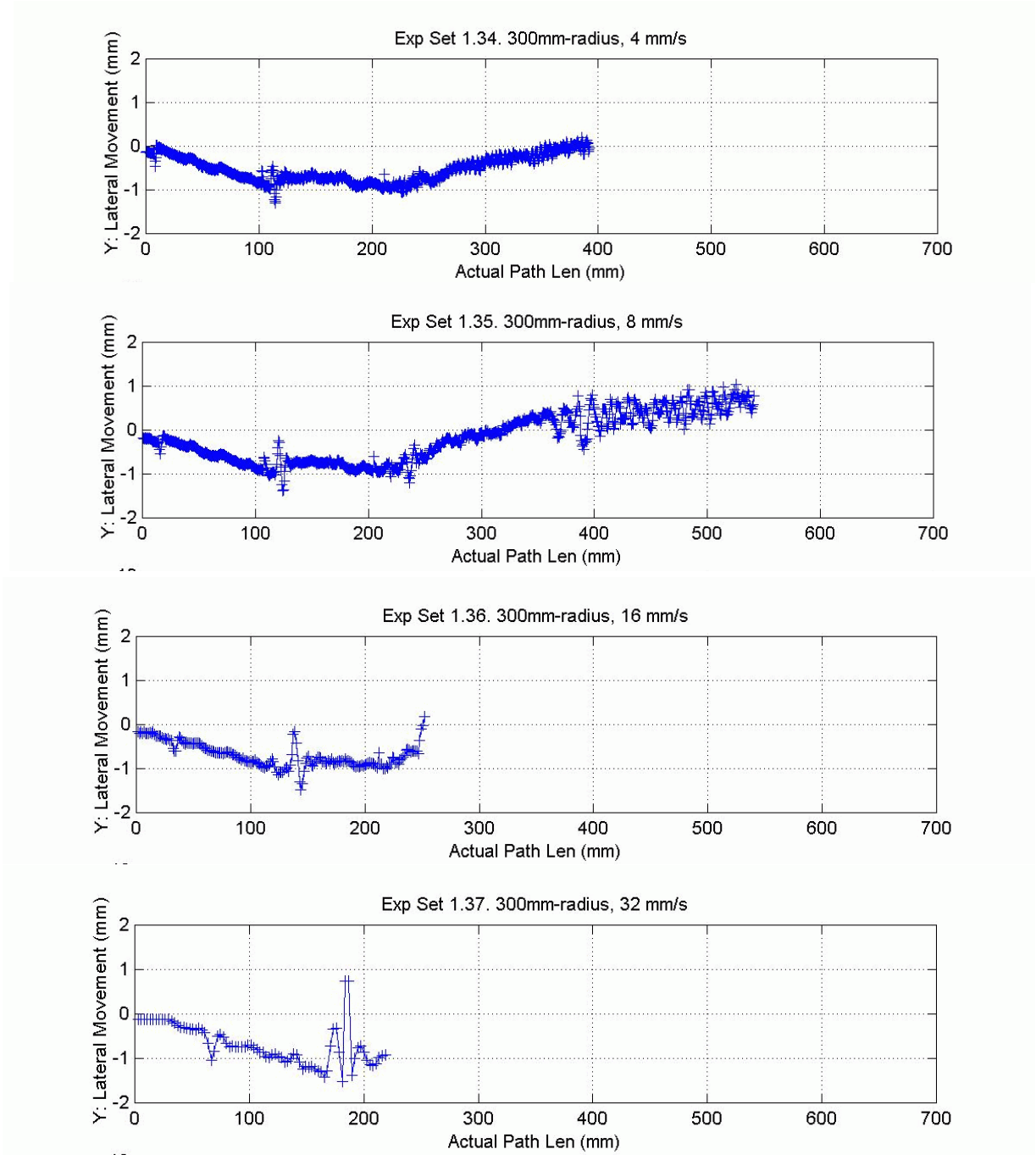


Figure 6.11: Lateral Seam Tracking system response to Ramp Input Radius of 300mm

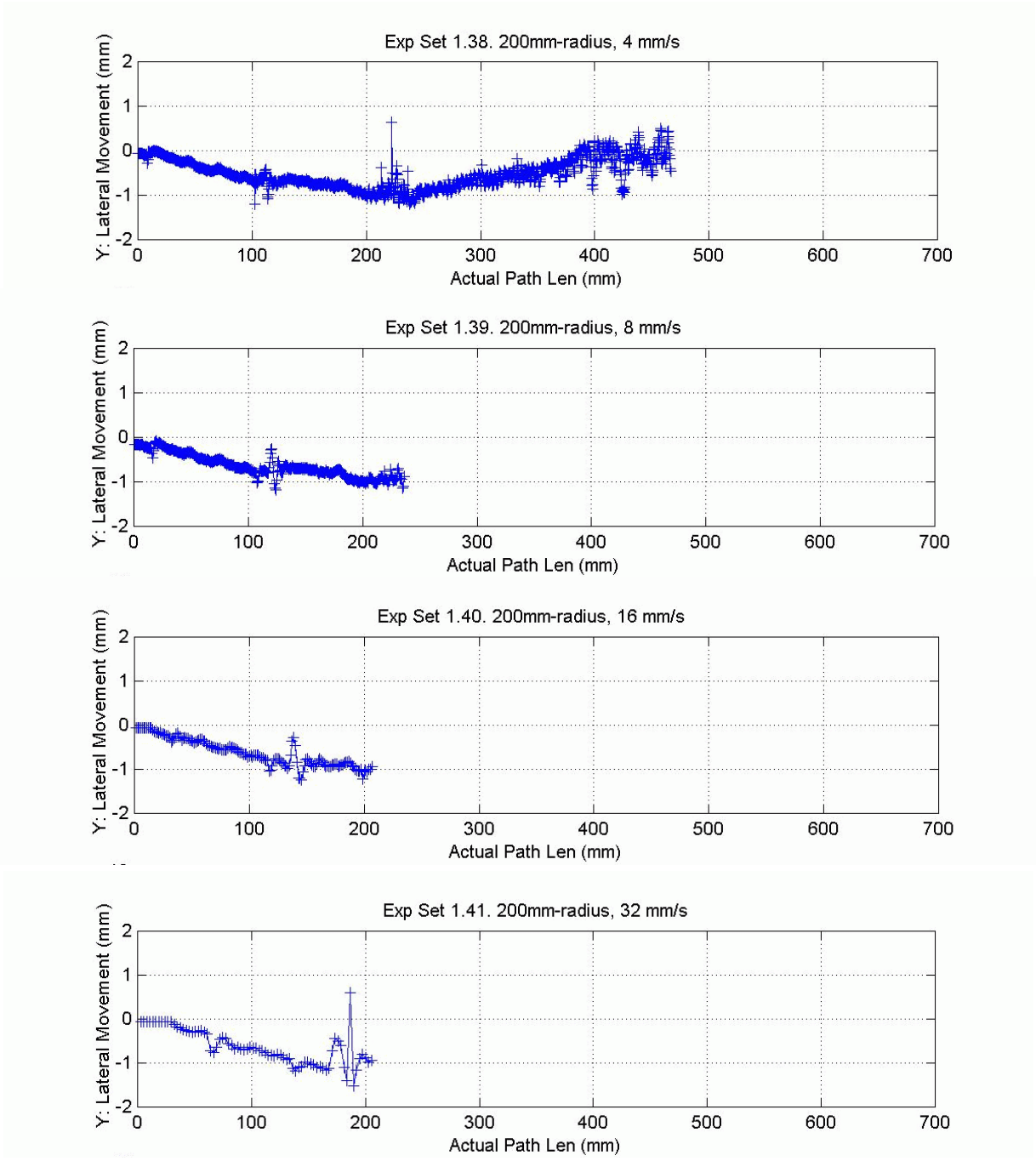


Figure 6.12: Lateral Seam Tracking system response to Ramp Input Radius of 200mm

In Figure 6.13, it can be seen that the X-axis carries a depth error of up to 5mm throughout the seam tracking trials. Later in the trial, oscillations can be seen which are centered at approximately the 5mm depth level indicating that the error is a steady-state error which could be eliminated if a degree of integral control were present. This described response is typical for the ramp input trials regardless of speed or ramp input which can be seen in Figure 6.13 which are selected X axis/depth data sets.

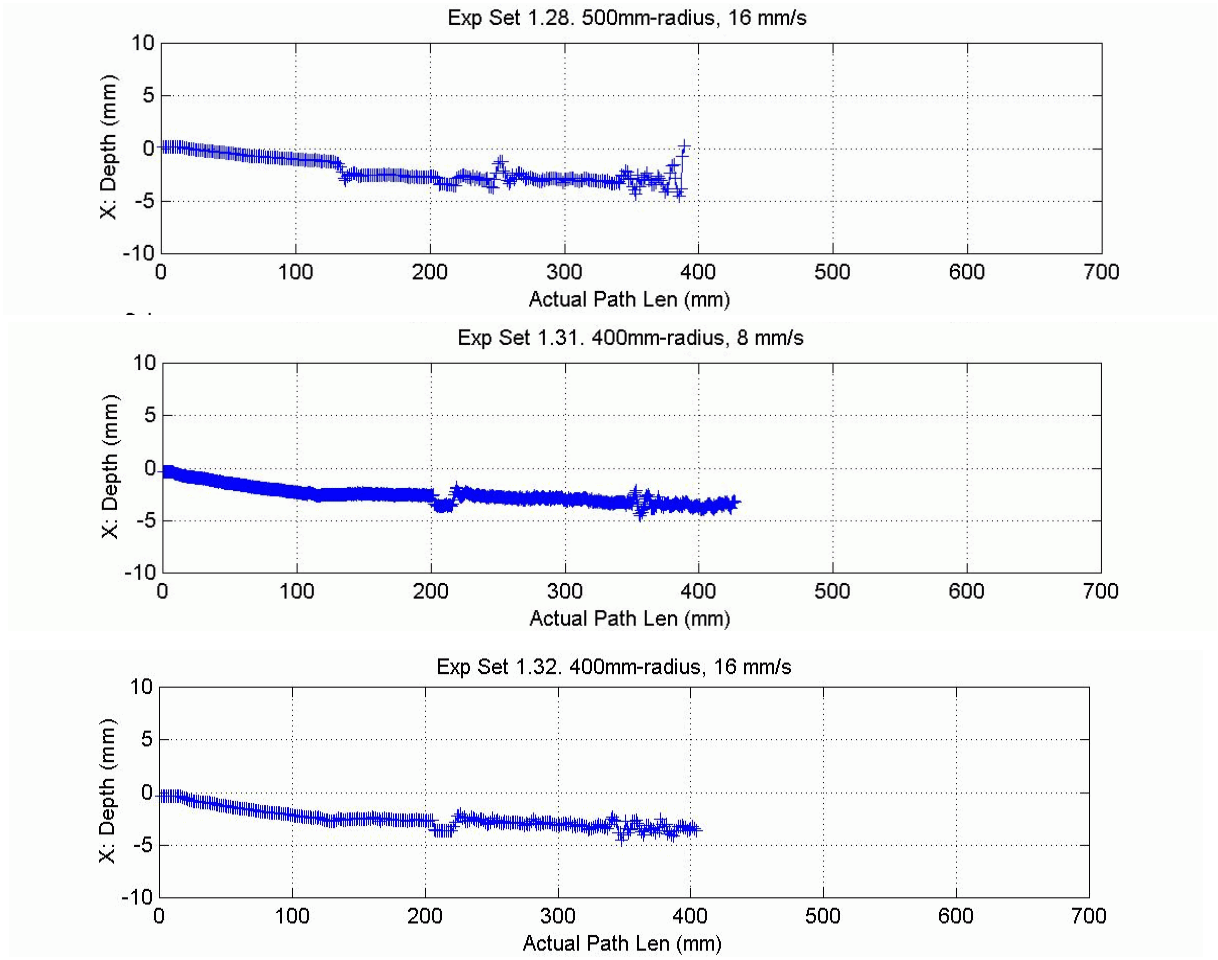
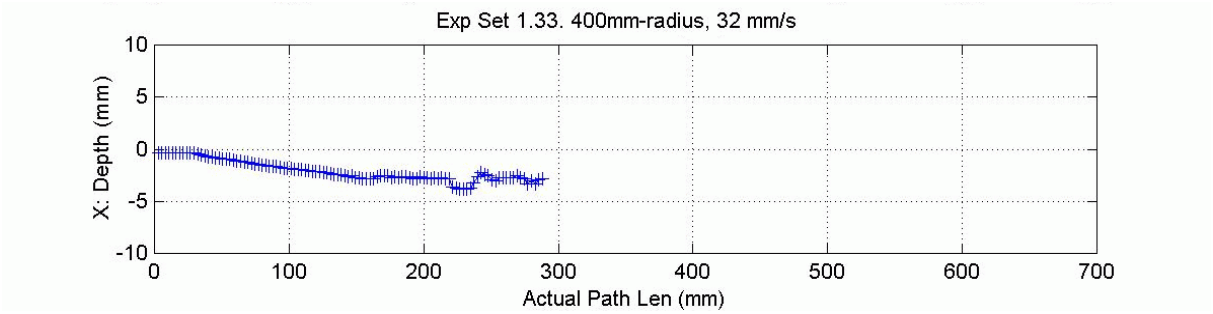
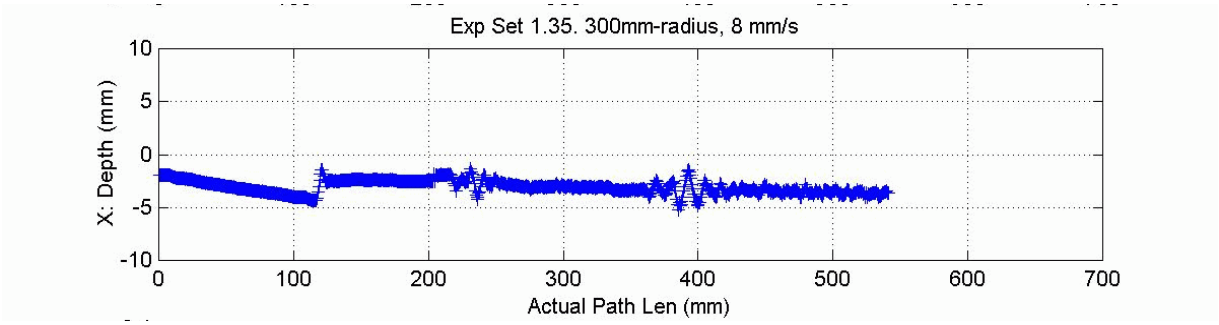


Figure 6.13. Selected X axis/Depth responses to ramp input





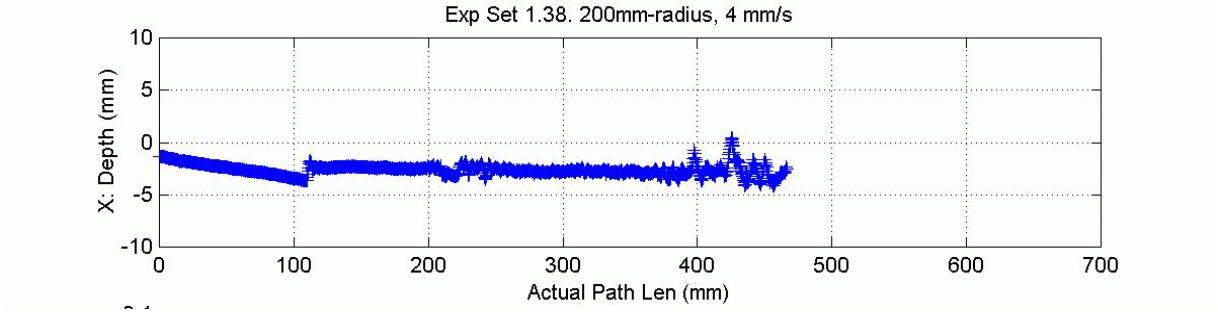


Figure 6.13. Selected X axis/Depth responses to ramp input (cont'd)

6.2.3. Experiment 2: Stability with Respect to Radial Distance

From Experiment 1, it was observed that as the Reis Robotstar V15 moved outwards with respect to the work sphere, the robot's wrist experienced an increasing oscillatory motion as the robot reached farther to get to its target point. In Experiment 1, this rocking motion was not collected, therefore, Experiment 2 monitors the angular data of the wrist as the tool tip moves closer to the outer boundaries of the work sphere. The robot begins seam-tracking on a radial line from the z-axis using the Step Input Experimental Workpiece and concludes at approximately 600mm. For each experiment the travel speed is varied at values of 4, 8, 16,32, 64 mm/s. Since the proportional gains are not yet optimized, the values were kept constant at 0.15 for the Yaw angle and 0.25 for the Pitch angle similar to Experiment 1.

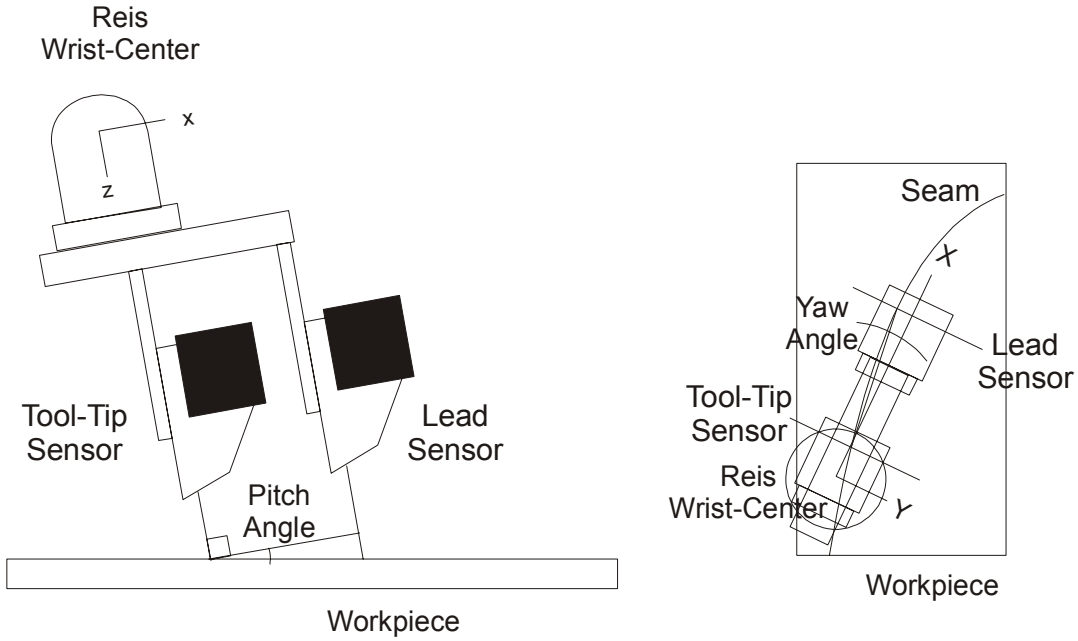


Figure 6.14. Determining the Yaw and Pitch Angles

The angular data is determined by collecting sensor data from both the lead sensor and the tool tip sensor. Since the length from sensor to sensor is known, the Pitch angle and the Yaw angle of the sensors can be calculated trigonometrically and plotted against the seam travel distance. Table

6.2 is a list of experiments to illustrate stability under different travel speeds.

Table 6.2. Stability with respect to Radial Distance and Travel Speed

ID	Track	Travel Speed	Orientation	Position
2.1	Straight-Line (step input)	4 mm/s	0^o roll, 0^o pitch	on xy radial
2.2	Straight-Line (step input)	8 mm/s	0^o roll, 0^o pitch	on xy radial
2.3	Straight-Line (step input)	16 mm/s	0^o roll, 0^o pitch	on xy radial
2.4	Straight-Line (step input)	32 mm/s	0^o roll, 0^o pitch	on xy radial
2.5	Straight-Line (step input)	64 mm/s	0^o roll, 0^o pitch	on xy radial

6.2.4. Experiment 2 Observations

The data collected is plotted and grouped into Lateral/Yaw axis data and Depth/Pitch axis data. In general, at low speeds such as 4mm/s and 8mm/s, the data is fairly smooth, however as the speed increases, fluctuation amplitudes increase especially at the latter part of seam tracking. At 32mm/s, seam tracking is cut short as the large amplitudes of oscillation cause the lead sensor to lose the seam feature thus ending the tracking. Figure 6.15 shows the lateral tool displacement as the speed increases while Figure 6.16 shows the Depth. One note of interest is that the radial line lies along a spherical partition boundary line, this causes the kinematics to switch wrist configurations given the position of the tool.

The Lateral/Yaw axis exhibits in-phase oscillations on both the lead sensor and the tool tip sensor throughout seam-tracking, thus canceling each other to produce a smooth yaw angle about zero. Figure 6.17. shows a trial at 16mm/s, although the oscillations are pronounced, they are none the less in-phase, hence there is no yaw angle oscillations. In Figure 6.18., we can see that seam tracking begins after the 100mm point has been reached on the seam. As the seam tracking continues, the system experiences increasing oscillations as it passes the 400mm mark.

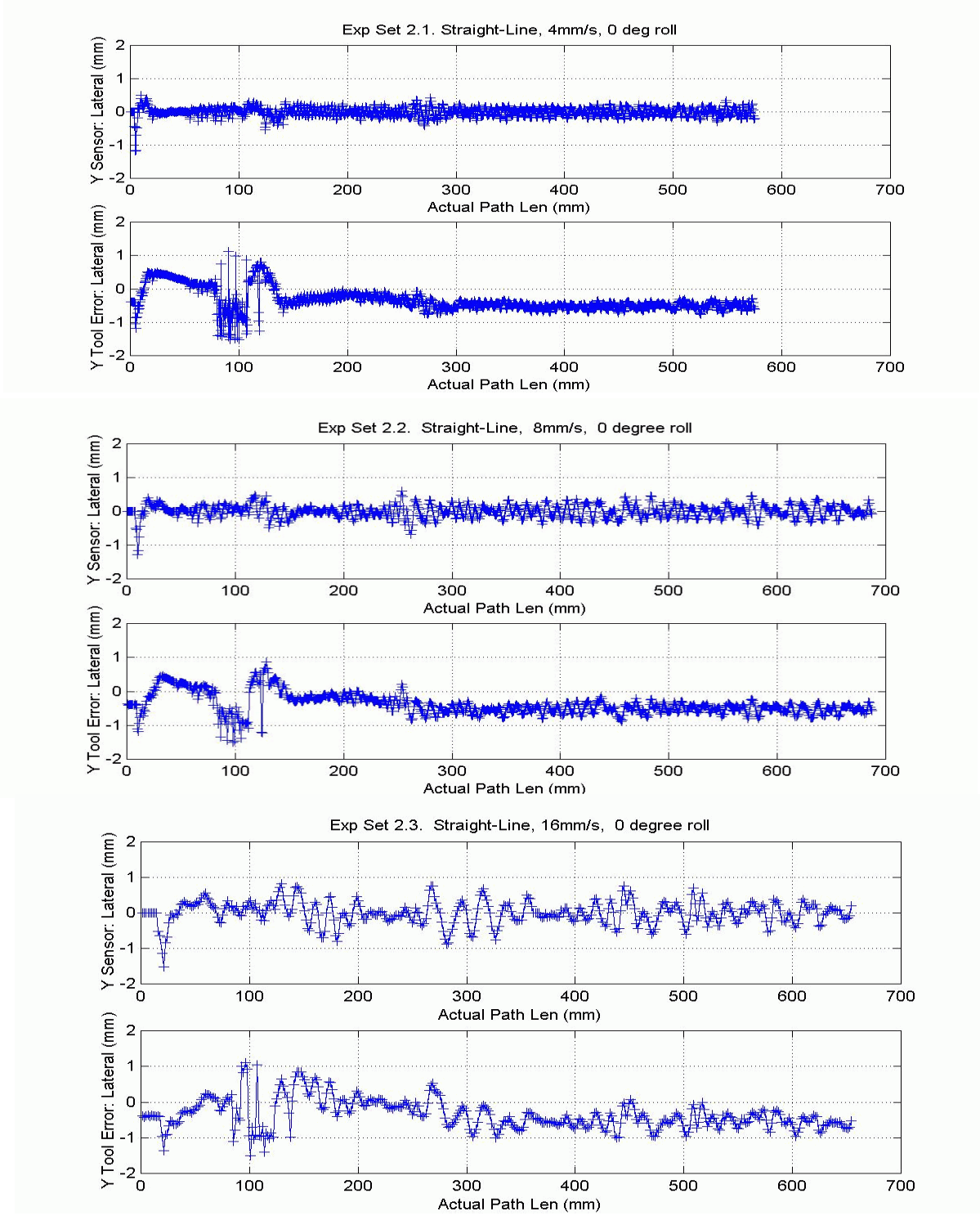


Figure 6.15. Lateral stability along a radial line as speed increases

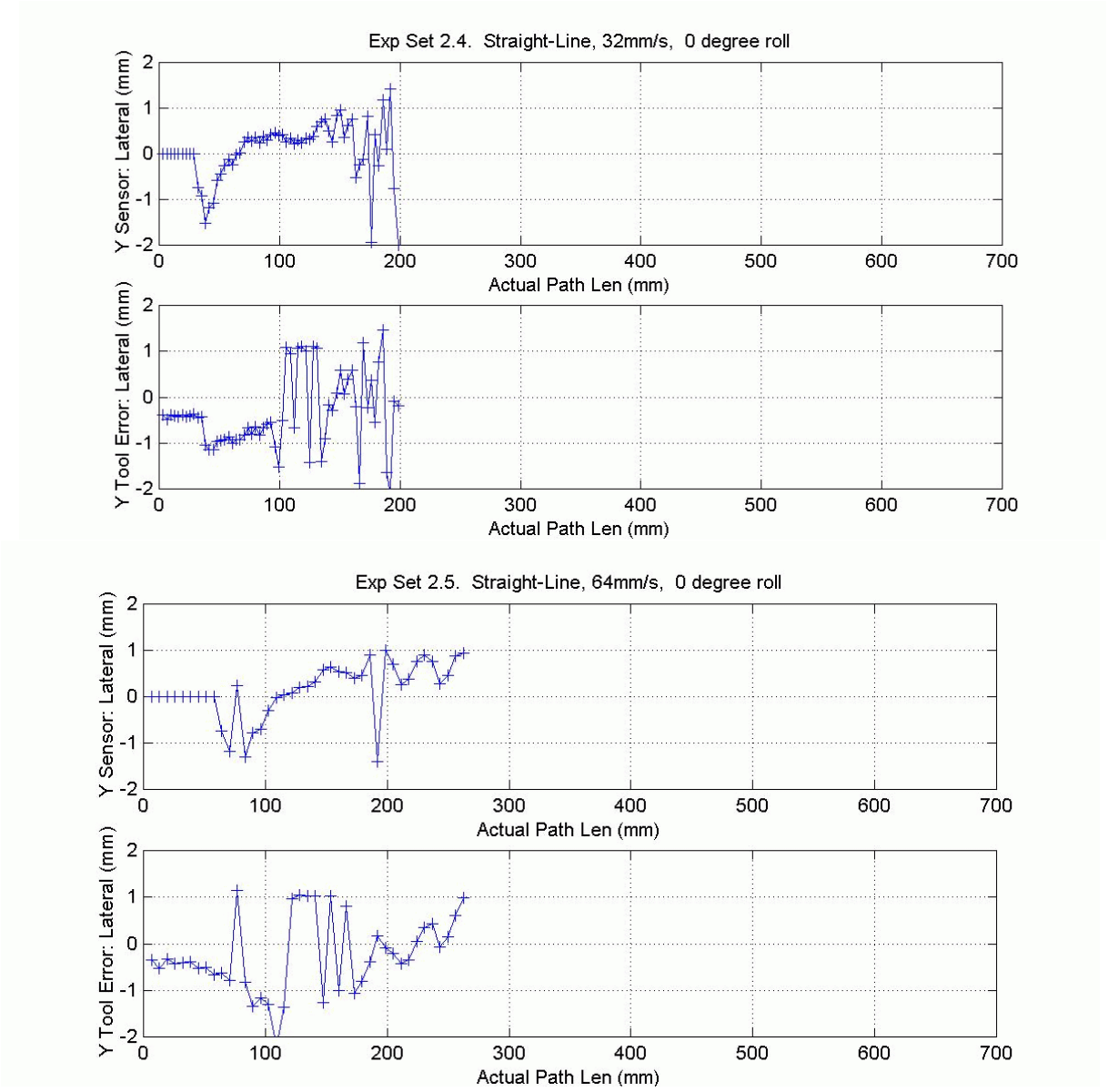
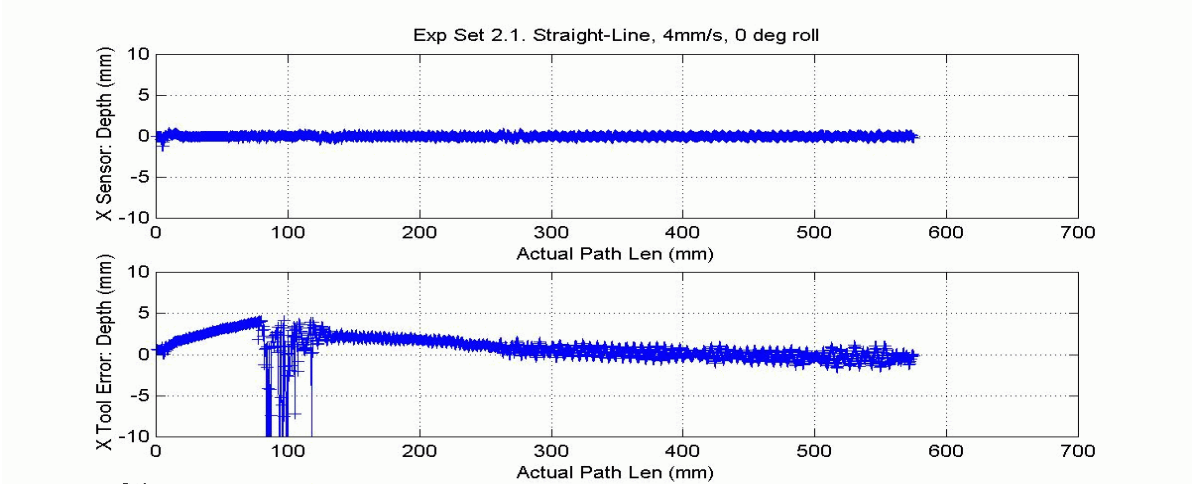
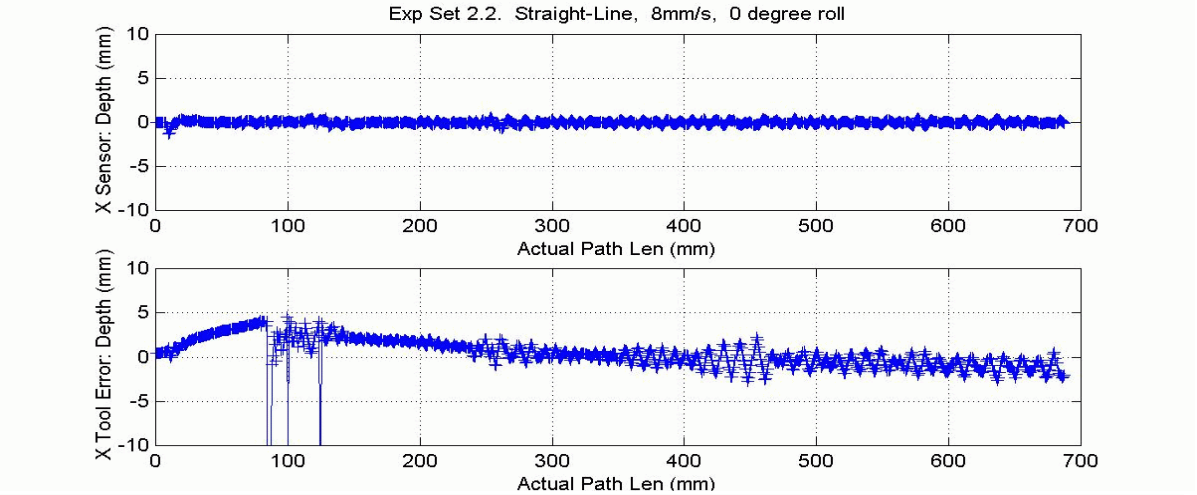
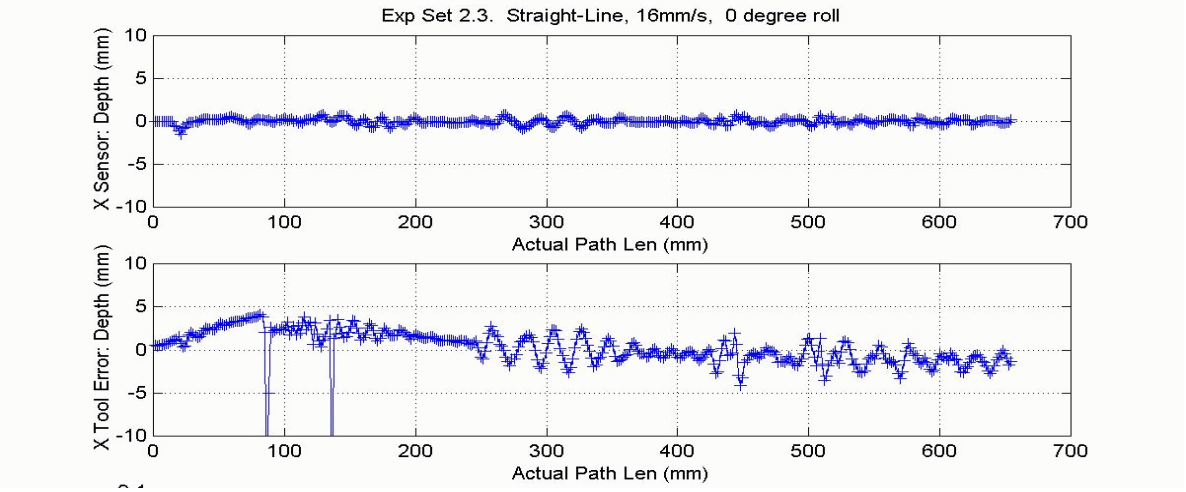


Figure 6.15. Lateral stability along a radial line as speed increases (cont'd)


Exp Set 2.1. Straight-Line, 4mm/s, 0 deg roll
X Sensor: Depth (mm)
X Tool Error: Depth (mm)
Actual Path Len (mm)


Exp Set 2.2. Straight-Line, 8mm/s, 0 degree roll
X Sensor: Depth (mm)
X Tool Error: Depth (mm)
Actual Path Len (mm)


Exp Set 2.3. Straight-Line, 16mm/s, 0 degree roll
X Sensor: Depth (mm)
X Tool Error: Depth (mm)
Actual Path Len (mm)

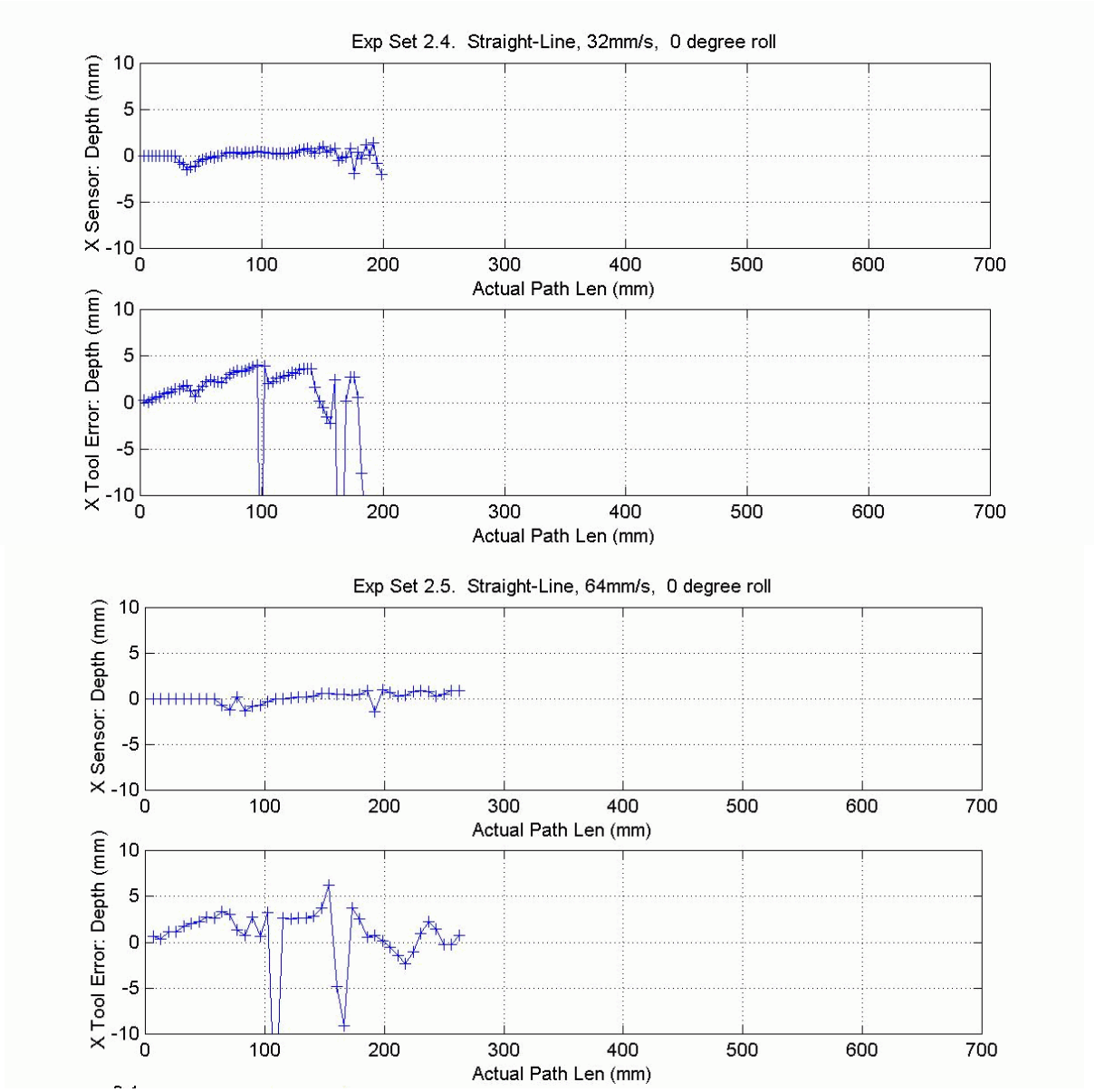


Figure 6.16. Depth stability along a radial line as speed increases

Figure 6.16. Depth stability along a radial line as speed increases (cont'd)

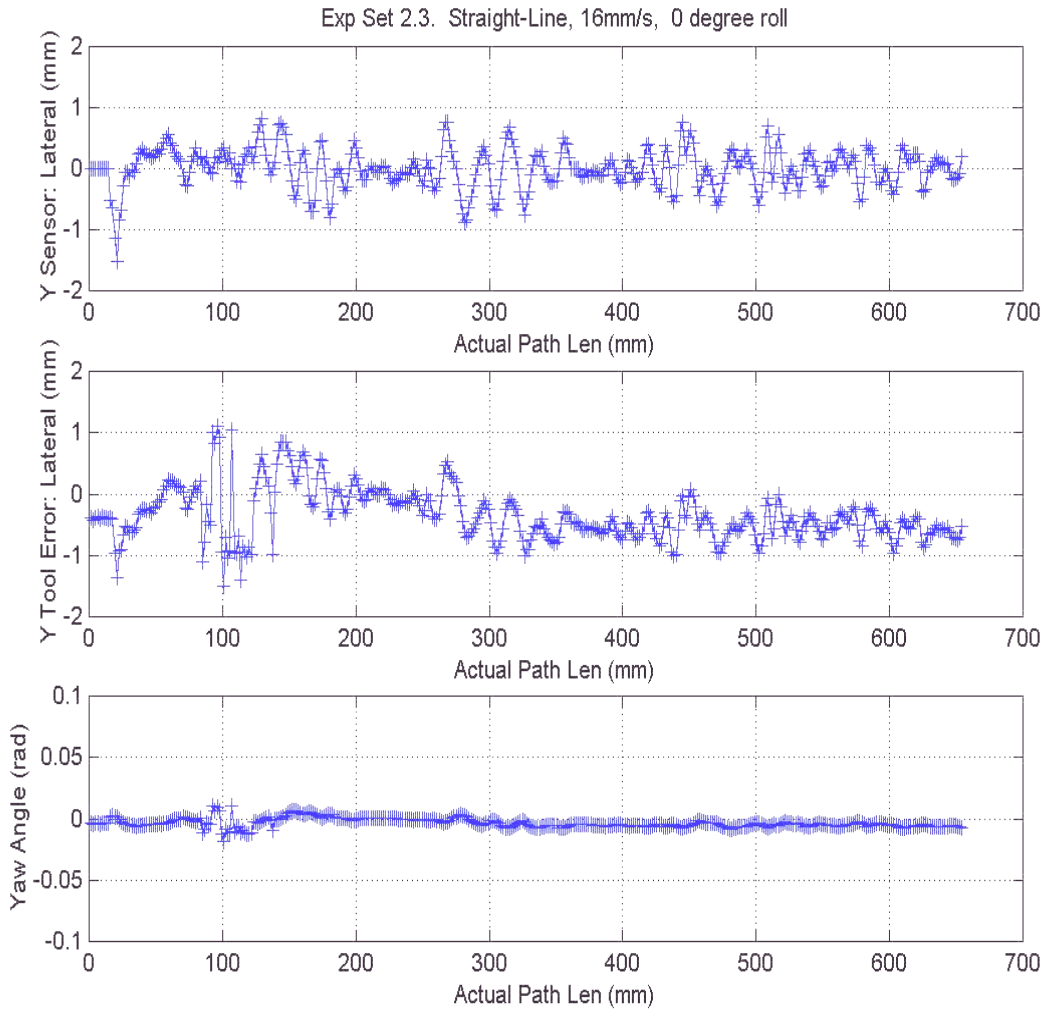


Figure 6.17. Lateral/Yaw axis stability on a radial line at 16mm/s

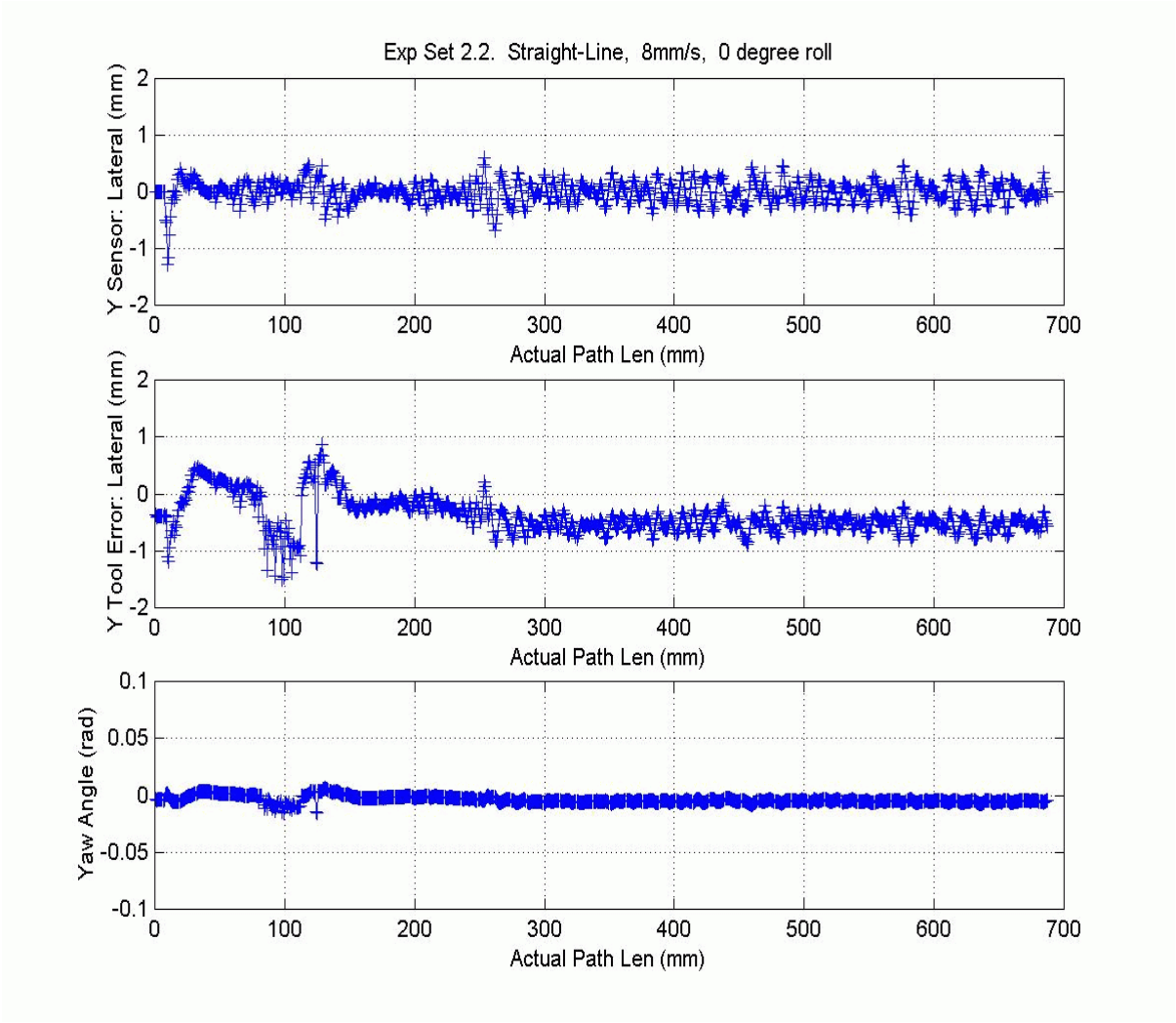


Figure 6.18. Lateral/Yaw axis stability on a radial line at 8mm/s

The Depth/Pitch axis data collected shows that the lead sensor data is maintained at zero by the proportional controller. We can see that seam-tracking begins after 100mm of seam have been traveled. At 16mm/s we can see that the depth data collected does not contain as much oscillations as the lateral data in Figure 6.15. There is a pronounced disturbance at 300mm which is in phase for both the lead sensor and the tool position resulting in a region of oscillation. As the seam tracking progresses oscillations become present and increase as the limit of the

workspace is reached. Since the lead sensor maintains itself at zero throughout the seam-tracking, any tool position movement results in a pitch angle rotation in addition. The oscillation peaks of the lead sensor are opposite in direction to the tool sensor resulting in larger rotations in the pitch axis. Figure 6.19. shows the Depth/Pitch angle data for a trial along the radial line conducted at 16mm/s and is very representative of the other experiments.

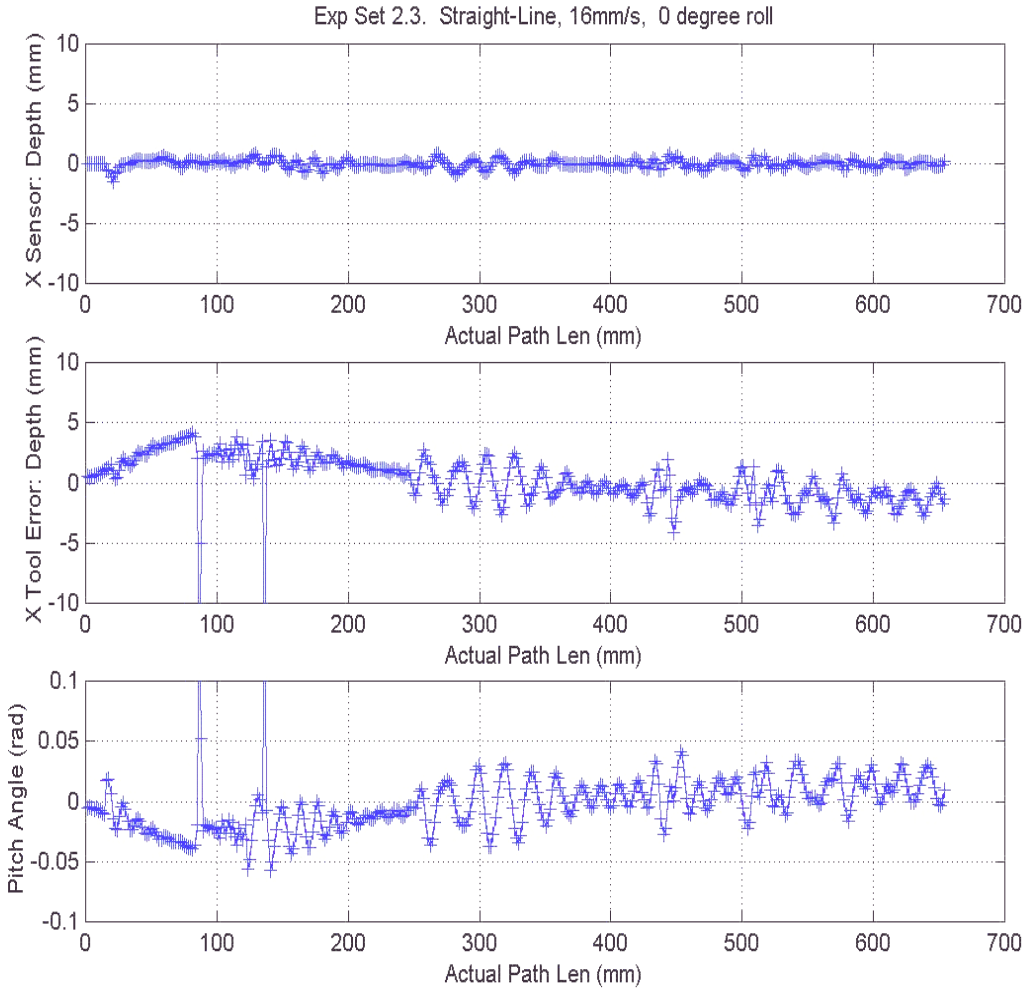


Figure 6.19. Depth/Pitch axis stability on a radial line at 16mm/s

6.2.5. Experiment 3: Gains and Transport Delays

Experiment 1 shows that at a 30 degree roll, a very stable configuration for seam tracking is reached with a step input. The purpose of Experiment 3 is to explore the effects of the proportional gain for the Pitch and Yaw axis controllers. By choosing the most stable configuration, we can eliminate almost all other variables and vary either the Yaw gain or the Pitch gain. For the Pitch gain experiments, the Yaw gain is kept constant at 0.15, while the Pitch gain is varied at values of 0.05, 0.15, 0.25, 0.35, and 0.45. For the Yaw gain experiments, the Pitch gain is kept constant at 0.15 while the Pitch gain is varied at values of 0.25, 0.35, 0.55, and 0.75. In addition, each set of gains were conducted at 16 mm/s and the other at 32 mm/s to show the difference in response due to the increased travel speed. Once again the stability of the system can be determined by looking at the Pitch and Yaw angle of the sensor configuration.

Transport Delays are interruptions in the delivery of data. Likewise, the 30 degree Roll configuration is used in this experiment set for stability under a step input. In these experiments, the sensory data from the MVS profiling sensor is delayed a number of control cycles (30Hz), and the stability is monitored. One of the safety features built into the seam tracking program is that if invalid data is supplied to the brain, the x, y and roll angle are set to zero. This usually causes the seam-tracking system to continue on a straight course and then resuming seam tracking once new data is available. If the delay is small, the Seam Tracking system should be able to handle the delay, however larger delays can cause instabilities.

For the Transport Delay Experiments, the Seam Tracking program was modified to support simulation of Transport Delays. A buffer corresponding to the size of Transport Delay cycles is created at the start of the program. At a fixed point the delay is executed. This causes the sensor data to be set to zero for the number of Transport Delay cycles. The actual data is stored in the buffer. When the delay is over, the Seam Tracking resumes but data which is delayed by the number of Transport Delay cycles is used instead. Actual data from the sensor is then routed to the rotating buffer and delayed before being used for seam tracking. Table 6.3. is a list of experiments for varying Gains and Transport Delays.

Table 6.3. Stability with respect to Gains and Transport Delays

ID	Track	Travel Speed	Orientation/ Position	Gain / Delay
3.1.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.05
3.2.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.15
3.3.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.25
3.4.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.35
3.5.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.45
3.6.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (yaw) = 0.05
3.7.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.25
3.8.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.35
3.9.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.55
3.10.	Straight-Line (step input)	16 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.75
3.11.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.05
3.12.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.15

ID	Track	Travel Speed	Orientation/ Position	Gain / Delay
3.13.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.25
3.14.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.35
3.15.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.45
3.16.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (yaw) = 0.05
3.17.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.25
3.18.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.35
3.19.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.55
3.20.	Straight-Line (step input)	32 mm/s	30º roll, 0º pitch, position (i)	Kp (pitch) = 0.75
3.21.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 1cycles 33.3ms
3.22.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 2 cycles 66.7ms
3.23.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 4 cycles 133.3ms
3.24.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 5 cycles 166.7ms
3.25.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 6 cycles 199.8ms

ID	Track	Travel Speed	Orientation/ Position	Gain / Delay
3.26.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 7 cycles 233.1ms
3.27.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 8 cycles 266.7ms
3.28.	Straight-Line (step input)	8 mm/s	30º roll, 0º pitch, position (i)	Delay = 16 cycles 533.3ms

6.2.6. Experiment 3 Observations

From the data collected from both the lead sensor and the tool-tip sensor we see that low level gains promote a smooth response to the step input. As the gains increase, stability especially at the latter part of seam tracking is decreased. However, the same gains at higher travel speeds exhibit a much smoother response. These observations, in general are for both the Y and X axis. The X axis, however is much more stable and less affected by the increasing gains. This is evident as the gain at which the X-axis begins to exhibit unstable behavior is 0.55 while the Y-axis becomes unstable at 0.35. Hence the gains should be carefully chosen with respect to travel speed as well as seam tracking performance. Figure 6.20. shows selected plots from the Pitch Gain Experiments. Figure 6.21. shows selected plots from the Yaw Gain Experiments.

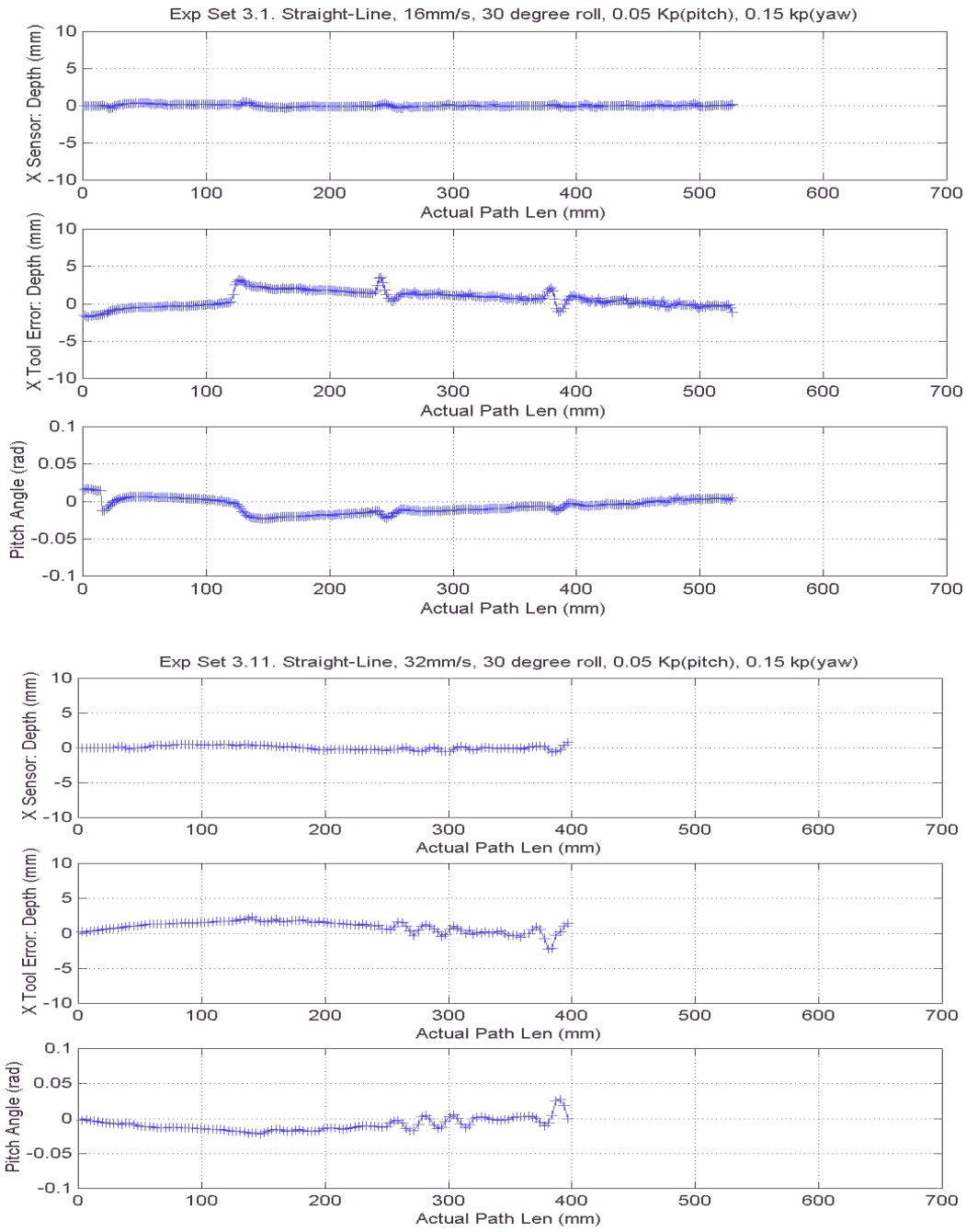


Figure 6.20: Pitch Gain Experiment, Depth stability with a step input

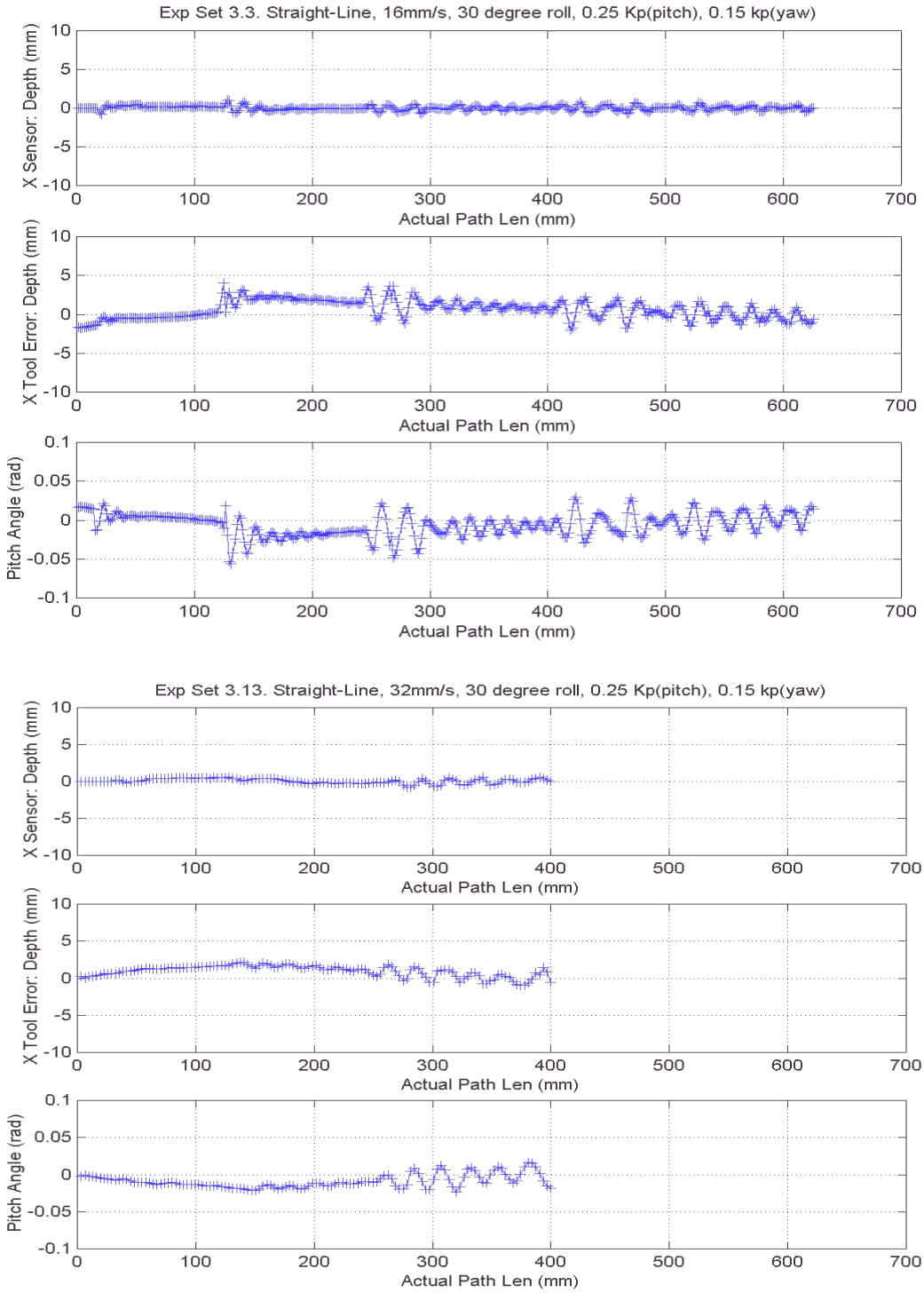
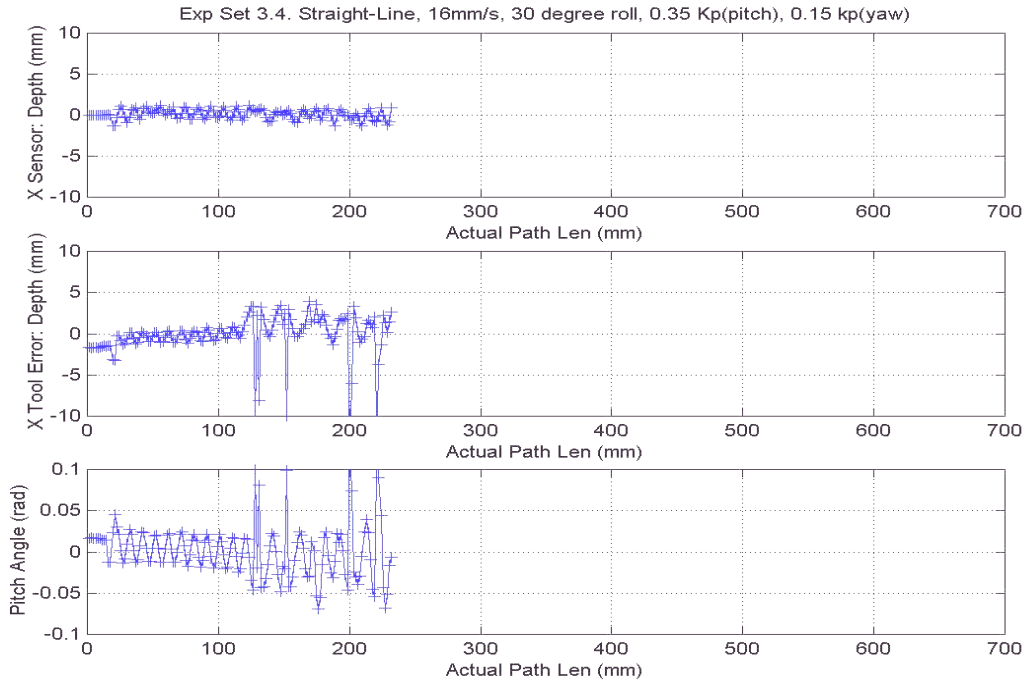
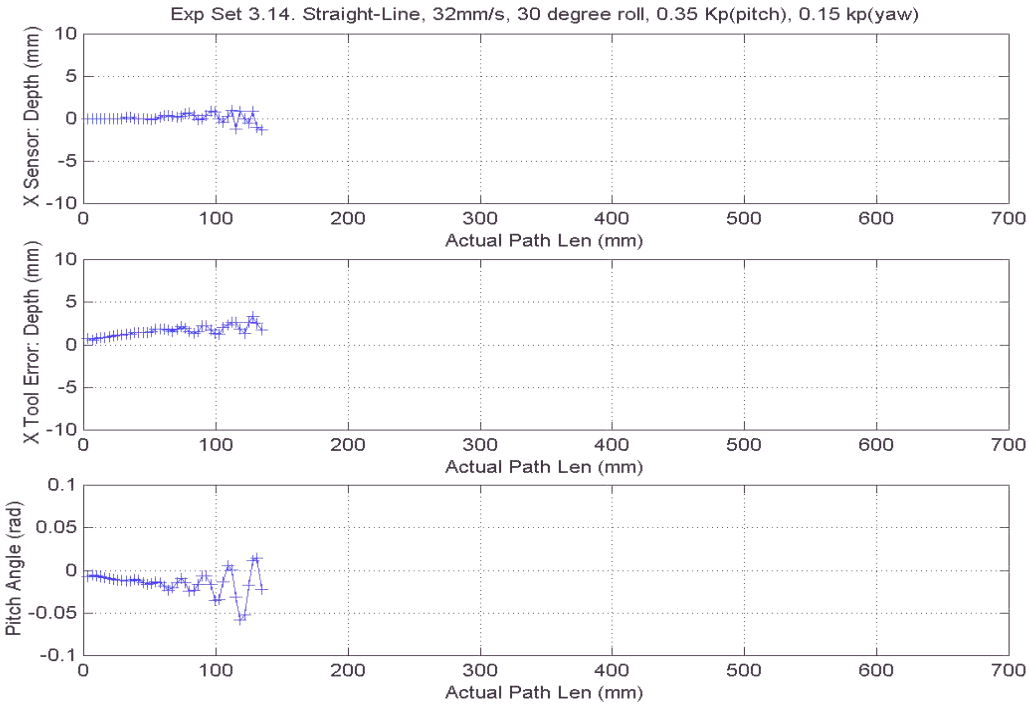


Figure 6.20: Pitch Gain Experiment, Depth stability with a step input (cont'd)


Exp Set 3.4. Straight-Line, 16mm/s, 30 degree roll, 0.35 Kp(pitch), 0.15 kp(yaw)
X Sensor: Depth (mm)
X Tool Error: Depth (mm)
Pitch Angle (rad)
Actual Path Len (mm)


Exp Set 3.14. Straight-Line, 32mm/s, 30 degree roll, 0.35 Kp(pitch), 0.15 kp(yaw)
X Sensor: Depth (mm)
X Tool Error: Depth (mm)
Pitch Angle (rad)
Actual Path Len (mm)

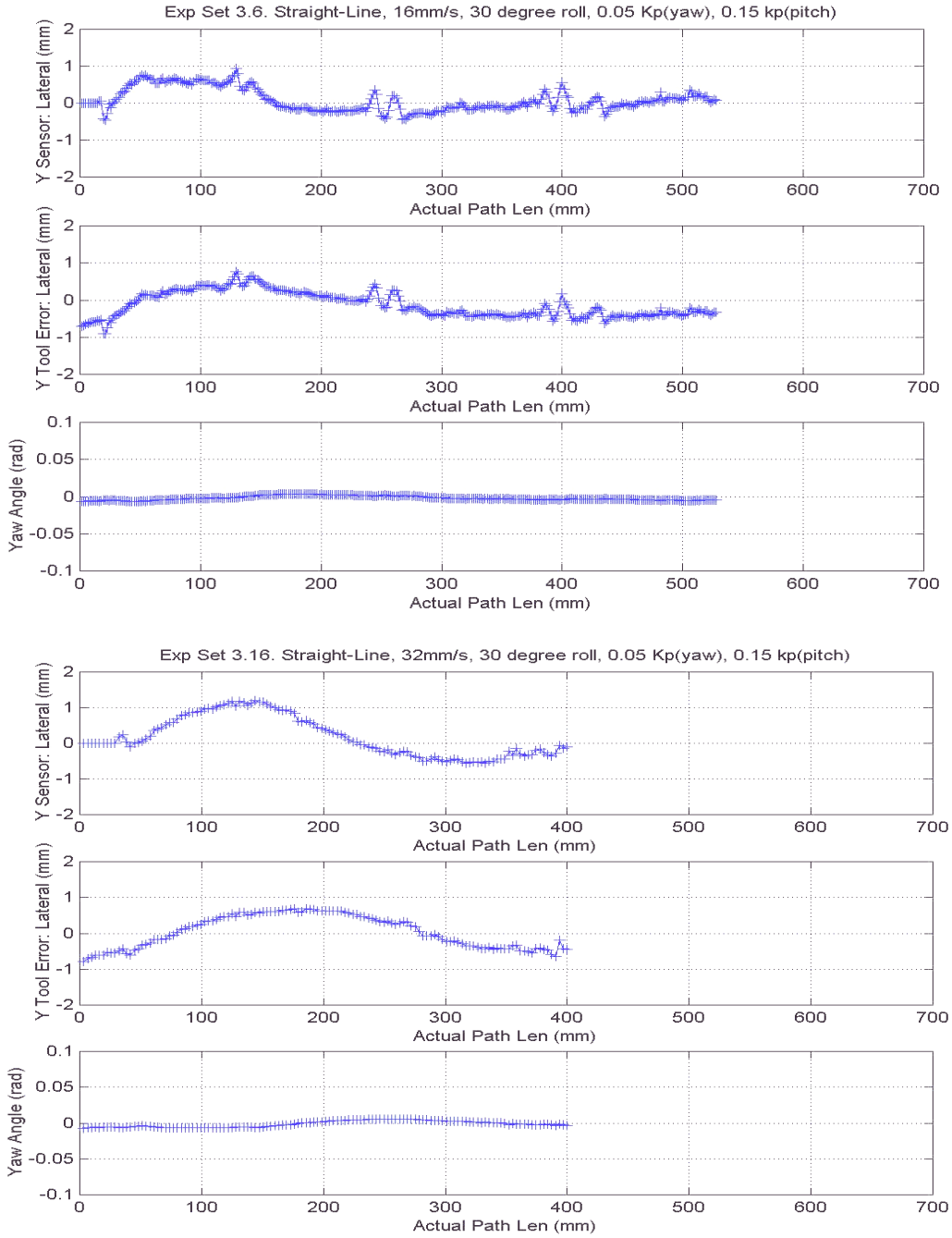


Figure 6.20: Pitch Gain Experiment, Depth stability with a step input (cont'd)

Figure 6.21: Yaw Gain Experiment, Lateral stability with a step input

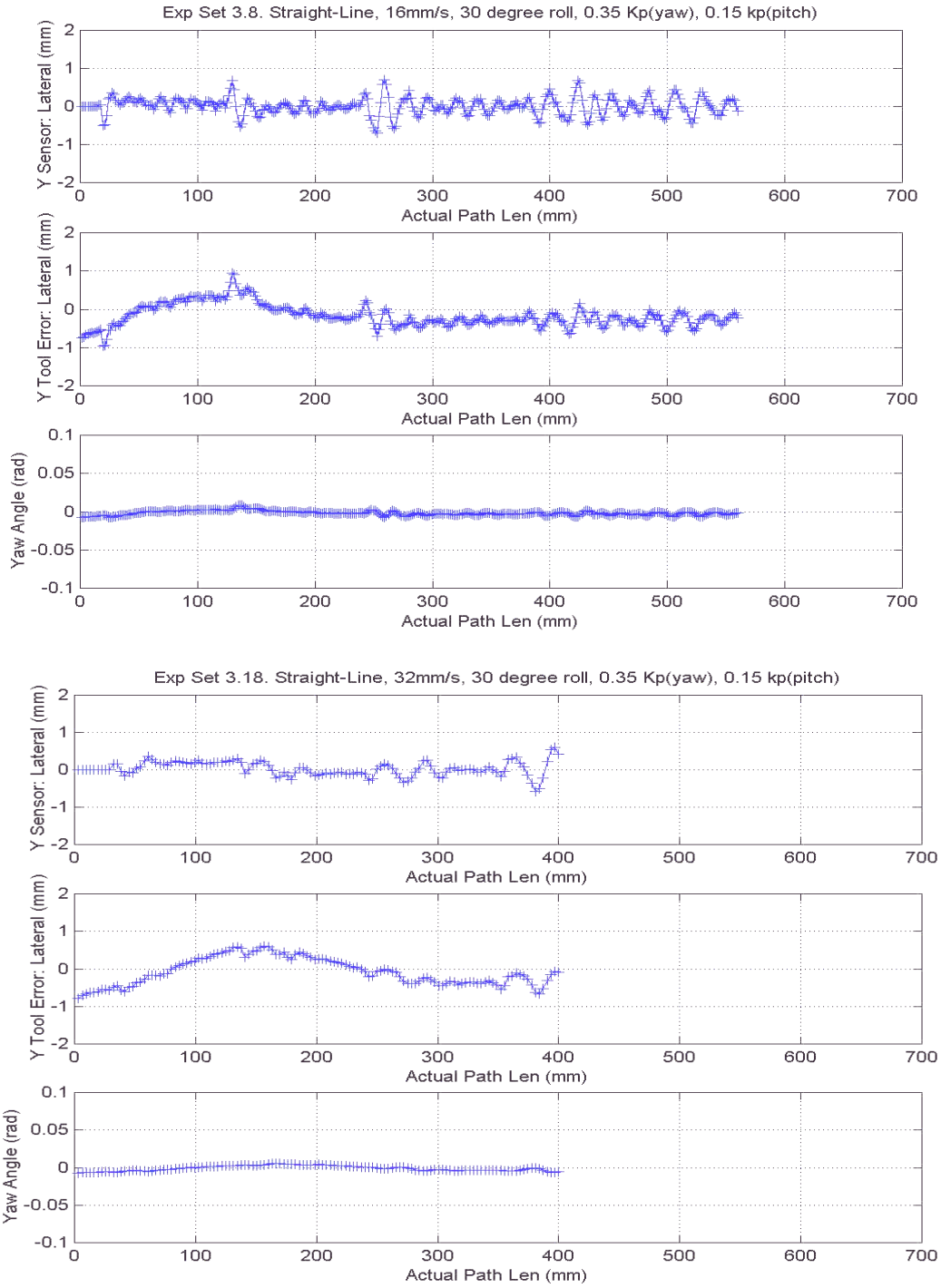
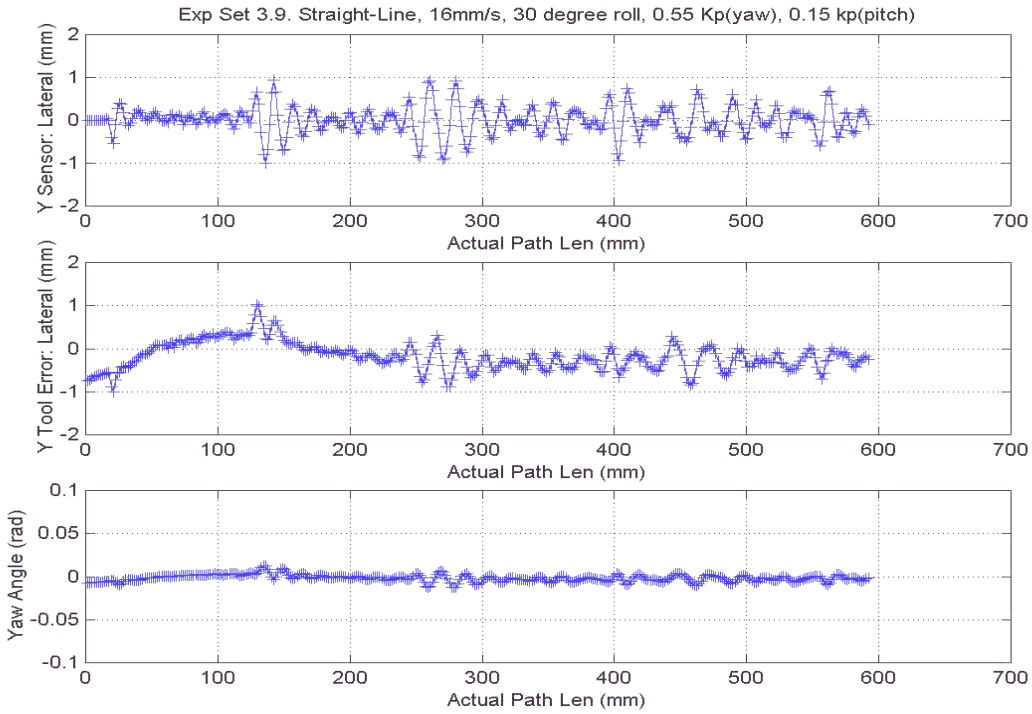


Figure 6.21: Yaw Gain Experiment, Lateral stability with a step input (cont'd)


Exp Set 3.9. Straight-Line, 16mm/s, 30 degree roll, 0.55 Kp(yaw), 0.15 kp(pitch)
Y Sensor: Lateral (mm)
Actual Path Len (mm)
Y Tool Error: Lateral (mm)
Actual Path Len (mm)
Yaw Angle (rad)
Actual Path Len (mm)

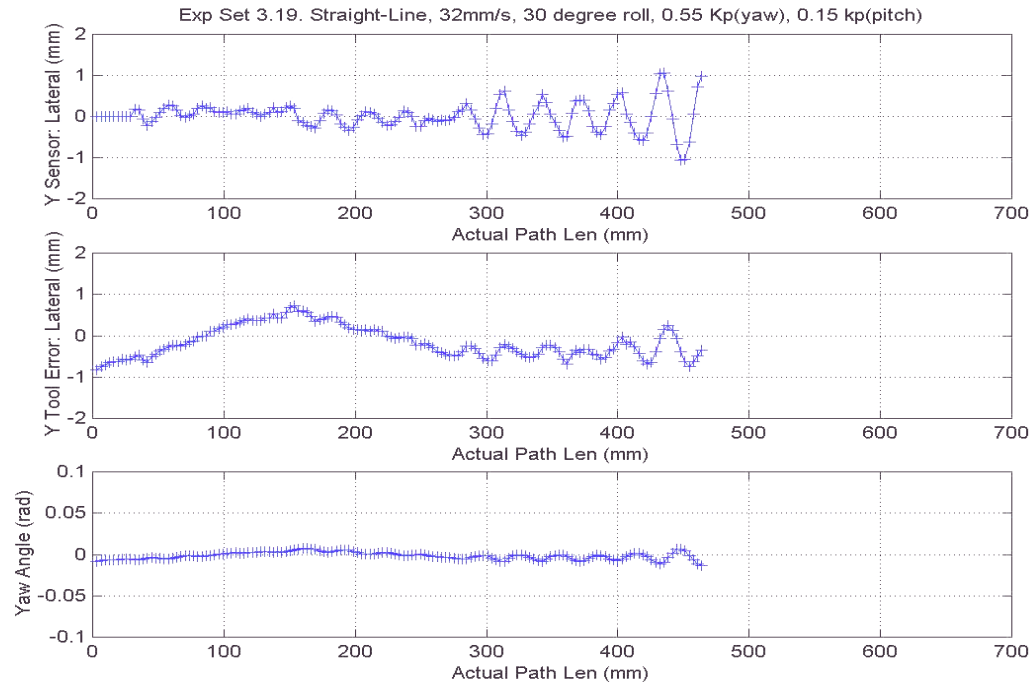

Exp Set 3.19. Straight-Line, 32mm/s, 30 degree roll, 0.55 Kp(yaw), 0.15 kp(pitch)
Y Sensor: Lateral (mm)
Actual Path Len (mm)
Y Tool Error: Lateral (mm)
Actual Path Len (mm)
Yaw Angle (rad)
Actual Path Len (mm)

Figure 6.21: Yaw Gain Experiment, Lateral stability with a step input (cont'd)

The observations for the Transport Delay experiments are discussed in this section. Actual seam tracking begins after 100mm has been traveled followed by slight oscillations at the latter part of seam-tracking which we have observed in previous experiments. The Transport Delays are injected at 150mm. From Figure 6.22 and Figure 6.23, it can be observed that a Transport Delay of 1 cycle/ 33.3ms or 2 cycles/66.7 ms has almost no effect on the Seam Tracking system and exhibit similar responses to the delay, as there is little disturbance surrounding the 150mm area.

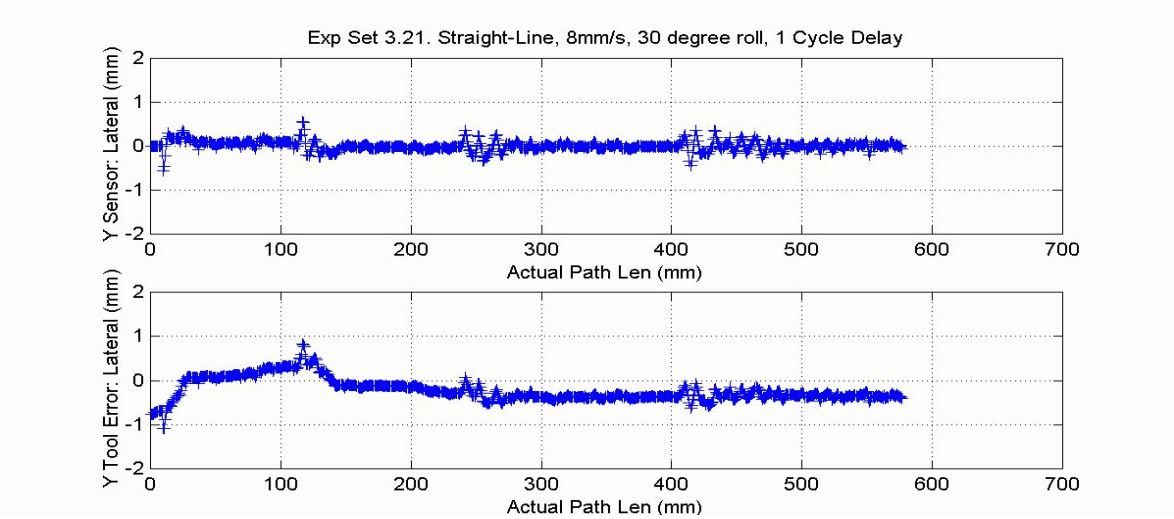
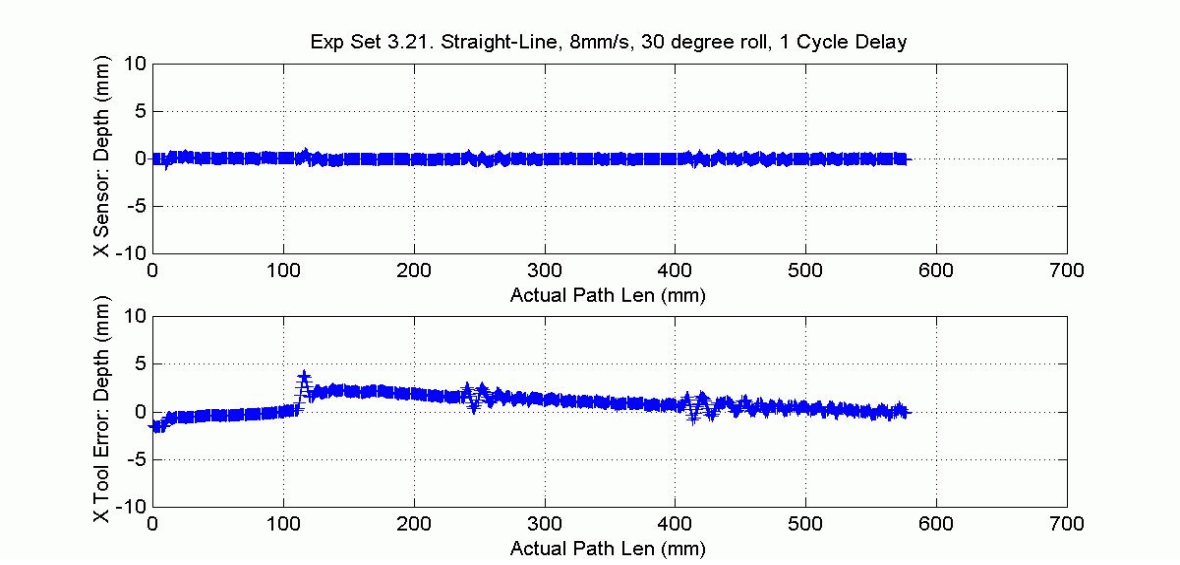


Figure 6.22. Stability data for 1 cycle Transport Delay



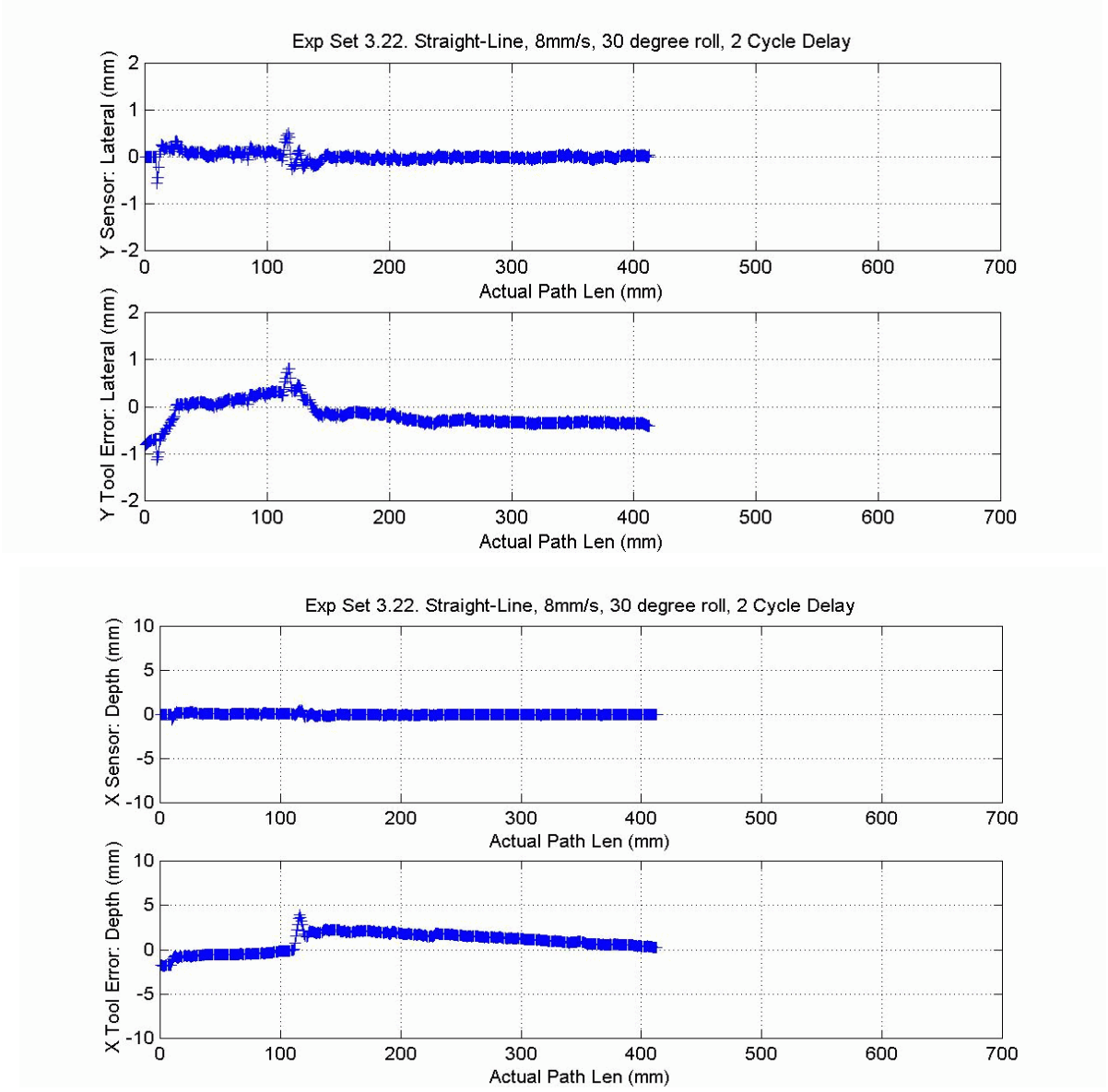


Figure 6.23. Stability data for 2 cycle Transport Delay

Figure 6.24 shows the trial conducted with a 4 cycle or 133.3ms delay . The data has the same initial behavior as the previous experiments, however oscillations begins at approximately 250mm for both the X and Y axis followed by increasing amplitude in the oscillations. At this point the system can be considered unstable.

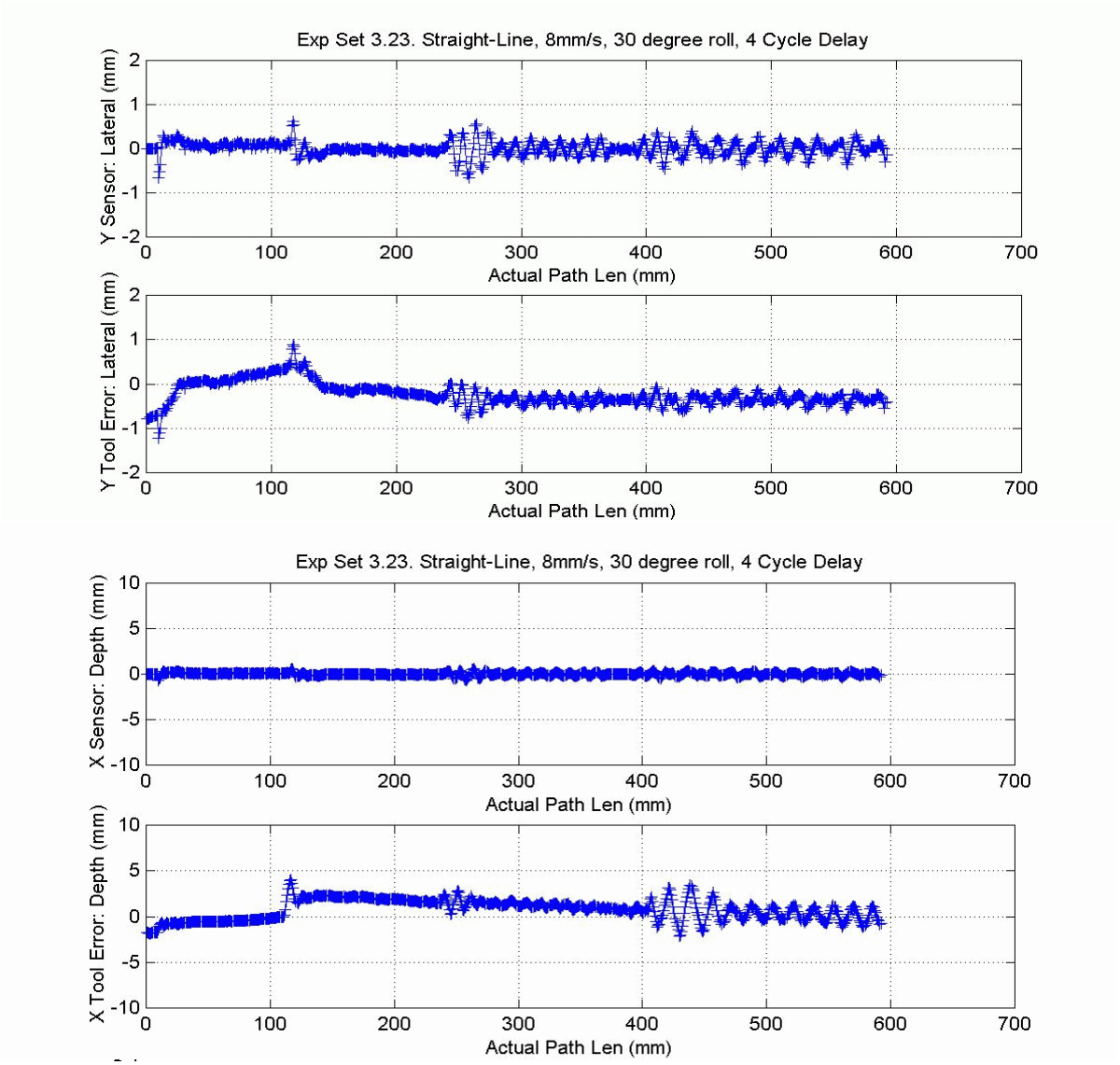


Figure 6.24. Stability data for 4 cycle Transport Delay

To better investigate the trends as the system becomes unstable, 5, 6, and 7 cycle Transport Delay plots are shown. Figure 6.25. shows the trial conducted with a 5 cycle/166.7ms delay inserted at 150mm. This figure shows a slight oscillation following the insertion of the delay, after 350mm the oscillations become significantly large and the system can be considered unstable after this point.

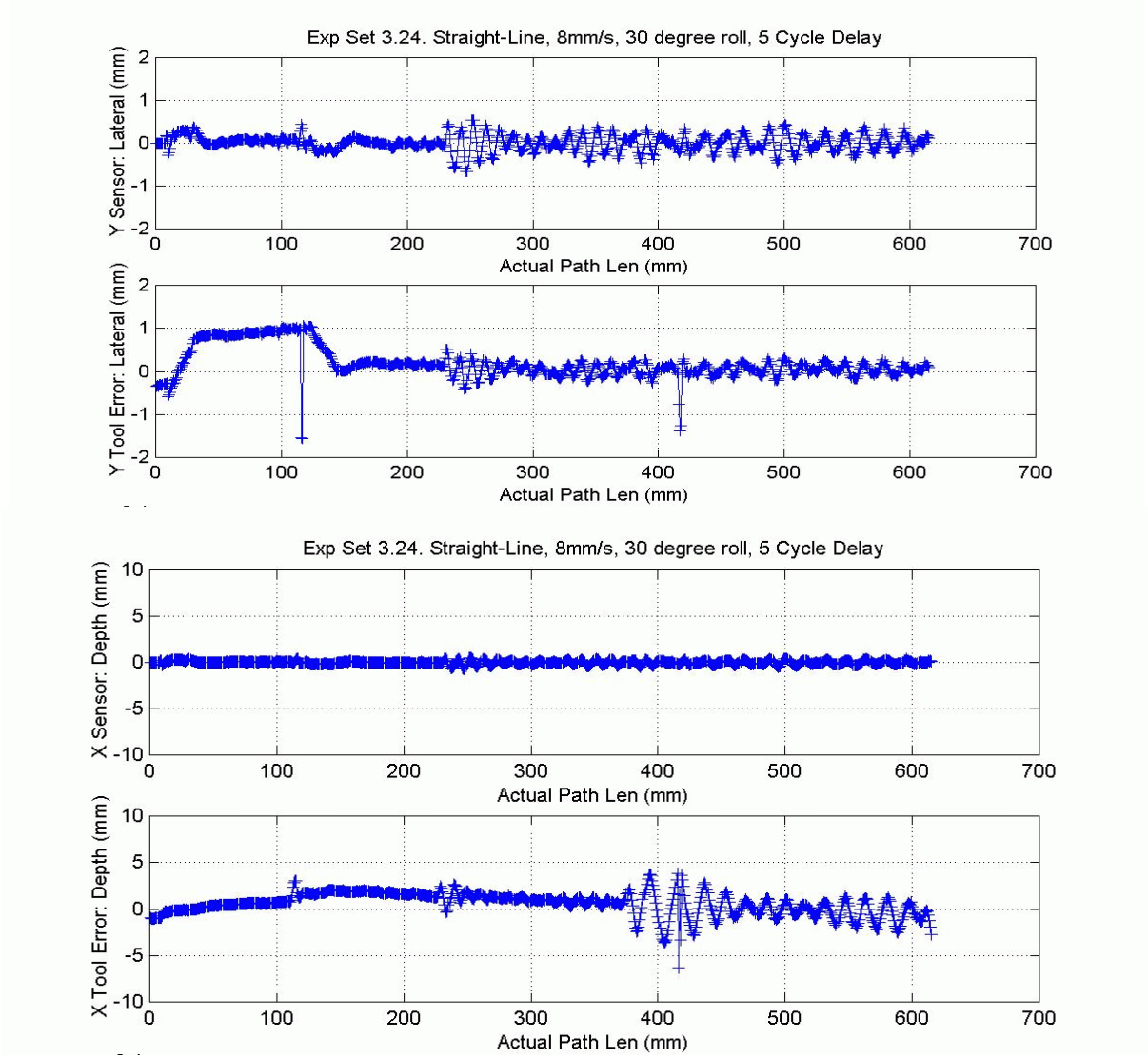


Figure 6.25. Stability data for 5 cycle Transport Delay

Figure 6.26. shows the trial conducted with a 6 cycle/200.ms delay inserted at 150mm. This figure shows a slight oscillation following the insertion of the delay, after 250mm the oscillations become significantly large and the system can be considered unstable after this point. Hence, the system is getting unstable earlier indicating that the system cannot recover from this delay.

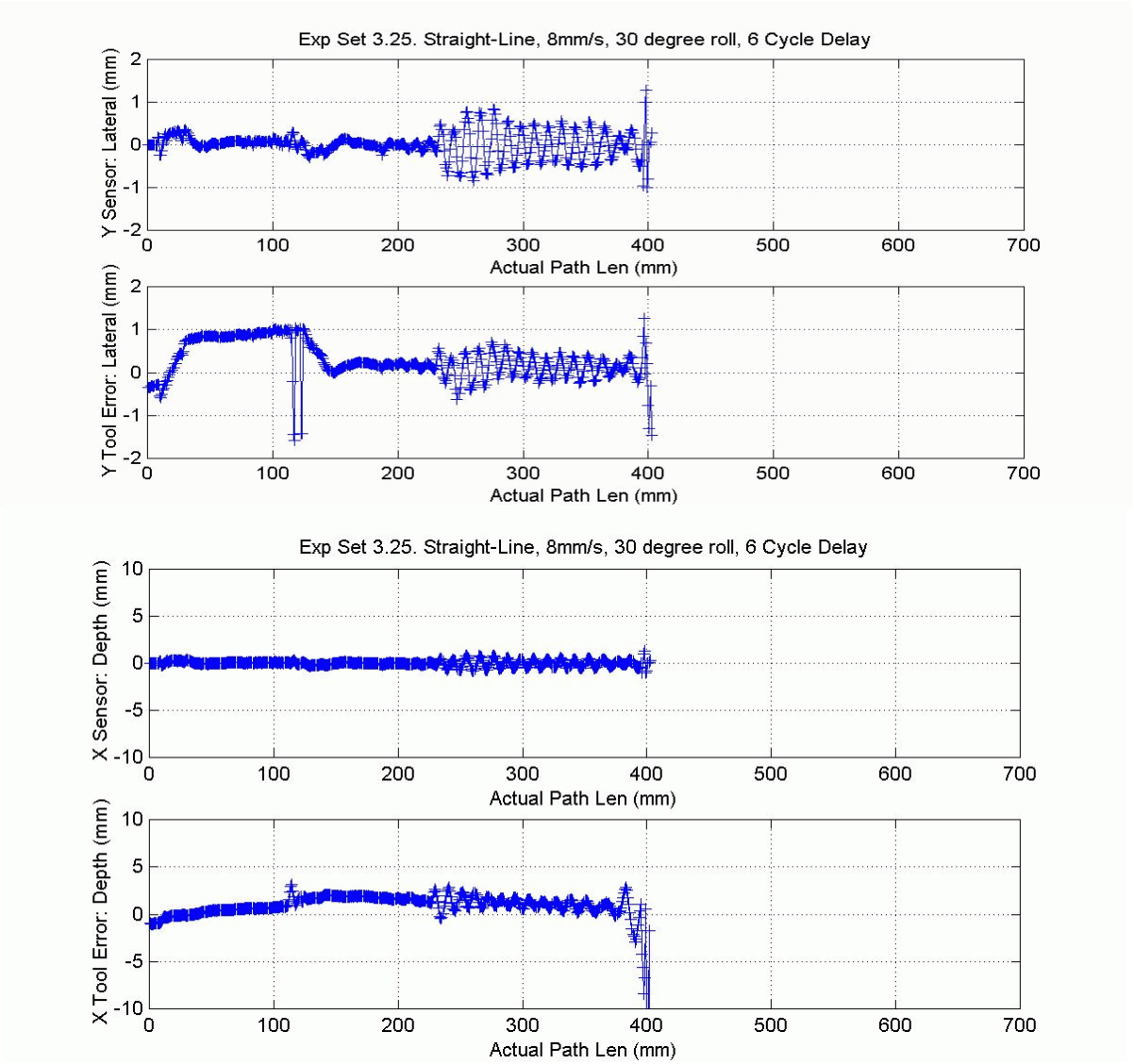
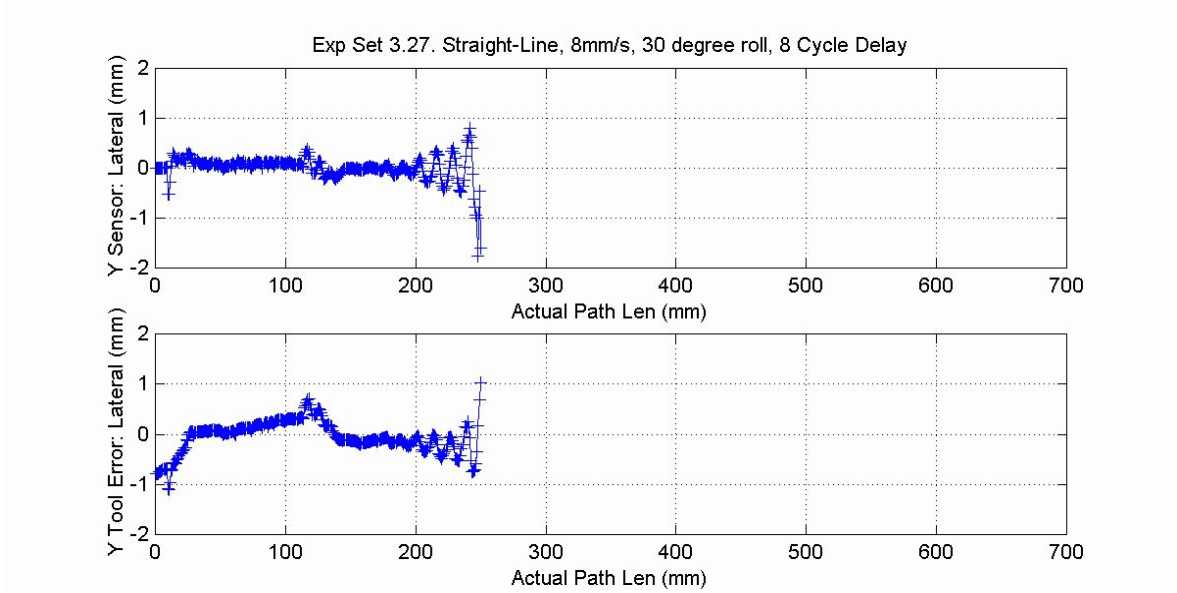


Figure 6.26. Stability data for 6 cycle Transport Delay

Figure 6.27. and 6.28 show the trials conducted with a 8 cycle/266.7.ms delay inserted at 150mm, and a 16 cycle/300ms delay. These figures shows a slight oscillation following the insertion of the delay at 150mm, after 250mm the oscillations become significantly large and the system can be considered unstable after this point.



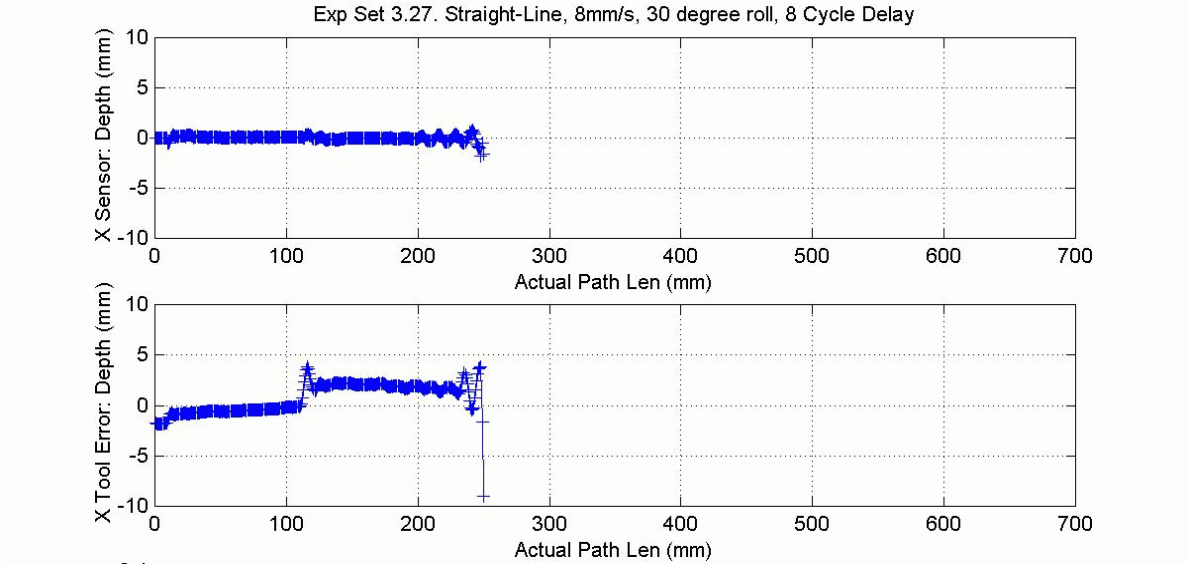
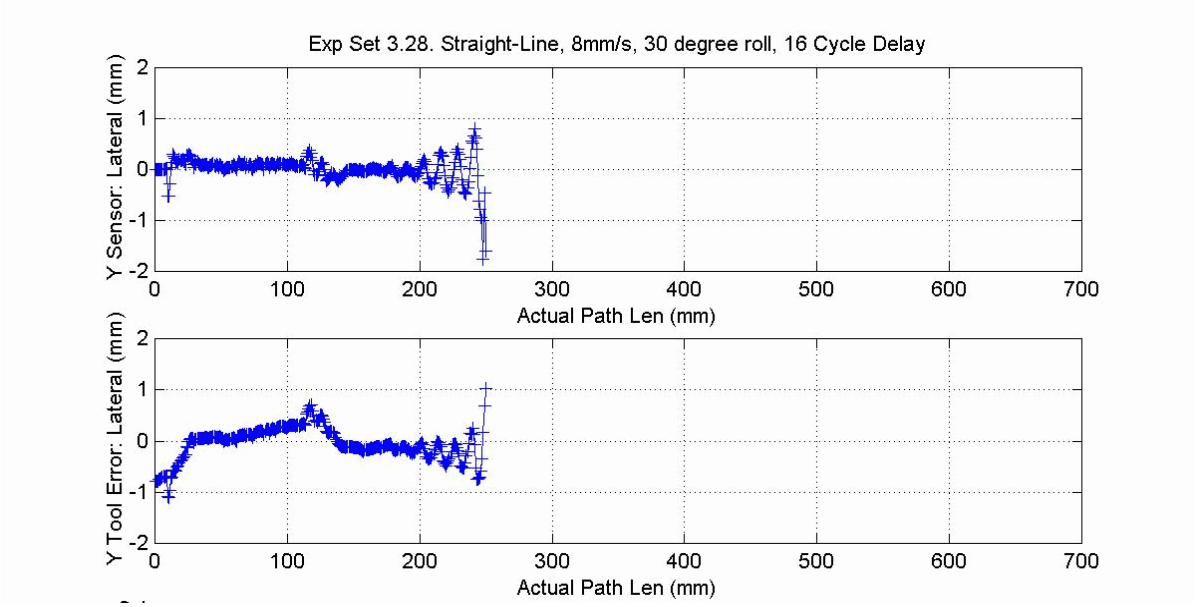


Figure 6.27. Stability data for 8 cycle Transport Delay



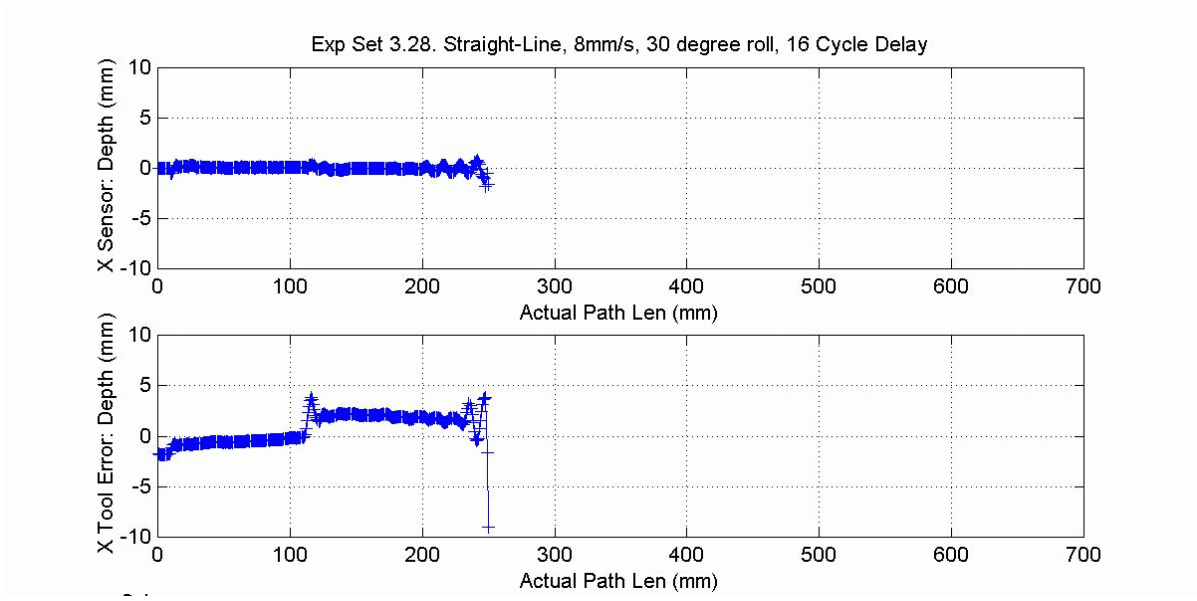


Figure 6.28. Stability data for 16 cycle Transport Delay

6.2.6. Experiment 4: Repeatability

Due to the dynamic nature of the seam tracking system, we must be able to assess the repeatability and consistency. The Repeatability Experiment consisted of using the step input workpiece with a Roll angle of 30 degrees and running the Seam Tracking program at 16 mm/s. This run was repeated 4 times with data being collected from both the lead sensor and the tool-tip sensor. Figure 6.30 and 6.31, shows the data collected during one of the executions.

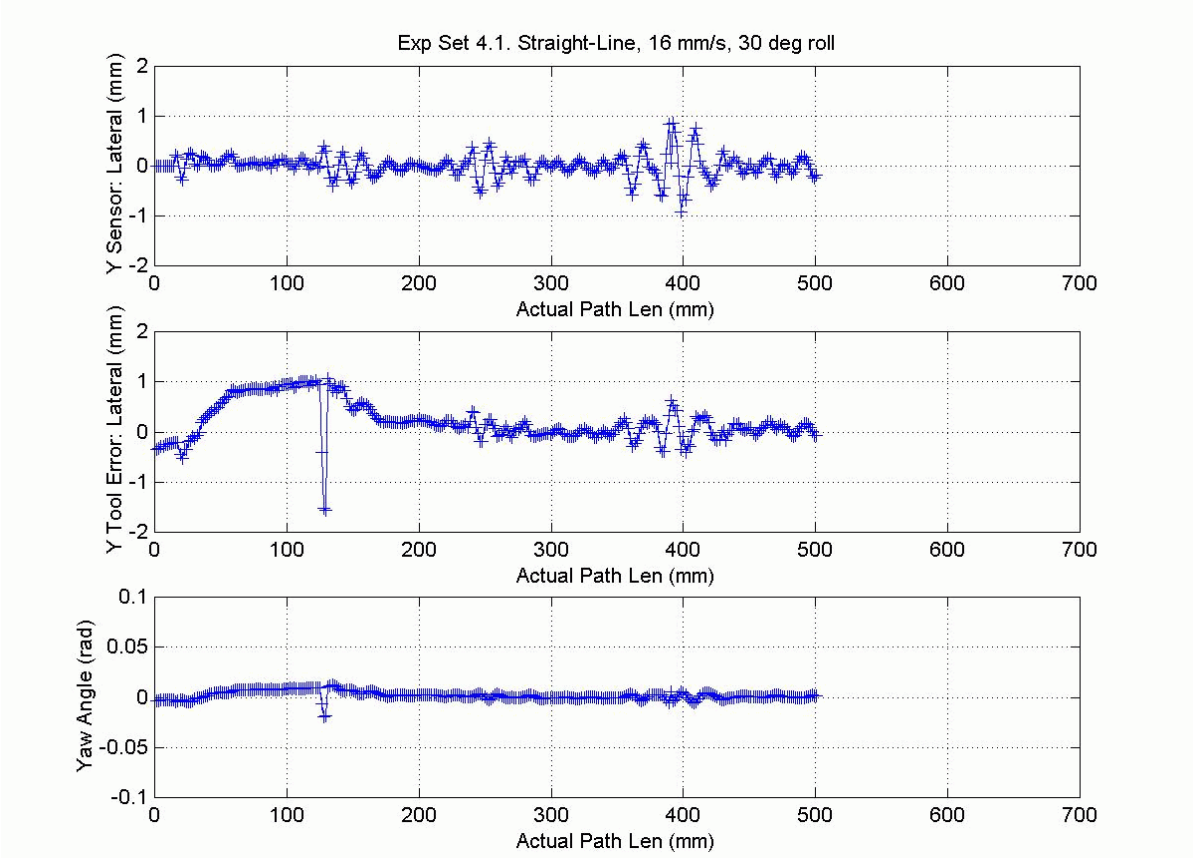


Figure 6.29: Lateral/Yaw Axis data for Repeatability Experiment

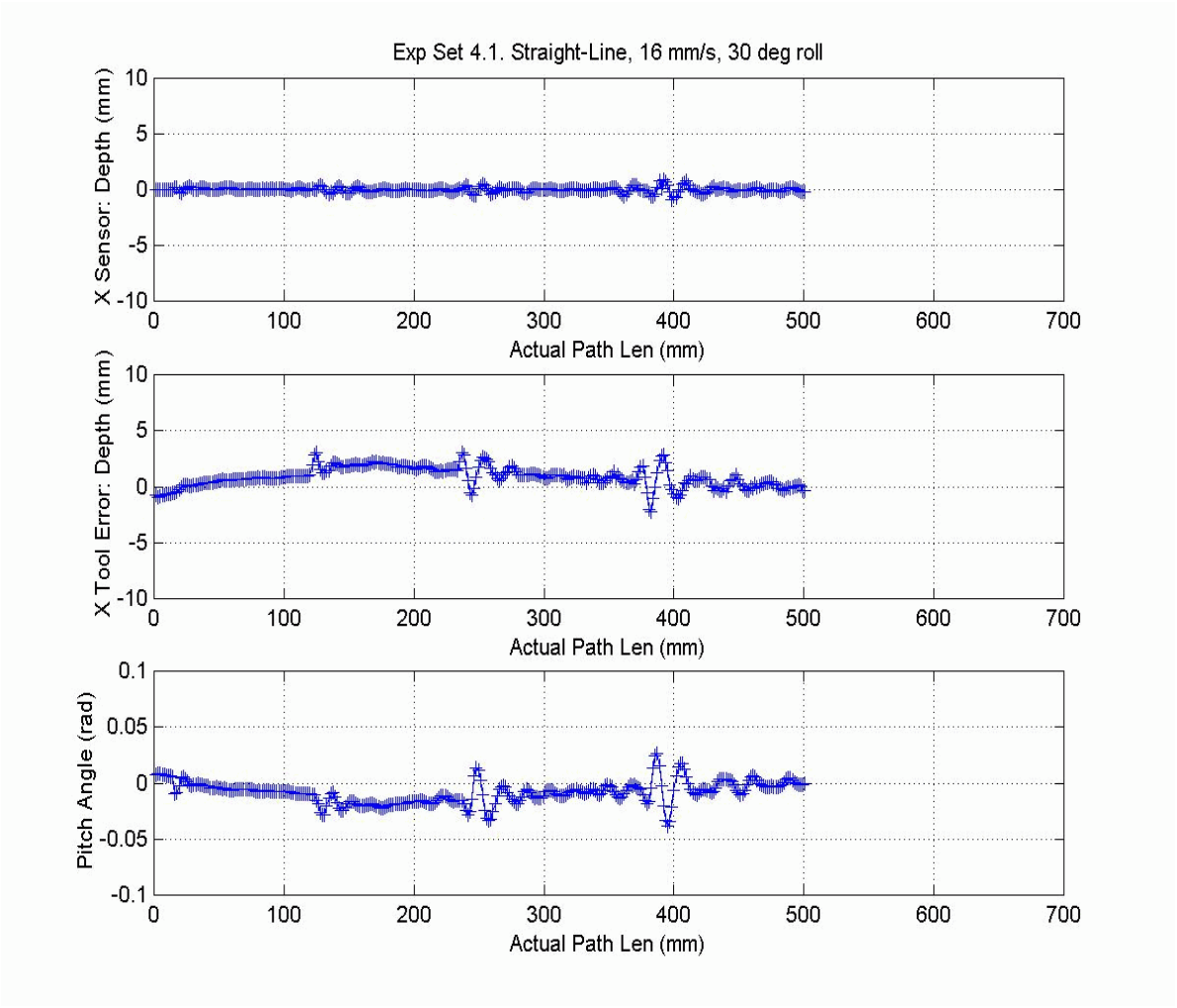


Figure 6.30: Depth/Pitch axis data for Repeatability Experiment

From the four trials, the standard deviation for each axis of each sensor is calculated. Since the angles are calculated values, they are not included in the calculation. The standard deviation calculation includes: the lead sensor lateral data, the tool lateral data, the lead sensor depth, and the tool depth. These calculations are plotted with respect to seam length in Figure 6.32 and Figure 6.33. We can observe that the data is very consistent, with the exception of the tool depth at the 400-500mm region.

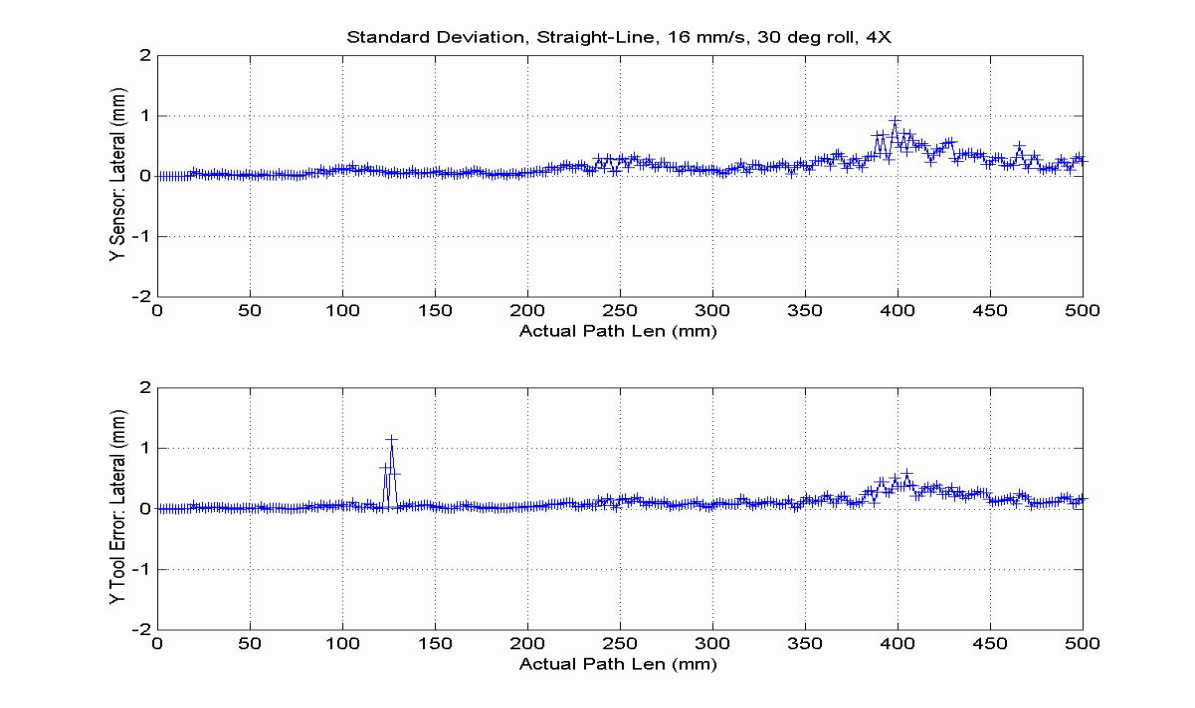
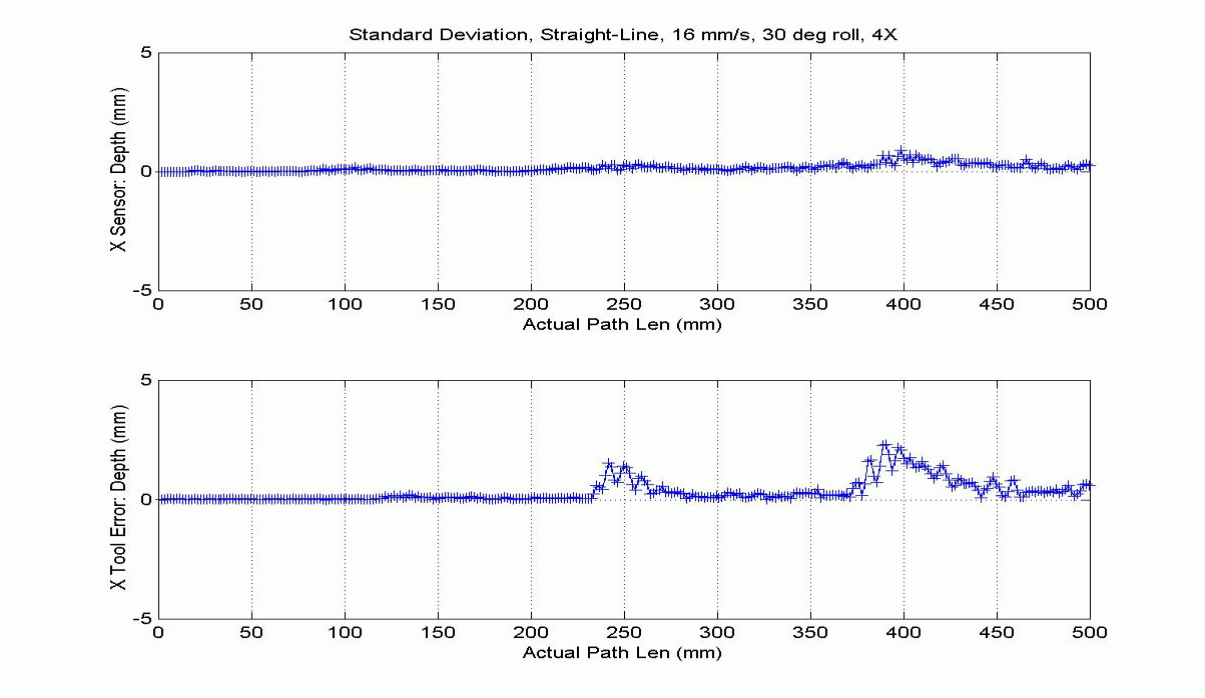


Figure 6.31: Standard deviation for Lateral data

Figure 6.32: Standard deviation for Depth data



6.2.7. Experiment 5: Alternate Wrist Setup

The purpose of the Alternate Wrist Setup Experiment is to briefly explore the possibility of another sensor configurations with respect to the Reis Robotstar's wrist. An extension block which is slanted at 30 degrees in the pitch axis is placed between the tool mounting plate and the dual sensor fixture. Therefore, Yaw axis movement will require moving joints other than just joint 6 as in the former case. However, Pitch axis movement will require less abrupt corrections and minimize some of the late oscillatory motions of the wrist . In addition, this configuration extends the range working range of the robot. Experiment Set 5 consists of a single execution of the seam-tracking program using start position (i), and conducted at a travel speed of 16mm/s.

Upon initial seam tracking trials with this configuration, it was observed that the system was unstable with the proportional gains set at 0.4 kp(yaw) and 0.2 for kp(pitch) which were determined to be optimal for the first wrist setup. Thus the gains were lowered by a factor of 4 to 0.1 kp(yaw) and 0.05 for kp(pitch) to maintain adequate stability during the trial. Figure 6.34 shows the Alternate Wrist Configuration.

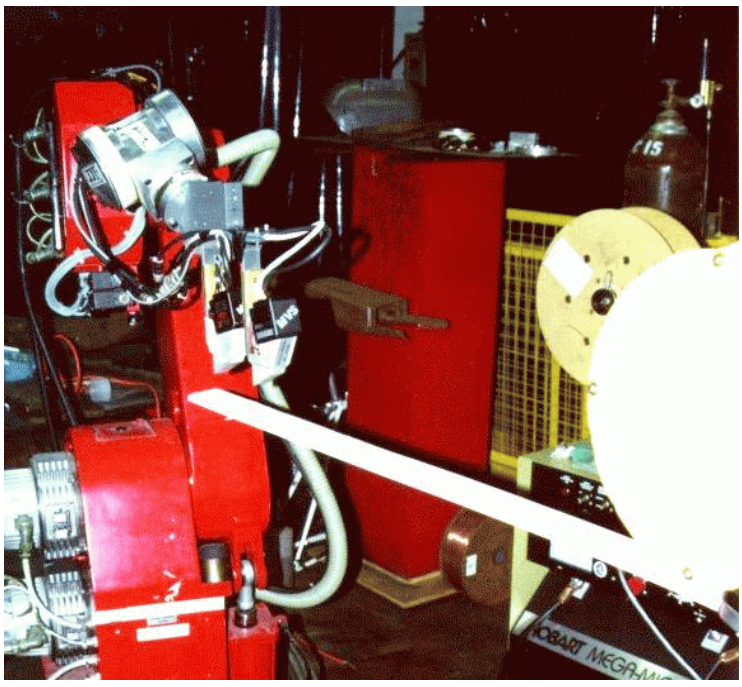


Figure 6.33. Alternate Wrist Configuration

6.2.8. Experiment 5: Observations

The raw data collected from each sensor exhibits a very long path indicating that this configuration does indeed increase the seam tracking range of the system. The lateral data shows lead sensor fluctuating slightly about the zero. The actual tool data does not begin to correct until 100mm is reached. As the seam tracking continues farther along the seam, it begins to experience a more dramatic oscillation about its target values for both the lead sensor and the tool point position. The calculated yaw angle is maintained at zero indicating minimal yaw angle oscillations and smooth travel in the yaw axis.

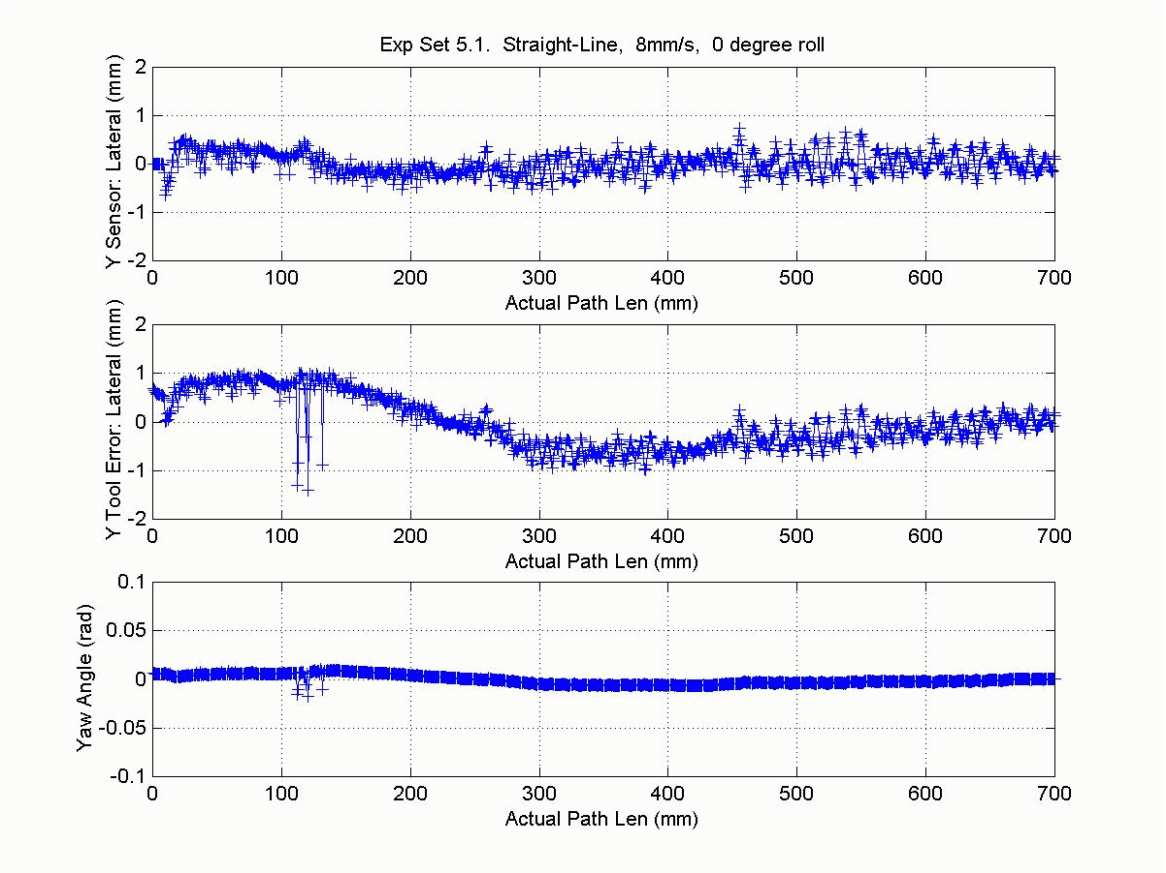


Figure 6.34: Lateral/Yaw axis data for Alternate Wrist Setup Experiment

The data for the depth is dramatically smoother than for the lateral case. The lead sensor remains at zero indicating excellent compensation for the Pitch axis. Once again correction for depth begins after 100mm of seam travel and seam tracking begins. The tool depth fluctuates slightly as the sensor moves farther along the seam. The calculated Pitch angle exhibits a slight rocking which increases in amplitude after 400mm.

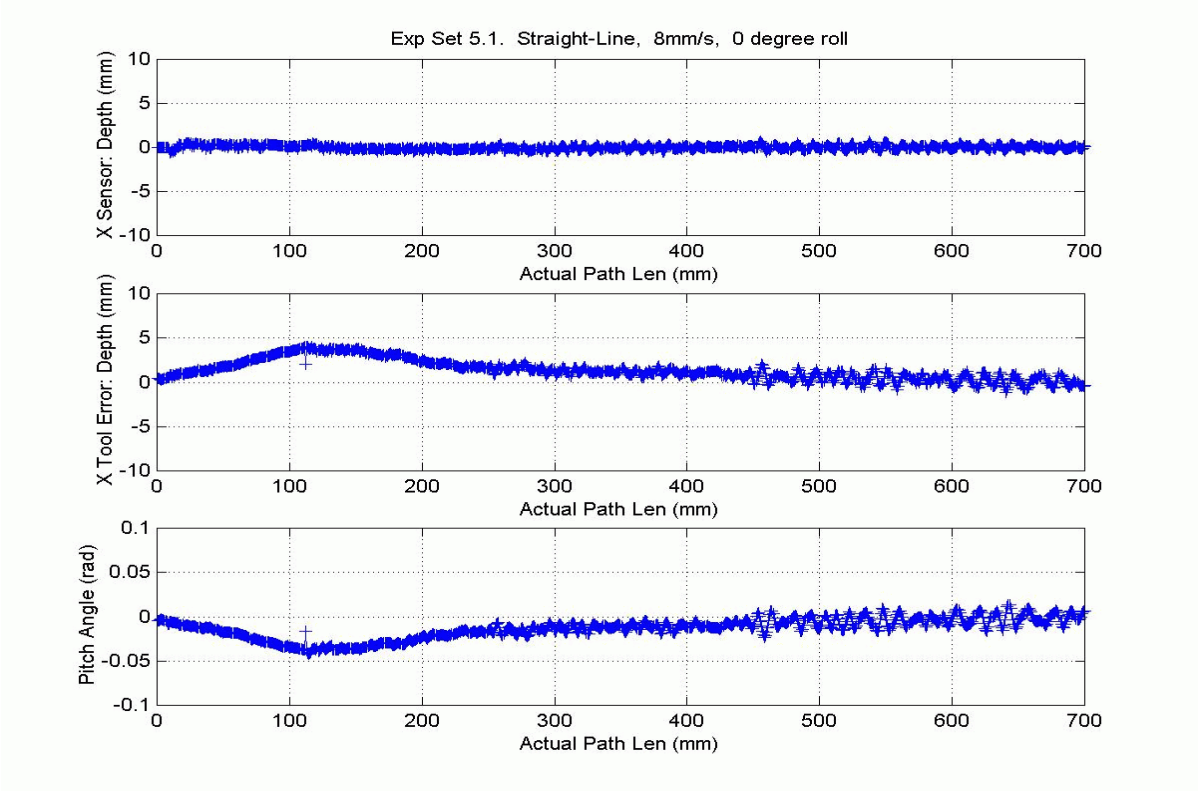


Figure 6.35: Depth/Pitch axis data for Alternate Wrist Setup Experiment

This configuration demonstrates some promising features such as extended range and excellent sensor compensation in both axis. The oscillations at the latter part of seam tracking must still be considered. In general, this sensor configuration, appears to exhibit better performance than the earlier configuration. Hence, future experimentation should be conducted to determine the optimum controller gains for this configuration.

Chapter 7.

Discussion and Recommendations

This thesis described the conception, design and preliminary trials of a distributed multiple robotic sensor-based system. This system called DROID which stands for a Distributed Robot of Intelligent Devices was designed to be an open architecture system capable of adapting to any given task. Based on conventional Personal Computer technology it makes extensive use of inexpensive high performance computing power and high bandwidth interconnect technologies. Coupled with a high performance real-time operating system and a native message passing interface, very powerful software based robot controllers and intelligent sensor can be built up and integrated together with relative ease.

To evaluate the DROID architecture, the task of Autonomous Multiple Robotic Welding was implemented. This application requires two robotic arm manipulators and a laser profiling sensor to be integrated into the distributed architecture and made to work together. The previous section reported the performance data collected from various seam-tracking trials. This chapter discusses the success of the DROID system in this application as well as some shortcomings. We conclude with recommendations for future possible work.

7.1. Discussion

Developing the DROID system for Autonomous Multiple Robotic Welding demonstrated the power of a modular and distributed architecture. Each robotic arm was considered a separate entity with a known interface, thus independent development of motion control interfaces could be built for each robot element. Although the Reis Robotstar V15 and the GM Fanuc S400 robots are completely different robots, the modular open architecture allows the Seam Tracking program to communicate with each robot in a simple and identical manner. In addition, sensors such as the profiling sensor and the joystick controller could be used on any of the two robots through the standard interface.

To connect each robot to the central intelligence computer (Brain Computer), standard 10Mbps Ethernet was used. Although the bandwidth is mediocre by today's standards it provides adequate transmission speeds and communication delays between robotic nodes are minimal, thanks to the Native Message Passing protocol of the QNX realtime operating system. This Ethernet connectivity also allows Brain Computers to be developed/programmed/trained off-line using simulation facilities and then integrated into the actual network with minimal down-time.

The Brain Computer also acts as a TCP/IP to QNX Native Message Passing Router such that it can be connected to the Internet while also allowing access to the QNX private network of robot and sensor computers. This capability allows remote access from any point in the world using TELNET, FTP, HTTP or Berkeley Socket Interfaces. Maintenance and supervisory control can be achieved remotely making this system ideal for remote and dangerous environments. The Brain Computer also has the ability to act as a development host environment, thus remote development teams can develop, debug, monitor and deliver code off-site thereby reducing expensive down-time and on-site integration.

Development for the DROID system primarily consists of building interfaces to existing robot manipulators and sensors. In theory, any robot or sensor could be integrated into DROID as long data streams can be interfaced to a PC computer. Most of the existing code base is developed using a high-level language such as C, the code is modular, reusable and easily understood. Thus significant development time is saved. Also, by using a UNIX-style operating

system such as QNX 4.25 programs are written with command line arguments and are stand-alone such that they can be incorporated into UNIX style scripts. Thus an extendable robot language can be developed based on these small programs. By using a multi-tasking operating system, very sophisticated monitoring systems can be built to aid in debugging these programs. Lastly, the operating system allows the programmer extensive access to low-level hardware, thereby allowing device driver to be written quickly. All the above aspects facilitate software development on the DROID system and set it apart from conventional embedded programming.

From the experimental data collected, it is evident that the tasks such as seam tracking can be implemented correctly on this multiple robotic platform. With such a centralized data system development of a task program can be very quick and straight-forward. In the case of the seam-tracking program, we were able to focus our development on seam tracking without having to worry about the underlying hardware.

It was determined that seam-tracking, since it is cartesian kinematic problem is significantly affected by the joint configuration of the robot. Depending on the location of the robots, differing performance levels resulted. Performance was best when the robot's wrist was confined in a given partition with minimal cross-over into an adjacent partition and if the robot did not have to "reach out". Over-extension of the robot in its workspace sphere leads to instabilities and rocking motion of the wrist configuration which was observed in the latter part of every seam tracking trial. Hence, the workpiece holder robot must keep the seam patch within the confines of a partition and well within the interior of the tool robot's reach for optimal seam tracking performance. In addition, alternative sensor configurations with respect to the wrist such as that in Experiment 5 prove to have better performance in terms of extended range and accuracy. Investigation into optimum mounting angles and controller gains should be pursued in future studies.

Based on the experimental results, it can be concluded that proportional gains for the seam-tracking system are affected by the rate of turn (radius), the travel speed, and the location/configuration of the wrist itself. This leads us to conclude a multi-variable controller is required. This controller, could take the form of a neural network or fuzzy logic controller.

Since seam-tracking is very dependent on joint configuration, it seams that a seam-tracking controller which controls the joints directly as opposed to making cartesian correction would be

more adequate for a 6 revolute jointed robot. This would make the system independent from the inverse kinematics of reaching the point. This coupled with a learning-based neural network/fuzzy logic system would lead to an optimal seam-tracking controller.

The extensive experimentation demonstrates that a rich set of data collected as well a vast set of circumstances can be simulated with the DROID system. Our preliminary experiments with transport delays shows the DROID system future depends on the ability to handle delays in data delivery. Currently, delays of 200ms can be handled fairly well with minimal adverse affect. However, this delay could be extended if data integrity programming is implemented.

7.2. Limitations of Design

Although, the DROID system improves and eliminates many of the problems associated with multiple robotic control systems it does have certain limitations. One of the difficulties experienced during developing the DROID system for Autonomous Multiple Robotic Welding was integrating existing robot controllers and sensors into this environment. Much of the responsibility for developing the DROID interface computer of each subunit rests on the system integrator. Fortunately, robots such as the Reis Robotstar V15 have a open bus architecture which allowed a fairly simple upgrade path to integrate into the DROID system. The GM Fanuc S400 robot however is extremely proprietary, thus only a partial motion control interface was implemented and somewhat limits the DROID system. Unfortunately, this is fairly widespread and the Reis example is more the exception than the rule.

QNX hardware support is rather limited as well. For example, Ethernet cards were selected not on performance issues but for compatibility. As well, many of the expansion cards such as data acquisition and video frame-grabbers which have available Microsoft Windows Operating System device drivers do not have such drivers for QNX. Once again, the responsibility of integrating expansion cards lies with the system integrator. This may require significant programming effort depending on the level of board information obtained form the manufacturer. In summary, the limitations of the DROID system itself is largely due to the proprietary nature of

the computer and robot manufacturers.

7.3. Recommendations

The DROID system is an ideal development system for multiple robotics. One of the intentions of this thesis was to develop an experimental robotic work-cell to conduct research in sensor-based research for multiple robots. The Autonomous Multiple Robotic Seam Tracking trials conducted represent a preliminary experiment to realize the DROID system concept of multi-robot and sensor coordination. The experience that we have gained opens the door to endless possible tasks which could be tackled with the DROID system. In order perform this type of experimentation or tasks, more sensors and robots must be integrated into the DROID system. Such integration involves the writing of device drivers and upgrading robot controllers to communicate with the Brain Computer via Ethernet. Sensors to integrate include force/torque transducers, video cameras and frame-grabbers, and ultrasonic transducers. In addition to hardware device drivers being developed, more sophisticated software controllers could be written for the DROID system robots such as neural network or fuzzy logic servo control modules, force/torque control, collision avoidance algorithms, visual servo control, and non-linear control. These modules could be written and easily integrated into the DROID Framework due to its modular concept which was designed primarily to facilitates the experimentation of different control paradigms and schemes on the DROID system.

Communication can also be improved by upgrading the Ethernet backbone to 100Mb/s or 1Gb/s using fast Ethernet adapters and Ethernet switches. This would minimize latencies inherent in the system as well as allowing richer data streams to be delivered between subunits and the "brain". This would extend the DROID system to be used in more sophisticated applications which use a large array of sensors such as sensor-driven autonomous robotic work-cells and mobile autonomous robotic applications. These applications would require better remote user interfaces for tele-operation and the development of simulation and visualization tools.

As computing power increases, the DROID system should be upgraded to advantage of the tremendous performance increases. The DROID system architecture makes it possible to constantly maintain the system at the cutting edge. With this in mind the only limits for the DROID system lies in the imagination of the system integrator.

References

[1] Yasuda, G., Tachibana, K., " Computer network based control architecture for autonomous distributed multirobot systems", *Computers & Industrial Engineering,* vol 31 no 3-4, p 697-702, Pergamon Press Inc., 1996.

[2] Ng, L., Huissoon, J. P., "A Distributed Multi-Robotic Environment for Flexible Manufacturing", *Proceeding of the 31st International Symposium on Robotics(ISR 2000)*, p 194-5, 2000.

[3] Kriegman, D. J., Triendl, E., Binford, T. O., " A mobile robot: sensing, planning and locomotion". *Proceedings of the 1987 IEEE International Conference on Robotics and Automation*, vol.1, p.402-8., IEEE Comput. Soc. Press, 1987.

[4] Huissoon, J. P., Strauss, D. L., "Automatic Control of a Robotic Gas Metal Arc Welding Cell", *Conference Proceedings of Fifth World Conference on Robotics Research*, p 16-1 to 16-11, 1994.

[5] Strauss, D. L. "Real-time Seam Tracking and Torch Control for a Welding Robot". M.A.Sc. Thesis, University of Waterloo, 1991.

[6] Stefanuk, W. "A Multiprocessor Based Robot Cntroller: Software Design, Force Sensor

Integration, and External Force Estimation". M.A.Sc. Thesis, University of Waterloo, 1988.

[7] Singh, S., Simmons, R., Smith, T., Stentz, A., Verma, V., Yahja, A. , and Schwehr, K., "Recent Progress in Local and Global Traversability for Planetary Rovers", *Proceedings of the IEEE International Conference on Robotics and Automation*, IEEE, 2000.

[8] Stewart, D. B., Schmitz, D. E., Khosla, P. K.," The Chimera II real-time operating system for advanced sensor-based robotic applications", *IEEE Transactions on System, man, and Cybernetics*, vol 22 no 6, p 1282-1295, IEEE , 1992.

[9] Gertz, M. W., Stewart, D. B., Khosla, P. K.," Human-machine interface for distributed virtual laboratories", *IEEE Robotics & Automation Magazine*, vol 1 no 4, p 5-13, IEEE , 1994.

[10] Yasuda, G., " Object-oriented multitasking control environment for multirobot system programming and execution with 3D graphic simulation", *International Journal of Production Economics*, vol 60, p 241-250, Elsevier Science, 1999.

[11] Villarroel, J. L., Silva, M. and Muro-Medrano, P. R., "Petri nets in a knowledge representation schema for the coordination of plant elements", *Preprints of IFAC/IMEKO Symposium on Intelligent Components and Instruments of Control Applications*, p 341-346, 1992.

[12] Alami, R., Fleury, S., Herrb, M., Ingrand, F., Robert, F., "Multi-robot cooperation in the MARTHA", *IEEE Robotics & Automation Magazine*, vol 5 no 1, p 36-47, IEEE, 1998.

[13] Jung, D., Zelinsky, A., "Architecture for distributed cooperative planning in a behaviour-based multi-robot system", *Robotics and Autonomous Systems*, vol 26 no 2-3,

p 149-174, Elsevier Science B.V., 1999.

[14] Parker, L. E, "ALLIANCE: An architecture for fault tolerant multirobot cooperation", *IEEE Transactions on Robotics and Automation*, vol 14 no 2, p 220-240, IEEE, 1998.

[15] Norberto, J. and Sada Costa, J. M. G., "Object-oriented and distributed approach for programming robotic manufacturing cells", *Robotics and Computer-Integrated Manufacturing*, vol 16 no 1, p 29-42, Elsevier Science Ltd, 2000.

[16] Rubini, A., *Linux Device Drivers*, O'Reilly & Associates, Inc., 1998.

[17] Shanley, T., Anderson, D., *PCI System Architecture Third Ed.*, Addison-Wesley Publishing Company, 1995.

[18] Tundra Semiconductor Company. *VMEbus Interface Components Manual, Universe II*, Tundra Semiconductor Company, Kanata, Canada, 1998.

[19] Wurll, C., Henrich, D., Woern, H., Schloen, J., Damm, M., Meier, W., " Distributed planning and control system for industrial robots", *International Workshop on Advanced Motion Control*, p 487-492, IEEE, 1998.

[20] Cavalieri, S., Di Stefano, A., Mirabella, O., "Impact of fieldbus on communication in robotic systems", *IEEE Transactions on Robotics and Automation*, vol 13 no 1, p 30-48, IEEE, 1997.

[21] QNX Software Systems Ltd., *Real-Time Programming under QNX 4: Course Notes*, QNX Software Systems Ltd. Kanata, Canada. 1999.

[22] Reis Machines Inc., *Reis Robotstar V15 User's Manual*, Reis Machines Inc., Straussburg, Germany, 1985.

[23] Xycom Automation Inc., *XVME-655 User's Manual*, Xycom Automation Inc., Saline, Michigan, USA, 1998.

[24] Fanuc Robotics Inc., *Fanuc S-400 Maintenance Manual*, Fanuc Robotics Inc., Ann Arbor, Michigan, USA 1985.

[25] Modular Vision Systems Inc., *LaserVision Basic System User's Documentation*, Modular Vision System Inc., 3195 De Miniac Montreal, Canada, December, 1988.

[26] Bollinger, J. G., Duffie, N. A., *Computer Control of Machines and Processes*, Addison Wesley Publishing Company, 1998.

[27] Craig, J. J., *Introduction to Robotics Mechanics and Control Second Ed.,* Addison-Wesley Publishing Company, 1989.

[28] McKerrow, P. J., *Introduction to Robotics*, Addison-Wesley Publishing Company, 1991.

[29] Levy, S., *Artificial Life: A Report from the Frontier Where Computers Meet Biology*, Vintage Books, 1992.

[30] Crymble, D. J. In Print. M.A.Sc. Thesis, University of Waterloo (In Progress)