AUTONOMOUS MOBILE MATERIALS HANDLING PLATFORM ARCHITECTURE FOR MASS CUSTOMISATION

UNIVERSITY OF
KWAZULU-NATAL

Anthony John Walker
School of Mechanical Engineering
University of KwaZulu-Natal

Submitted in fulfillment of the academic requirements for the degree of
Master of Science in Engineering

November 2008

(a) 947825

HC 09/03360

As the candidate's Supervisor I agree/do not agree to the submission of this dissertation:

Signed: ______________________

Professor Glen Bright, 23 December 2008

T
621.86
WAL

Declaration

The author hereby declares that he has produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been submitted in identical or similar form to any other University examination board.

The work was completed by the author at the School of Mechanical Engineering, University of KwaZulu-Natal from January 2007 to November 2008.

Signed: ______________________
A.J. Walker

First and foremost I dedicate this thesis to my loving parents, Cheryl and Raymond, who have provided me with the utmost guidance and support throughout my life. To my sister, Caryn, for always looking out for my best interests and sustaining a good friendship.

Finally, to Klaus, William and Jessica, who, although represent the family pets, contribute to my health.

Abstract

In order to facilitate the materials handling requirements of production structures configured for Mass Customisation Manufacturing, the design of requisite materials handling and routing systems must encompass new conceptual properties. Materials handling and routing systems with the capacity to support higher-level management systems would allow for mediated task allocation and structured vertical integration of these systems into existing manufacturing execution and management systems. Thus, a global objective in designing a materials handling and routing system, for such production configurations, is to provide a flexible system mechanism with minimal policy on system usage.

With the recent developments in mobile robot technologies, due to various advancements in embedded system, computational, and communication infrastructures, mobile robot platforms can be developed that are robust and reliable, with operating structures incorporating bounded autonomy. With the addition of materials handling hardware, autonomous agent architectures, structured communication protocols and robotic software systems, these mobile robot platforms can provide viable solution mechanisms in realising real-time flexible materials handling in production environments facilitating Mass Customisation Manufacturing.

This dissertation covers the research and development of a materials handling and routing system implementation architecture, for production environments facilitating Mass Customisation Manufacturing. The materials handling and routing task environment in such production structures is characterised in order to provide a well defined problem space for research purposes. A physical instance of a functional subset of the architecture is constructed consisting of a semi-autonomous mobile robot platform equipped with the infrastructure for materials handling and routing task execution. The architecture orientates the mobile robot platform in such a way as to present a collection of functional units, integrated and configured for a range of applications, and prevents viewpoints in the sense of monolithic mobile robots less susceptible to reconfiguration and stochastic utilisation.

Acknowledgements

First and foremost I would like to acknowledge my supervisor, Professor Glen Bright, for providing me with the suitable environment and guidance to further my education in Engineering, and allow me to develop as a researcher.

I would also like to acknowledge my fellow research colleagues, Jared, Louwrens, Riaan and Shaniel, for putting up with my long-winded discussions about modern engineering practice, production systems, and Open Source software development, in the late afternoons down in the laboratory.

Lastly, I would like to acknowledge the entire Open Source software community for caring enough about the Open Source movement to take time out of their lives to endure harsh debugging sessions and follow standardised coding practises to release source code in the open free market.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

''Great thoughts reduced to practice become great acts'' - William Hazlitt

This dissertation covers research in the area of advanced manufacturing and specifically focuses on the development of mobile robot systems to facilitate the materials handling and routing tasks required for production rate sustainability under customer-induced variations in production requirements in Mass Customisation Manufacturing (MCM) implementation.

1.1 Brief Problem Space - Solution Space Overview

The problem addressed lies in facilitating a specific form of materials handling and routing task associated with customer-induced variations in production requirements. In the context of MCM, these variations in production requirements are caused through deterministic changes in product design, which can produce variations in work flow through a production plant when processing requirements fall outside the capability scope of manufacturing infrastructure subsets associated with standard work flow and routing.

Characterisation of the problem space associated with this research is achieved by exposing the concepts and properties of MCM production structures. This characterisation leads to the development of a functional definition for a materials handling and routing task that facilitates customer-induced work-flow $\leftrightarrow$ payload routing variations.

As manufacturing systems have technical properties and characteristics that are product and process technology specific, the development of absolute physical systems, as solution mechanisms, does not provide a credible research output. Therefore, an effort is made to focus on generic requirements and develop solution structures on the basis of system scalability and extensibility, allowing for integration of application specifics to facilitate unique production plants.

Exposure of the properties and characteristics of flexible materials handling and routing tasks develops into a materials handling and routing system implementation architecture that provides a generic solution structure incorporating the core functionality required in order to facilitate the materials handling and routing tasks of customer-induced variations in production requirements. A physical prototype is developed based on a subset of the functional units of the implementation architecture in the form of a semi-autonomous mobile robot platform.

Materials handling and routing of payloads between distributed manufacturing infrastructure subsets through the utilisation of mobile devices, such as mobile robots, requires research and development of motion controllers for such robot devices in order to execute materials handling and routing tasks. Due to this requirement, the majority of the research output is based on the motion control problem. Full state feedback stabilisation of differential drive mobile robot platforms is quantified along with the development of Lyapunov stable[1] motion

[1]Lyapunov stability is introduced in section 5.2.1.2

controllers. Control solutions take the form of asymptotically stable nonlinear, and piecewise continuous hybrid control laws.

1.2 Research Project Objectives

1. To research materials handling and routing for Mass Customisation Manufacturing (MCM), in order to expose, characterise, and define the routing tasks associated with its production structure in order to provide a well defined problem space for research.
2. To develop a layered implementation architecture that encapsulates the core functionality required for materials handling and routing task execution.
3. To research, design and construct a mobile materials handling and routing robot platform, in alignment with the layered implementation architecture, to provide a test-bed for experimentation and performance testing purposes.
4. To utilise the mobile materials handling and routing robot platform to test and validate the motion control aspects associated with facilitating the material routing requirements imposed through MCM production structures.

1.3 Dissertation Overview

The main topics in each chapter are listed and briefly described below in order to provide an overview of this dissertation as a whole.

Chapter 2
Preliminary concepts and characteristics pertaining to Mass Customisation, as a niche market facilitator, are presented. These preliminary concepts are required in order to understand the unique production environment created through MCM implementation. An overview of the plant layout aspects that affect materials handling requirements in MCM production is presented in order to provide insight into passive measures that can minimise materials handling and routing requirements. The chapter ends by defining a Flexible Material Routing Primitive (FMRP), and presents a layered implementation architecture designed to facilitate the execution of such.

Chapter 3
The concept of Mechatronics is presented in order to highlight the design method used in developing the mobile materials handling and routing robot platform prototype, which provides a test-bed for experimentation purposes. The mobile robot platform's design history is presented including relevant theory and pivotal design aspects in its implementation.

Chapter 4
The design and operating structure of the Open Source software system used in implementing Hardware Abstraction Layer (HAL) functionality for the mobile materials handling robot platform is presented. The software system provides software scalability and allows for code re-use by abstracting the hardware specifics of the mobile materials handling and routing platform behind well defined generic software abstractions over an IP network, through the use of interface specifications.

Chapter 5
The motion control algorithms implemented in order to facilitate the motion control aspects associated with material payload transportation between distributed manufacturing infrastructure subsets is presented. Multiple motion control algorithms are implemented and tested on the mobile materials handling and routing robot platform prototype in order to quantify the motion control performance provided by each algorithm.

Chapter 6
A summary of the outputs achieved during the research project provide insight into future research in advanced manufacturing systems and MCM. A discussion on the importance of an international development community in the realisation of next generation materials handling systems for Mass Customisation Manufacturing provides preliminary insight into task force development. Common Model Development (CMD) is discussed with reference to the layered implementation architecture.

1.4 Chapter Summary

The layout and main content of this dissertation has been presented in order to provide the reader with a well directed outlook on the work covered.

Chapter 2

Constructing the Autonomous Material Transportation Specification

''To make contributions of this kind, the engineer requires the imagination to visualise the needs of society and to appreciate what is possible, as well as the technological and broad social-age understanding to bring his vision to reality'' - Sir Eric Ashby

The aim of this chapter is to expose, characterise, and define a "Flexible Material Routing Primitive" (FMRP), a task that has been conceptualised to represent a specific form of materials handling and routing operation, associated with customer-induced variations in production requirements.

From the characteristics of a FMRP, an implementation architecture is introduced that encapsulates the core functionality required to facilitate the execution of such materials handling and routing tasks. This implementation architecture forms the major component in the development of a mobile materials handling and routing robot platform prototype in chapter 3.

The concepts and characteristics exposed in this chapter act as precursors to the introduction of design and performance specifications for the mobile materials handling and routing robot platform developed during this research project. These specifications are presented in section 2.4.

2.1 Mass Customisation and DFMC

It is important to understand and quantify how customers, in the context of MCM, create variations in production requirements, in order to develop manufacturing structures that are capable of economically operating under such conditions. In order to understand MCM, one must first understand the notions of Mass Customisation and Design for Mass Customisation (DFMC).

2.1.1 Mass Customisation - A Brief History and Definition

The concept of Mass Customisation first appeared in the book *"Future Shock"*, by Alvin Toffler [45]. Toffler, much appreciated for his propositions and explanations for modern sociological phenomena, described Mass Customisation as a method of catering to niche markets. "Mass Customisation" was formally termed by Stan Davis. In his book *"Future Perfect"*, Davis projected trends encompassing micro-segmentation of consumer markets and unique product development for customers [14]. The definition provided by Frank Piller best describes Mass Customisation fundamentals, and thus, it is the formal definition in this dissertation [36].

Mass Customisation. "**Customer co-design** process of products and services which meet the needs of each individual customer with regard to **certain product features**. All operations are performed within a **fixed solution space**, characterised by **stable but still flexible and responsive** processes. As a result, the costs associated with customisation allow for a price level that does not imply a switch in an upper market segment", [36].

The key aspects, in bold font in the above definition, require reiteration.

- **"Customer co-design"**: Customers, in the context of Mass Customisation, are integral product design elements and implicitly create production dynamics.
- **"certain product features"**: Customisation is limited to certain product features thus ensuring globally bound and deterministic variations in production requirements.
- **"fixed solution space"**: The production environments implementing Mass Customisation have limited, but well defined, capabilities.
- **"stable but still flexible and responsive"**: Production rate volatility is minimised through responsive production operations.

Therefore Mass Customisation, or more specifically MCM, is not only the manufacture of customised products at mass production efficiencies, [25], but even more so, is a production structure developed to economically facilitate bounded, customer-induced variations in production requirements, under near constant production rates.

2.1.2 Design For Mass Customisation

DFMC represents the notion of designing products that are more susceptible to efficient and economical manufacture in a MCM context [27]. DFMC is an extremely complex task and this is only a brief overview of the aspects of DFMC that are capable of affecting production stability.

2.1.2.1 Design For Manufacture (DFM)

Taking processing requirements into consideration when designing products provides smooth transitions into production. Design for fixturing, design for easy fabrication and assembly, design for minimal setup and design for minimal utilisation of cutting tools are all design methods used in DFM. DFM can produce products that require less overall processing which implicitly creates simpler logistics and material flow through the production plant thus reducing materials handling and routing tasks.

2.1.2.2 Modularisation

Modular products allow for an increase in the degree of customisability for a particular product model. Modularity allows for the structured utilisation of manufacturing resources and plays a major role in facilitating and satisfying the psyche of demanding and particular customers. There exists a strong correlation between Mass Customisability of a product with regard to modularity [33][32].

2.1.2.3 Paramaterised CAD/CAM product models

This is a fundamental requirement in DFMC and allows customers to configure their desired product through the use of configurators, which represent software systems with Graphical User Interfaces that allow customers to edit the configuration of a custom product. This not only aids in customer satisfaction but also prevents unnecessary choices from manifesting into unseen production conflicts.

2.1.3 Summary

DFMC can thus be seen as a passive measure of ensuring production rate stability by designing products that "behave well", in terms of developing bounded and controlled variations in production dynamics, under deterministic customer-induced, changes in product design. The successful implementation of Mass Customisation, in the context of facilitating custom product manufacture through MCM, is largely dependent on the design and customisability of the products offered by a particular manufacturing firm. DFMC can be seen as the first line of defense in providing stable production in an environment perturbed by customer-induced changes in product design, and must not be overlooked.

2.2 The MCM Production Structure

In order to implement production structures that are capable of MCM, all elements that affect or control the "magnitude" or "frequency" of customer-induced production rate variations must be collected and concurrently analysed as a whole. Concurrent analysis can allow for insight into methods of quantifying and achieving production stability, recall the **"stable but still flexible and responsive"** aspect of Mass Customisation, section 2.1.

2.2.1 Concurrent Analysis Through Control Theoretic Constructs

There will always be a semantic breakdown regarding the communication of concepts across engineering disciplines. For this reason, the author has encapsulated the concepts of MCM production structures in a control theoretic construct, in order to partially provide semantic homogeneity for this discussion.
Conceptual insight into concurrency, in the design and implementation of MCM production operations, can be realised by encapsulating MCM in a SISO control loop construct, Figure 2.1. In such constructs, customers can be regarded as deterministic input disturbances into production plant that are set up, or optimised, around a set of standard processing requirements. These input disturbances, i.e. customers, effectively produce variations in production requirements and production rate, $y(t)$, due to changes in standard CAD/CAM models and associated production plans, stored in a Standard Product Model Library[1] (SPML).

In this control theoretic model, DFMC is regarded as a pre-filter or feed-forward controller that conditions Company Wide Decisions (CWD), such as target consumer market and product strategy, into more suitable parameterised CAD/CAM product models. This notion reiterates the importance of DFMC in achieving successful MCM at pre-determined economic production

[1]In the context of MCM, and for this representation, a SPML stores a collection of product CAD/CAM models in "standard" configuration. Standard configurations can be thought of a those product configurations that represent best estimates of customer preference, determined through market research.

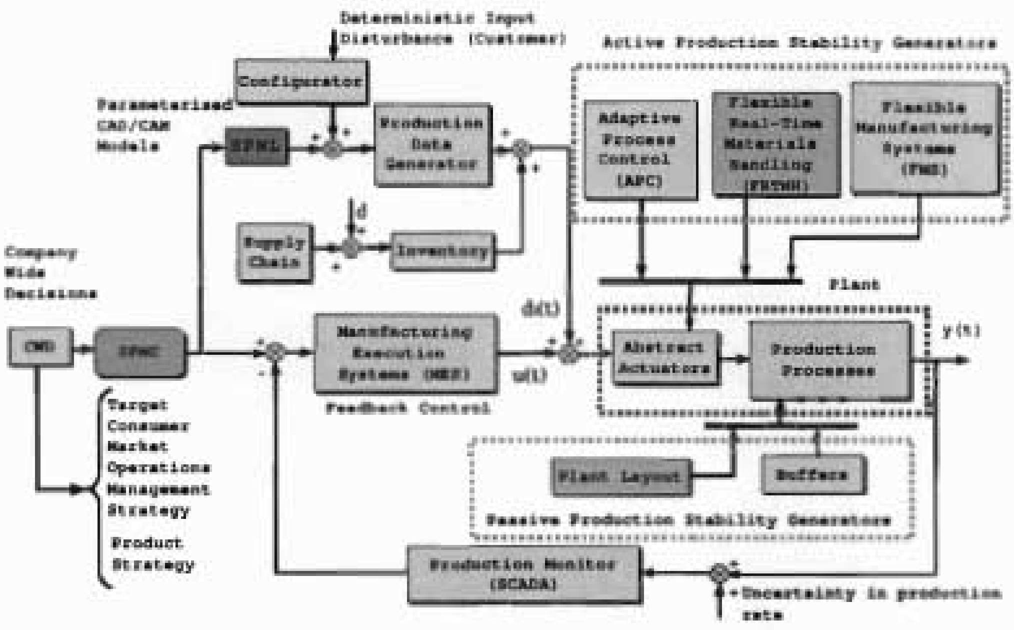


Figure 2.1: Control Theoretic Description of Mass Customisation Manufacturing - In this model, customers act as input disturbances, which offset standard production operations. These input disturbances require regulation by the MCM production controller

rates, section 2.1.2.

Manufacturing infrastructure, such as Flexible Manufacturing Systems (FMS) can be regarded as abstract actuators that are driven by control signals from the production controller, i.e. with reference to Figure 2.1, Manufacturing Execution Systems (MES), to absorb process variations in order to maintain nominal production rates under deterministic input disturbances created through customer-induced changes in standard product design. In the same sense, Flexible Real-Time Materials Handling, (FRTMH), as an "active" manufacturing component, can be envisaged as an abstract actuator that integrates, through distributed materials handling and routing task execution, various FMS processing cells in order to provide required process integration. In terms of this control theoretic construct and production model, a conceptual control problem statement can be constructed as follows.

The MCM Control Problem. Find/select/choose/design and concurrently develop a feed-forward controller (DFMC), a feedback controller (MES), actuators (APC, FRTMH, FMS), sensors (SCADA, QC), and include passive measures such as structured plant layouts and product flow buffers, to produce a production plant that operates so as to regulate deterministic customer-induced input disturbances in the form of process and material routing variations at a pre-determined stable and economic production rate.

Note: With regard to the aspects concerning material routing variations, plant layout, as a "passive" manufacturing component, is equally as important as Flexible Real-Time Materials Handling, as an "active" manufacturing component, in achieving input disturbance regulation and production rate stability. This aspect exposes the multi-dimensional solution spaces of modern manufacturing, and associated materials handling environments. These components are shown as yellow blocks in Figure 2.1. Plant layout and its correlation to materials handling requirements is discussed in section 2.2.2.

2.2.2 Plant Layout and Material Routing Efficiency

Current plant layout structures, such as process, product and cellular layouts are designed for a particular product mix and production volume, which is assumed to remain constant for extended periods of time, between three to five years [3]. These plant layouts are designed utilising metrics that describe long term materials handling efficiency. When production requirements change, through a change in product mix or production volume, a reconfiguration in plant layout is required in order to sustain materials handling efficiency and production rate stability. This is a costly exercise and it is not uncommon for plant managers to prefer to live with the inefficiencies of an existing plant layout structure, rather than perform a costly plant layout reconfiguration [4].

2.2.2.1 Functional Plant Layouts

In modern manufacturing environments, functional layouts, also known as process layouts [19], in which manufacturing infrastructure of similar functional scope are grouped together, are considered the most flexible for high product variety and/or low production volumes, Figure 2.2.

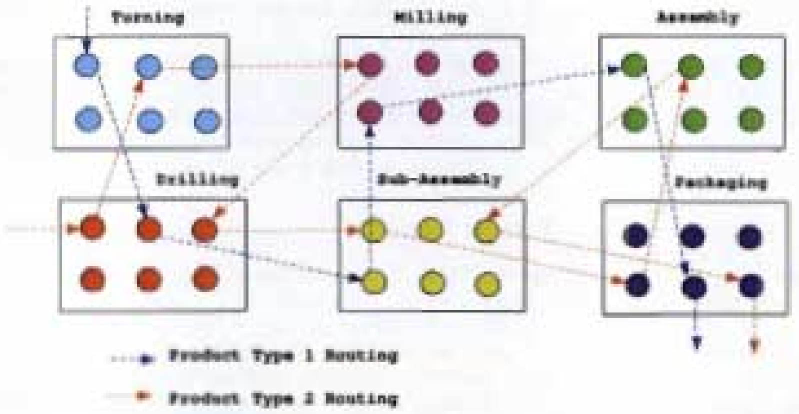


Figure 2.2: Functional Plant Layout Structure - Common colors represent manufacturing infrastructure of the same functional scope

The materials handling efficiency in such plant layout structures is dependent on the spatial distribution of consecutive processing stations that provide the necessary processing primitives required to facilitate the manufacture of a particular product. Functional layouts have poor materials handling and routing efficiency, and can produce inefficient materials handling and routing in cases where a particular product requires the utilisation of consecutive processing primitives that have high spacial distribution. This can result in poor production rates, bottlenecks, and under utilisation of manufacturing resources.

A credible objective in designing a functional layout is to determine a relative configuration of processing work-cells, such as milling work-cells and turning work-cells, so as to minimise the average required materials handling and routing distance for a particular product mix. In practical implementations, layout designs based of minimising functional descriptions incorporating metrics such as Total Materials Handling Distance and Total Adjacency Score are utilised frequently [24].

2.2.2.2 Cellular Plant Layouts

Cellular layout configurations are developed by grouping manufacturing infrastructure into functional cells. Each processing cell is dedicated to the manufacture of a family of products with similar processing requirements. This is also known as a Group Technology layout and is used frequently in batch production operations [19], Figure 2.3.

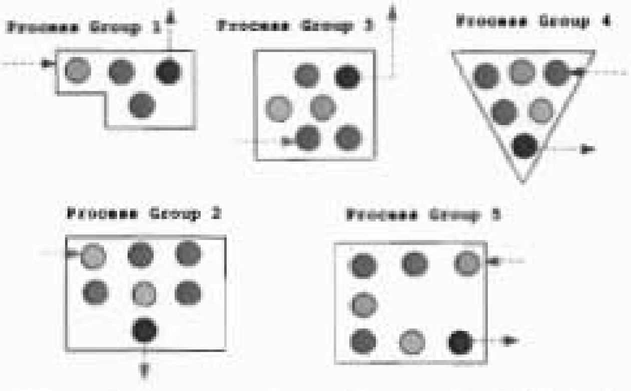


Figure 2.3: Cellular Plant Layout - In modern implementations of cellular layouts, a robotic manipulator is centrally situated in each cell and provides articulated intra-cell materials handling

Cellular plant layouts simplify workflow and reduce the materials handling and routing requirements for a particular product mix, [19]. Once again however, cellular layouts are designed under the assumption that product life cycles are sufficiently long and that the demand for the product is stable. Once established, there is minimal inter-cell materials handling and routing. This aspect makes cellular layouts sensitive to changes in product demand or mix.

2.2.2.3 Conceptual MCM Plant Layouts

There are no universally accepted methods of designing plant layouts, and thus materials handling systems for MCM implementation, although there is much literature on the subject [23][11]. One can suggest, however, that increasing the functional scope of layout structures through responsive and flexible materials handling and routing systems would allow for hybrid plant layouts, incorporating the structures associated with both functional and cellular layouts, to provide robust production rates through flexible process integration in MCM. Through careful DFMC, active manufacturing infrastructure, such as Flexible Manufacturing Systems could be grouped into functional cells that facilitate a subset of the variational aspects of custom product design. Flexible materials handling and routing operations could then integrate these functional subsets, by executing flexible[1] material routing tasks, to facilitate the manufacture of custom products at economic production rates.

It has become common practice in high variety production to use customisable production platforms that incorporate and quantify the particular processing, and materials handling and routing aspects, in the design and manufacture of a custom product [26][18]. These customisable production platforms encapsulate the processing requirements of a base product configuration, which all custom products are based upon through changes in modular component attachment and dimensional variation [20].

By balancing the production aspects associated with base product component manufacture with the production aspects of providing flexible customisability on top of base components, MCM plant layouts can be designed to facilitate high-volume ↔ low routing flexibility, in material flow between processing stations producing base product components, and low volume ↔ high routing flexibility, between processing stations adding customisable elements to base product components.
Figure 2.4 presents a conceptual layout of a MCM production plant based on this balancing structure. The layout is hybrid in the sense that the materials handling system integrates the elements of both functional and cellular layout configurations.

[1]Flexible in the sense that the material payload is not constrained to move along pre-determined paths

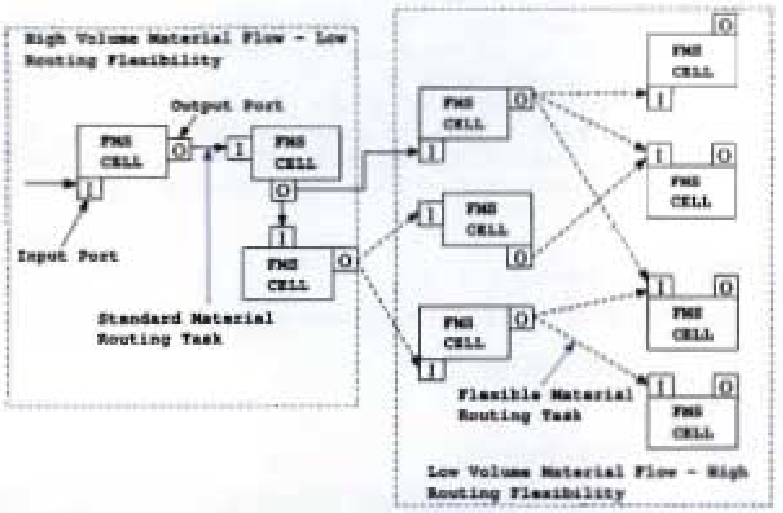

High Volume Material Flow - Low Routing Flexibility
Output Port
Input Port
FMS CELL
I
O
Standard Material Routing Task
Flexible Material Routing Task
Low Volume Material Flow - High Routing Flexibility

Figure 2.4: Conceptual MCM Layout Configuration - Conventional conveyor and gantry systems can facilitate materials handling requirements in FMS processing stations producing base components. Mobile platforms then facilitate flexible material routing tasks between FMS processing stations for custom component integration.

2.2.3 Summary

An important aspect in the implementation of MCM is the concurrent development of production structures that can regulate the negative effects of customer-induced variations in production requirements. These production structures can be designed by concurrently integrating all passive and active manufacturing execution components in order to minimise the negative effects on production rate, either magnitude or frequency, due to customer-induced variations in production requirements, recall Figure 2.1.

The flexibility provided by plant layouts and materials handling systems is vital in providing the required process integration for custom product manufacture. The design of functional and cellular plant layouts however, are based on deterministic paradigms, where product demand and production volumes are known with high degrees of certainty. Metrics, such as Total Material Handling Distance and Total Adjacency Score used in their design, do not aid in providing insight into determining the flexibility and reconfigurability of a particular plant layout. One can suggest that there is a need to develop new metrics that can describe the flexibility of a particular plant layout and materials handling and routing system as a single collective unit. In MCM production operations, over stimulation of plant layout reconfiguration procedures every time new production requirements develop is not an option. The problem of maintaining production efficiency and stability in MCM, now becomes a factor of both plant layout design and materials handling and routing flexibility. Therefore, for MCM implementations, plant layout and the associated materials handling systems design should be regarded as a single problem and treated as a collective unit, i.e. with reference to Figure 2.1, treat both yellow blocks as the same block.

2.3 A Flexible Material Routing Primitive - FMRP

From the previous description, it is apparent that customers in the context of MCM, and for fixed hybrid plant layouts, induce material routing primitives that fall outside of the scope of standard materials handling infrastructure, such as fixed conveyor and gantry systems. When this occurs, flexible infrastructure must be commissioned to execute off-standard, or "flexible", material transportation tasks. Flexible materials handling and routing infrastructure, such as mobile materials handling robots, have an upper bound on the material flow volume that can be facilitated by their operating structure. This is based on exposing the limits associated with performing point to point payload transportation's under single payload instance, and limited physical capacity and motion capability. In this regard, conveyor systems are more suited to high volume material payload transfer in that the payloads can be distributed along the entire conveyor system. Therefore, these flexible material routing systems should cover the material routing aspects associated with product variety only, where the required "utilisation frequency"[1] is relatively low in comparison to that being facilitated by conveyor systems, recall Figure 2.4.

2.3.1 FMRP Characterisation

All materials handling and routing tasks can be decomposed into three basic phases.

1. Material Loading or Pickup phase.
2. Transportation phase.
3. Material Off-load or Set-down phase.

In terms of a FMRP, both the material loading, and off-loading phases are critical in providing robust material transfer between manufacturing infrastructure and materials handling devices. In this sense they can be treated as an equivalent materials handling task. The transportation phase does not explicitly concern the handling of materials, but rather, the gross movement of material between distributed manufacturing infrastructure subsets.

A graphical representation of a FMRP including the various materials handling and routing phases is shown in Figure 2.5.

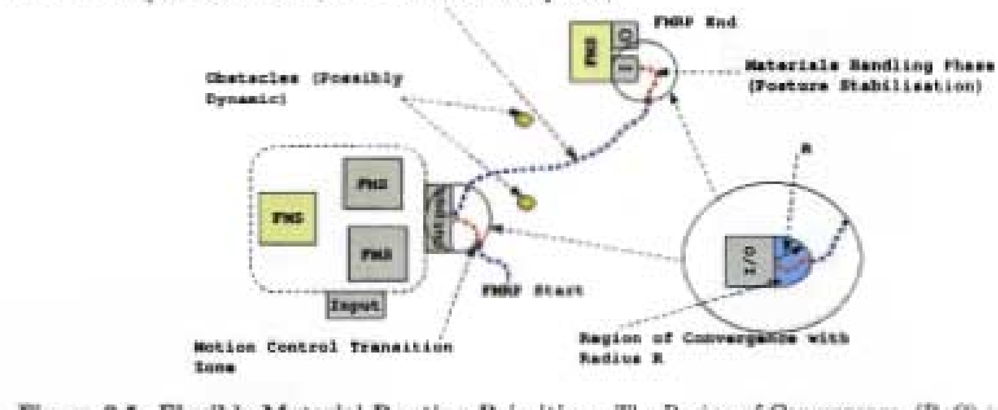

Material Transportation Phase (Local and Global Navigation)
FMRP End
Materials Handling Phase (Posture Stabilisation)
Obstacles (Possibly Dynamic)
R
FMS
FMS
FMS
FMS
Input
I/O
FMRP Start
Motion Control Transition Zone
Region of Convergence with Radius R

Figure 2.5: Flexible Material Routing Primitive - The Region of Convergence, (RoC) is a conceptual space, established to incorporate higher-level mutually exclusive access specifications to input/output port infrastructure via manufacturing management frameworks. This is described in section 2.3.1.2

2.3.1.1 Crucial Aspects in Material Loading and Off-Loading Phases

During these phases, transfer of a material payload is taking place, therefore the relative position and orientation of the materials handling hardware and manufacturing infrastructure subset[2] is critical to ensure successful transfer without damaging the material payload and/or materials handling hardware. During this phase the materials handling hardware should configure its alignment with the manufacturing infrastructure subset, absolutely, to ensure that

[1]This could also be considered as a task execution frequency
[2]Such as the storage buffer conveyor into a FMS cell

position and orientation errors inherited by non-ideal transportation hardware does not destroy the successful execution of the material payload transfer operation. For this purpose, the materials handling hardware must be able to move and align itself relative to the underlying transportation hardware.

Situated around each input/output port in Figure 2.5 is a restricted Region of Convergence, (RoC). The region is restricted in the sense that only one mobile materials handling device may occupy the region, to gain access to the input/output port of the manufacturing infrastructure, at any particular time instant. This has been conceptualised in order to incorporate mutually exclusive access rights, provided by higher-level management frameworks [47], into a FMRP definition. The border of the region represents a transition zone for the "type" of motion control required during material payload transportation. Inside the RoC, motion control is in the form of posture stabilisation. This is a critical aspect in which the mobile materials handling and routing robot platform aligns itself, i.e. achieves a required position and orientation $[x_p, y_p, \theta_p]^T$, with the input/output port in such a way as to ensure successful and robust material payload transfer. The access rights associated with the RoC will assure that posture stabilisation motion control can disregard the explicit requirement of obstacle avoidance as no other robot platforms will occupy the region.

2.3.1.2 Crucial Aspects In the Transportation Phase

The transportation phase is concerned with the routing of a material payload between two distributed manufacturing infrastructure subsets, such as two FMS cells. This phase could potentially require large transportation distances with arbitrary start and end locations.

Outside the RoC, the motion control must be in the form of global and local navigation with explicit real-time obstacle avoidance. The global navigation is required in order to establish path planning between the input/output ports of distributed manufacturing infrastructure subsets. The local navigation is required to establish awareness and perception in the dynamic environment of advanced MCM production plants.

These notions of Regions of Convergence and the associated motion control primitives are analogous to satellite attitude control systems where the control infrastructure is distributed and performs a different function based on its accuracy and sensitivity. For example, in the attitude adjustment of a satellites communications equipment with a receiver on earth, thrusters are used for large attitude adjustments and smaller more accurate magnetic torque generators for final alignment of the communications infrastructure with the receiver on earth.

2.3.2 FMRP Definition

A FMRP is best described by composing two functionally different motion primitives, Figure 2.6.

2.3.2.1 Materials Handling Primitive

With reference to Figure 2.6, a materials handling primitive consists of the following operations.

1. Posture stabilisation from a pose[1] on the boundary of a RoC, $[x_r, y_r, \theta_r]^T$, onto a goal pose, $[x_p, y_p, \theta_p]^T$, in a pre-determined vicinity of an input/output port.
2. An absolute alignment of the materials handling infrastructure with the input/output port using degrees of freedom above those of the underlying transportation device, recall section 2.3.1.1.
3. A material payload transfer task, either loading or off-loading.

[1]Pose and configuration are analogous

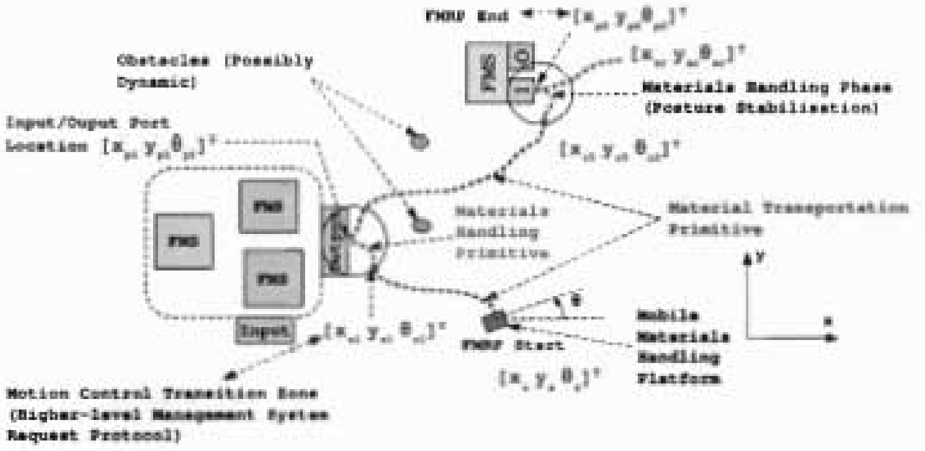

FMRP End
Obstacles (Possibly Dynamic)
Materials Handling Phase (Posture Stabilisation)
Input/Ouput Port Location
FMS
FMS
FMS
Input
Materials Handling Primitive
Material Transportation Primitive
y
x
θ
Mobile Materials Handling Platform
FMRP Start
Motion Control Transition Zone (Higher-level Management System Request Protocol)

Figure 2.6: Flexible Material Routing Primitive Task Instance - Posture stabilisation, represented by the red dashed line in the above figure, of differential drive platforms is a difficult control problem and is covered explicitly in chapter 5

2.3.2.2 Material Transportation Primitive

With reference to Figure 2.6, a material transportation or routing task consists of the following operation.

1. A global navigation operation from a location, $[x_1, y_1]^T$, to a second location $[x_2, y_2]^T$, while avoiding obstacles in real-time.

2.3.2.3 Flexible Material Routing Primitive

A FMRP is defined as follows.

1. A materials handling and routing task assignment from a higher-level manufacturing management system to a mobile materials handling and routing robot platform.
2. A Material Transportation Primitive (NULL)[1] with $[x_1, y_1]^T = [x_s, y_s]^T$ and $[x_2, y_2]^T = [x_{r1}, y_{r1}]^T$, a boundary point of a RoC.
3. A query to a higher-level management framework to request access to an input/output port followed by an outcome acknowledgment.
4. A Materials Handling Primitive (Loading).
5. A Material Transportation Primitive to the off-load RoC with $[x_1, y_1]^T = [x_{p1}, y_{p1}]^T$ and $[x_2, y_2]^T = [x_{r2}, y_{r2}]^T$.
6. A query to a higher-level management framework to request access to an input/output port followed by an outcome acknowledgment.
7. A Materials Handling Primitive (off-loading).
8. A Material Transportation Primitive, (NULL), with $[x_1, y_1]^T = [x_{p2}, y_{p2}]^T$ and $[x_2, y_2]^T = [x_{ar}, y_{ar}]^T$.

[1]NULL specifies that no material payload is present

2.3.3 A Generic Implementation Architecture

A physical materials handling and routing robot platform requires certain basic core capabilities in order to execute a FMRP defined in section 2.3.2.3. In a functional sense, hardware is required to facilitate materials handling as well as provide low-level motion primitives in order to transport a material payload between manufacturing infrastructure subsets. Active sensory infrastructure is required to provide the mobile robot with environmental perception and information on its local working environment. This allows the platform to sense both static and dynamic obstacles in order to perform local navigation and obstacle avoidance during material transportation. Due to the possibly heterogeneous[1] materials handling platforms, executing FMRP's in a production plant, software systems are required that provide HAL functionality[2] in order to provide scalability in control and management structures. Implementing such software systems allows for the development of generic communication standards with homogeneous semantics. A communication subsystem is required to enable the passing of messages, such as FMRP task instances, from higher-level manufacturing management systems to the materials handling and routing device, and allow for task status reporting. At the highest level of abstraction, a materials handling and routing robot platform requires an agent architecture to provide problem solving ability during a FMRP task instance. The agent architecture can also provide facilities to store local task information, such as start and destination location, material payload characteristics and production priority metrics such as due date.

Oven the many years of system development, Engineering architectures have been developed to allow for the structured encapsulation of the concepts and specifications required to implement internationally recognised and scalable systems. Following in this approach, an implementation architecture has been developed to encapsulate the above mentioned core capability requirements for FMRP task execution. The architecture is termed the Autonomous Material Transportation Specification (AMTS), Figure 2.7.

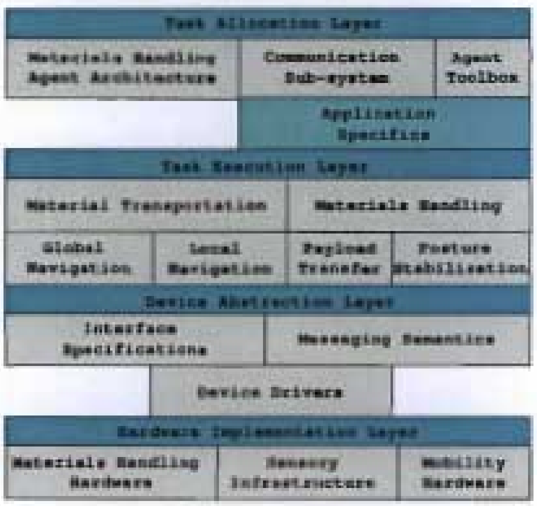


Figure 2.7: Autonomous Material Transportation Specification - The Application Specifics sub-block encapsulates that which is required in order to interface generic task allocations with the product and manufacturing technology specifics of a production plant

The architecture is layered, hierarchial, and consists of four main levels of functional implementation. An overview of each functional layer follows.

[1]In terms of hardware implementation
[2]See section 4.2 in chapter 4

2.3.3.1 Hardware Implementation Layer

The Hardware Implementation Layer (HIL) is a specification on the capabilities of the hardware implementations designed to facilitate the physical requirements of a FMRP. The HIL consists of three sub-blocks that are assigned output specifications on the necessary capabilities for physical FMRP execution. A physical instance of the HIL is covered in chapter 3.

2.3.3.2 Device Abstraction Layer

The Device Abstraction Layer (DAL) is a specification on the structure and implementation of the software systems used to abstract hardware specifics into generic abstractions pertaining to concepts associated with FMRP execution. The DAL has been incorporated into the implementation architecture in order to explicitly include the requirement of scalability in the control structures developed to control physical robot device implementations by allowing higher-level control software to operate in terms of generic device abstractions[1]. The software implementation used during this research project is covered in chapter 4.

2.3.3.3 Task Execution Layer

The Task Execution Layer (TEL) is a specification on the elements that constitute the motion primitives associated with the transportation and materials handling aspects of a FMRP, recall section 2.3.2.3. Specifications are placed on posture stabilisation, and local navigation and obstacle avoidance capabilities. The TEL has access to the HIL through the DAL in order to perform the motion primitives required in order to execute a FMRP. The motion controller implementations developed to facilitate the TEL are covered in chapter 5.

2.3.3.4 Task Allocation Layer

The Task Allocation Layer (TAL) is a specification on the communication infrastructure that allows the materials handling and routing device to accept, interpret and locally manage an assigned FMRP task. The TAL consists of a materials handling agent architecture and toolbox to allow higher-level software systems to gain access to the capabilities provided by the TEL. The TAL facilitates the routing and transmission of task data. To date, this layer of the architecture has not been implemented as research regarding the components, such as the agent architecture and task allocation and scheduling algorithms associated with its implementation are still an open subject in the Mass Customisation Manufacturing research community. Although not explicitly represented in this work, the TAL could place specifications on the status reporting protocols for FMRP task completion or the error codes thrown in FMRP task instance failure.

The mobile materials handling and routing robot platform prototype developed during this research project is based on a subset of the functional components in the AMTS implementation architecture, Figure 2.8.

2.3.4 Summary

A Flexible Material Routing Primitive has been characterised and defined in terms of functionally disjoint materials handling and routing operations. This has allowed for its critical aspects to be encapsulated in an implementation architecture termed the Autonomous Material Transportation Specification, (AMTS).

[1]The DAL is analogous to the Hardware Abstraction Layer (HAL) of an Operating System (OS). Hopefully this analogy makes the functional aspect of the DAL more apparent.

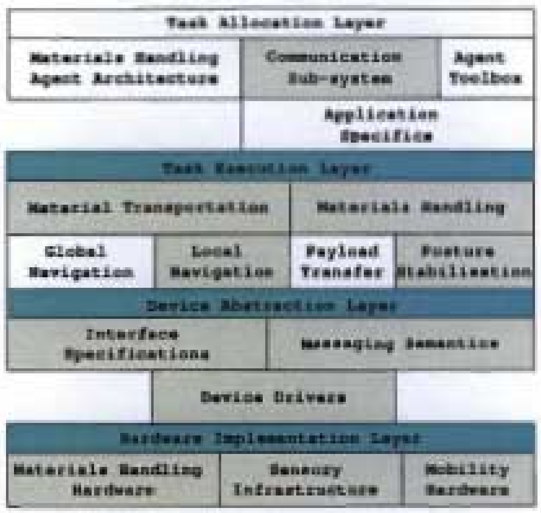


Figure 2.8: Components of the AMTS Implemented in this Work - Although not in its full capacity, the Communication Sub-system sub-block was implemented during this research project and is covered in section 4.1.1.3

2.4 Project Specifications

By characterising a FMRP, specifications on physical implementations to facilitate the execution of such can now be introduced.

In order to establish research bounds and design goals, specifications were placed on functional as well as performance aspects of the mobile materials handling and routing robot platform developed during this research project.

2.4.1 Dimensional and Dynamic Specifications

1. The maximum height of the mobile robot platform, including the materials handling infrastructure, should be 800 mm to interface with the conveyors used by the Computer Integrated Manufacturing (CIM) cell in the Mechatronics and Robotics laboratory, Figure 2.9

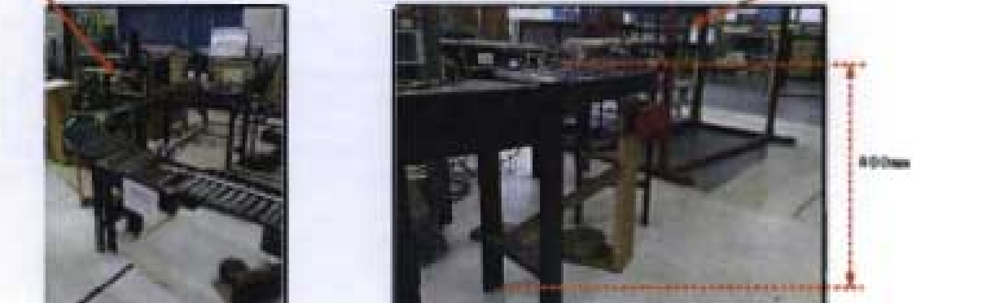

Gantry Based Material Transfer Device
Automatic Storage and Retrieval System (ASRS)
800mm

Figure 2.9: CIM Cell Conveyor System - The gantry based transfer device replicates the infrastructure associated with Flexible Manufacturing Systems

2. Width and breadth of the platform should both be less than 650 mm to allow the platform to navigate through doorways in the laboratory.

3. To allow for feasible application of first order kinematic models during the development of motion controllers for the mobile robot platform, rise times for any step input in velocity around nominal operating conditions should be under a second.

2.4.2 Sensory Specifications

1. The mobile platform must be able to gather perceptive data, using active sensors, about its surrounding environment in all directions on a horizontal plane to minimise directional bias in local navigation performance.

2.4.3 Motion Specifications

1. The mobile platform must be able to perform asymptotic posture stabilisation and have a lyapunov stable motion control system.

2. The mobile platform must be able to locally navigate its surrounding environment in real-time.

2.4.4 Summary

Performance specifications have been placed on certain design aspects of the mobile materials handling and routing robot platform prototype, in order to establish design bounds for physical implementation.

2.5 Chapter Summary

Successful MCM implementation is dependant on the design and customisability of the products offered by a particular manufacturing firm. In this regard, Design For Mass Customisation, (DFMC) is extremely important and must not be overlooked.

MCM can be seen as a unique production environment with production characteristics that span past the scope of conventional batch and mass production. MCM implementation requires a concurrent outlook on the integration of components that affect or control the magnitude and/or frequency of customer-induced variations in production rate. This can be achieved by conceptualising MCM production in a control theoretic construct.

Due to the unique nature of MCM production, it is fundamentally important for firms implementing MCM to understand how customers affect production dynamics, in order to create and engineer production systems that are capable of economically facilitating customers needs through effective plant layout structures and flexible materials handling systems. These plant layouts and materials handling systems must be regarded as a single functional or collective unit.

A Flexible Material Routing Primitives (FMRP), provides a mechanism that can allow for the process integration of distributed manufacturing infrastructure subsets in MCM production plant. For the purpose of generic and structured encapsulation, an implementation architecture has been presented that encapsulates the fundamental capabilities required in order to execute a FMRP, in order to maintain production rates under deterministic changes in standard production requirements.

Chapter 3

Hardware Implementation Layer

''Always design a thing by considering it in its next larger context --
a chair in a room, a room in a house, a house in an environment,
an environment in a city plan.'' - Eliel Saarinen

Chapter 2 aimed at characterising the materials handling environment in MCM production operations, in order to expose and define the notion of a **Flexible Material Routing Primitive** (FMRP). This progressed into the development of an implementation architecture, termed the **Autonomous Material Transportation Specification (AMTS)**, to facilitate the execution of such.

This chapter presents the prototype mobile robot platform developed in alignment with the Hardware Implementation Layer of the AMTS, Figure 3.1.

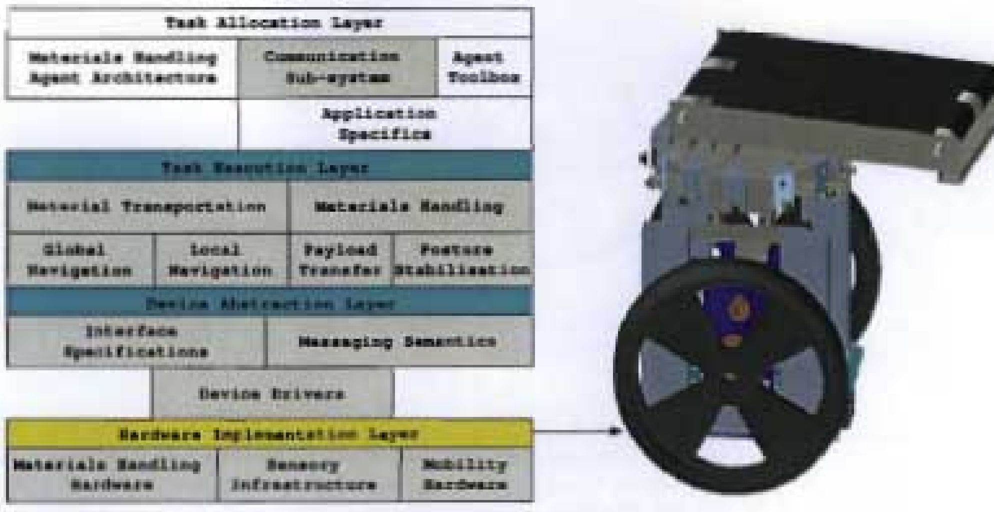


Figure 3.1: Hardware Implementation Layer - The CAD model shown in the figure represents the integrated physical implementation of the underlying hardware sub-blocks in the Hardware Implementation Layer

Description of the hardware prototypes presented here follows a particular format. Firstly, the relevant hardware sub-block of interest is highlighted, including its functional and output specification. Any relevant theory and pivotal design parameters pertaining to the hardware sub-block is presented. Lastly, the hardware prototype developed to facilitate the output specification is presented.

3.1 The Concept of Mechatronics

In recent years, an increasingly inter-disciplinary approach has been taken in solving complex engineering problems. Modern cars, manufacturing infrastructure, such as machine tools, and numerous other systems ranging from hard disk drives to washing machines are examples of the integration between electronic control and communication systems, and mechanical engineering, [46]. The term "Mechatronics" is used to describe this process of integration. Many informal definitions for mechatronics exist in research literature and books on the subject, all roughly describing the same concepts. One such definition follows [46].

Mechatronics. The **synergistic** and **concurrent** integration of the infrastructure associated with the disciplines of Mechanical Engineering, Electrical and Electronic Engineering, Computer and Software Engineering, and Systems and Control Engineering to provide lean, responsive solutions to complex engineering problems

Emphasis is placed on the synergy associated with concurrent integration of existing infrastructures. This aspect separates Mechatronics from solution methods based on achieving performance specifications through the composition of infrastructure, designed from first principles, in disjoint design and solution spaces.

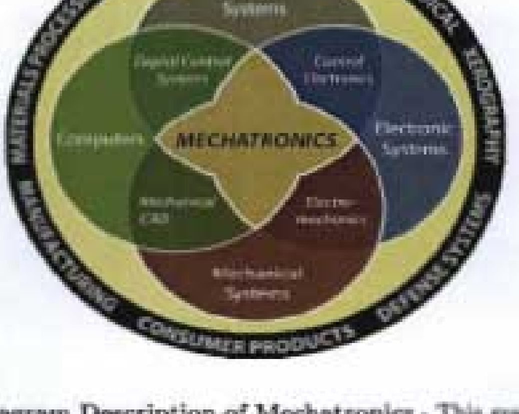


Figure 3.2: Venn Diagram Description of Mechatronics - This graphical representation of the concept of Mechatronics identifies the Mechantronic design space as a unified intersertion of four engineering disciplines, [38]

The hardware prototypes developed for the HIL of the AMTS use multiple embedded systems. Appendix A has been devoted to the introduction of the technology used in implementing these embedded systems. This introduction is required, as reference is made to the technology during the introduction of the Device Abstraction Layer of the AMTS in chapter 4. A review of Appendix A should clarify the technical terminology used in following sections.

3.2 Mobility Hardware Sub-Block

3.2.1 Functional Specification

The Mobility Hardware sub-block specifies the physical device interface to the factory floor that provides low-level motion primitives to facilitate a material transportation task.

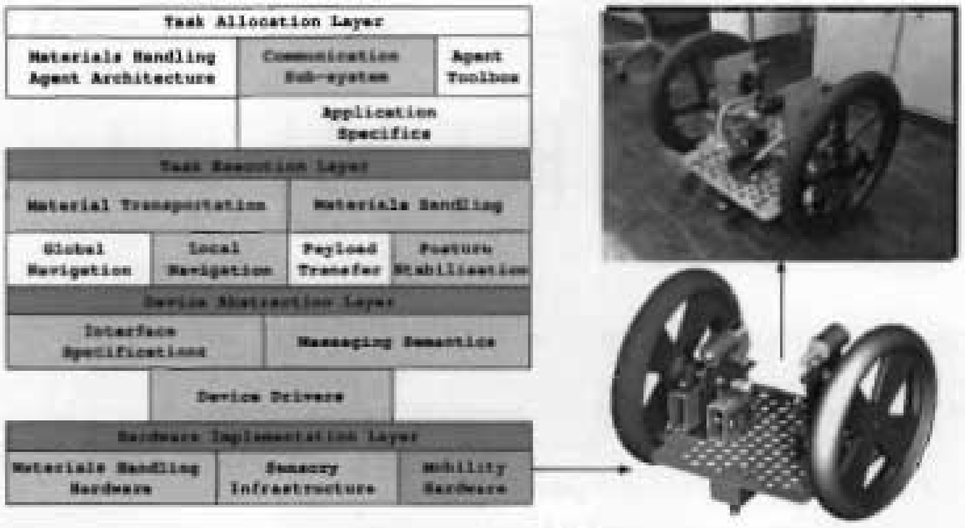


Figure 3.3: Mobility Hardware Sub-Block - As can be seen from the photograph in the figure, the smooth floor of the research laboratory makes high traction wheels vital in proving accurate motion control of the mobile platform

3.2.1.1 Output Specification

The output specification is based on the assumption that the factory floor, on which the mobile device operates, is flat and smooth, thus allowing motions on a plane to facilitate the transportation of material payloads between distributed manufacturing infrastructure subsets. This is a reasonable assumption as safety specifications limit factory floors to smooth flat surfaces.

- Output Specification:
  Planar motion primitives in the configuration space $\mathbb{R}^2 \times SO^1$.

In order to provide planar motion primitives in $\mathbb{R}^2 \times SO^1$, a two wheeled differential drive platform was designed and implemented. The operating specifics of differential drive platforms is covered in section 3.2.2.

3.2.2 Differential Drive Platforms

This section provides an overview of the operating characteristics and implementation aspects associated with differential drive platforms, in order to provide insight into developing a physical implementation to facilitate the output specification of the Mobility Hardware sub-block.

3.2.2.1 Mechanical Configuration and Theory of Operation

The most common mechanical configuration for implementing differential drive platforms consists of two independently driven active drive wheels attached to a main structural support framework, stabilised by passive caster wheels. Most mobile robots used in academic research utilise this standard mechanical configuration.
Differential drives can produce planar motions consisting of combinatorial translations and rotations, Figure 3.4. Translations are achieved by establishing identical angular velocity in both drive wheels. Equal, but opposite, angular velocities in the drive wheels produce pure rotations. Various combinations of translations and rotations can be achieved by varying the difference in angular velocity between the active drive wheels, hence the term differential drive.

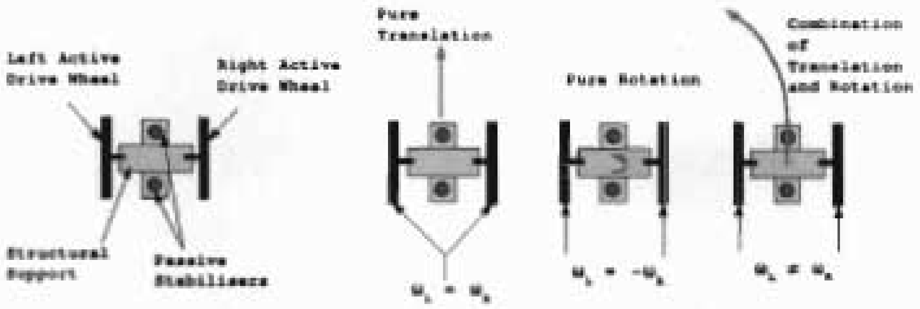


Figure 3.4: Mechanical Configuration and Theory of Operation for a Differential Drive Platform - In some physical implementations of differential drives, only one passive stabilising wheel is used to stabilise the mobile platform. This only provides robust stability in cases where the centre of mass is sufficiently centred over the single caster wheel

3.2.2.2 Important Implementation Aspects

There are multiple aspects that affect the motion performance of a differential drive platform. Active drive wheel slippage is one of the main contributors to positioning errors during odometric calculations on mobile differential drive platforms. Therefore, active drive wheels with good traction characteristics must be selected in their implementation. Also, selecting active drive wheels with large diameters relative to the width of the differential drive allows for lower angular velocities and accelerations on the drive wheels to produce feasible linear velocities, while avoiding inertial based wheel slippage during tight turning motions.

Utilising regular caster wheels for stability induces disturbance torques during tight turning motions, due to the eccentric offset between the caster wheel and its axial attachment, thus contributing to positioning errors. Ball transfer units, however, can be used instead, without the negative impact of disturbance torques. This allows for the development of a simple kinematic model of the differential drive platform.

Differential drives are sensitive to angular velocity differences in the drive wheels. Closed loop motion control systems around each active drive wheel are required to allow for pure translation and rotation motions and overall motion performance.

3.2.2.3 Odometry

The pose of a differential drive represents its position, in terms of location and orientation, with respect to a global co-ordinate system. This is more commonly known as its configuration.

The configuration of a differential drive, $\vec{q}$, in a generalised combination of Cartesian and polar co-ordinates, is described by Eq. 3.1 and shown in Figure 3.5.

$$\vec{q} = [x\, y\, \theta]^T \in \mathbb{R}^2 \times \mathrm{SO}^1 \tag{3.1}$$

Odometry is a means of implementing dead-reckoning, which is a method of determining a mobile platforms current configuration based on a previous known configuration and knowledge of higher-order differential information about the mobile platforms configuration, such as its configuration velocity.
Translation, Eq. 3.2, and anti-clockwise rotation, Eq. 3.3, of a differential drive platform can be specified explicitly in terms of its active drive wheel radii, r, the axle length between its active drive wheels[1], L, and the left and right drive wheel angular velocities, ω_L and ω_R respectively.

$$v = \frac{r}{2}[\omega_R + \omega_L] \tag{3.2}$$

[1]Although only one drive axle is shown in the figure, differential drives have independent drive axles

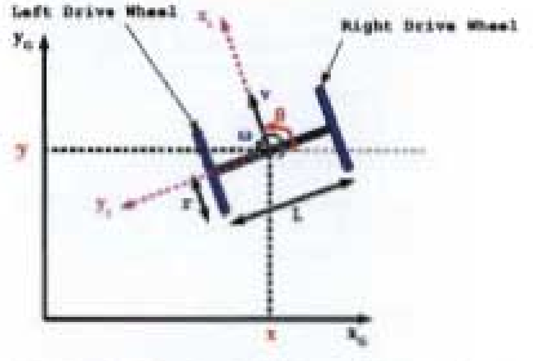


Figure 3.5: Differential Drive - Left and right drive wheels can be controlled independently in order to provide translational and rotational motions

$$\omega = \frac{r}{L}\left[\omega_R - \omega_L\right] \tag{3.3}$$

From the above parameterised translation and rotation, theoretical odometry can be performed on a differential drive platform by integrating its configuration velocity, Eq. 3.4, over time, Eq. 3.5.

$$\dot{\vec{q}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.4}$$

$$\vec{q}_{t+\Delta t} = \vec{q}_t + \int_t^{t+\Delta t} \dot{\vec{q}}\,\mathrm{d}t \tag{3.5}$$

This form of odometry describes the ideal case and can only be considered trustworthy if ω_L and ω_R, as well as all other system parameters, such as L and r, are known with absolute certainty. This is however, not achieved in practice due to imperfect measurements and uncertainty in system parameters, as well as uncertainty in the environment in which the differential drive platform is operating. Robotics research is plagued by uncertainty, which has forced researchers to set moderate performance specifications in view of achieving more robust solutions, insensitive to bounded uncertainty.

Practical differential drive platform implementations utilise embedded control and data acquisition systems to perform odometry. The majority of physical implementations use optical incremental encoders, more specifically quadrature encoders, fitted to the active drive wheels shafts, to measure drive wheel angular velocity. The output signals from the quadrature encoders provide input into embedded micro-controllers that continuously run an embedded odometry algorithm, which essentially represents a discrete form of Eq. 3.5. These quadrature encoders also provide feedback signals for closed loop control of the drive wheels. This is covered in section 3.2.4.

In practice, the configuration error, i.e. the difference between the actual configuration and that estimated through an embedded odometry implementation, grows over time and is known as drift. There exists passive numerical methods that can be used when implementing odometry algorithms in order to minimise drift and prevent degraded positional accuracy of the differential drive platform. One such numerical method is covered in section 3.2.4.3.

One of the drawbacks associated with differential drives is the nonholonomic kinematics of the resulting mobile device. This is a result of the differential constraint imposed on the differential drives generalised configuration velocity, Eq. 3.4, by the rolling without slipping condition[1] exhibited by the drive wheels. It is important to understand these constraints when designing feedback motion plans for a mobile robot platform utilising a differential drive for mobility generation. This requires the implementation of nonlinear, discontinuous or time-varying motion control laws, see section 5.2.2.5.

3.2.3 Physical Implementation

In order to minimise the utilisation of once-off, or non-standard hardware components, an effort was made to construct the differential drive platform by integrating as many standard and/or Commercial Off-The-Shelf (COTS) components as possible.

3.2.3.1 Prototype Overview

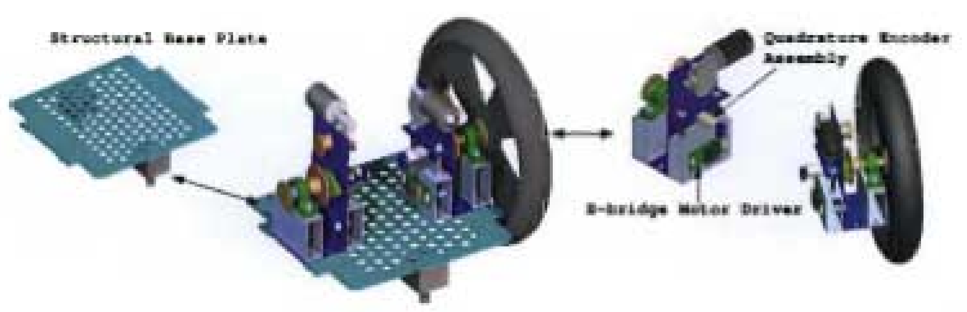


Figure 3.6: Partial Differential Drive Platform Representation - The drive chain has been omitted from the CAD figure for clarity purposes

A partial CAD representation of the differential drive platform developed to facilitate the Mobility Hardware sub-block, is shown in Figure 3.6. A structural base plate with passive stabilising wheels provides structural support for integrated drive units. Two powered drive wheels housed in drive units with integrated quadrature encoder and H-bridge motor driver provide the physical differential drive interface to the factory floor.

3.2.3.2 Structural Base Plate

In the scope of Mechatronics, some form of initial condition or readily understood constraint is required in order to instantiate design progression. In this light and in view of the dimensional specification set on the mobile platform prototype, recall section 2.4.1, the structural base plate was designed as a once-off component. Ball transfer units were selected, over regular caster wheels, to provide static stability for the differential drive platform as they do not induce disturbance torques during tight turning motions, recall section 3.2.2.2.

3.2.3.3 Integrated Drive Units

For reasons of availability and ease of modification, two 20 inch BMX MAG wheels were selected and modified to house drive shafts in order to function as the active drive wheels. These wheels are capable of providing ample traction for the differential drive platform and have good mechanical strength properties, required to facilitate the high impact loads of their native application environment[2].

The drive-shaft support assembly and power train was designed by considering the drive-shaft as the focal component. Peripheral hardware was selected and integrated in order to provide a mechanical support structure for housing and powering the drive-shaft. The drive-shaft is supported by a sub-assembly that integrates the shaft with a quadrature encoder, Figure

[1]Discussed under the motion control constraints in chapter 4
[2]BMX vertical ramp or "vert" riders utilise MAG type wheels often, with great success

3.7.

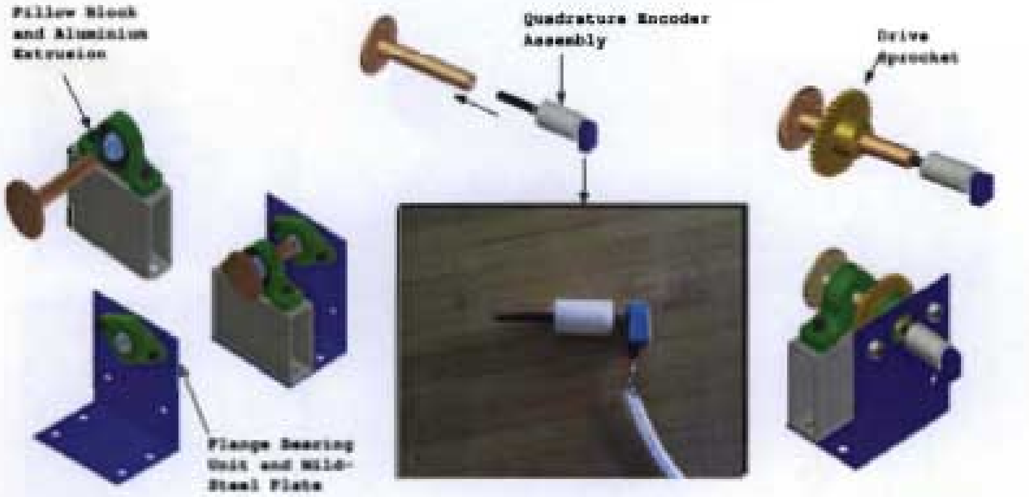

Pillow Block and Aluminium Extrusion
Quadrature Encoder Assembly
Drive Sprocket
Flange Bearing Unit and Mild-Steel Plate

Figure 3.7: Drive-Shaft Support Sub-assembly - Flange and pillow block bearing units provide solid drive-shaft support as well as allow for fine axial adjustments of the drive-shaft

A pillow block bearing unit seated on standard aluminium extrusion supports the wheel hub interface side of the drive-shaft. Support of the drive-shaft terminates at a flange bearing unit on right-angled mild-steel plate. This mechanical configuration provides a compact sub-assembly. Angular velocity feedback information, for control and odometry purposes, is provided by an HEDS-5701-F00 panel mount quadrature encoder that is press-fitted into a nylon housing and M10 bolt assembly. The HEDS-5701-F00 is a 256 count-per-revolution (cpr) quadrature encoder and outputs TTL square waves on 2 channels that are electrically out of phase by 90 degrees. Quadrature decoding of the output signals allows for the angular velocity and position of the drive shaft to be determined. The quadrature encoder assembly was inserted into the drive-shaft after an M10 drill and tap operation was performed on the opposite side to the wheel hub interface.

The drive-shaft is powered through a chain and sprocket power transmission unit driven by a 12 Volt 40 Watt DC geared motor, fitted to a screw-jack-adjustable motor bracket with integrated H-bridge motor driver, Figure 3.8.

The MD03 motor driver by Devantech [43], was selected to function as the electronic motor driver. The MD03 is a medium power fully integrated motor driver and provides both a low-power logic interface and a high-power drive interface, Figure 3.9.
The actual H-bridge circuitry on the MD03 is driven internally by an embedded PIC16F872 (PIC16) micro-controller using PWM at 15kHz. User based control of the H-bridge circuitry occurs implicitly through the embedded PIC16 via its exposed logic interface. Factory flashed firmware embedded on the PIC16 provides four user based control modes, selectable via a DIP switch mounted on the MD03's PCB.

Mode 1: 0v - 2.5v - 5v Analog, 0v full reverse, 2.5v stop and 5v full forward.

Mode 2: 0v - 5v Analog with separate direction control.

Mode 3: RC Mode. Controlled by standard radio control system. Direct connection to RC receiver with 1ms - 2ms pulse with 1.5ms neutral.

Mode 4: IIC interface. Full control with acceleration and status reporting. Up to 8 modules can reside on the same IIC Bus. SCL speed up to 1MHz.

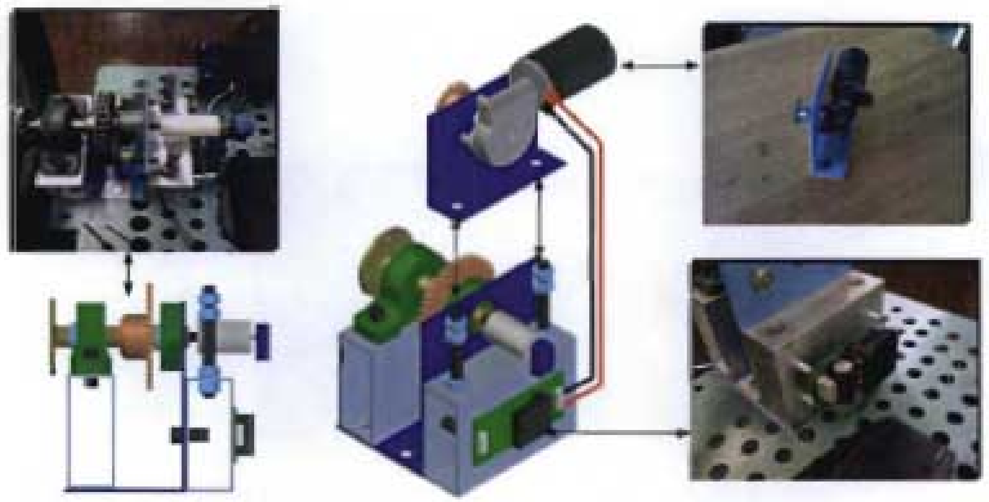


Figure 3.8: Adjustable Motor Bracket With Integrated Drive Electronics - Drive electronics is in the form of an integrated H-bridge motor driver.

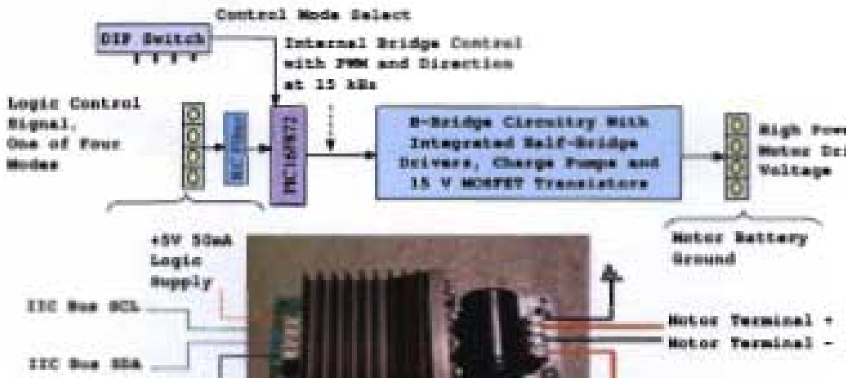

Control Mode Select
DIP Switch
Internal Bridge Control with PWM and Direction at 15 kHz
Logic Control Signal, One of Four Modes
PIC16F872
H-Bridge Circuitry With Integrated Half-Bridge Drivers, Charge Pumps and 15 V MOSFET Transistors
High Power Motor Drive Voltage
Motor Battery Ground
+5V 50mA Logic Supply
IIC Bus SCL
IIC Bus SDA
Logic Ground
Motor Terminal +
Motor Terminal -
+ V Battery (12V)

Figure 3.9: Logic and Drive Interfaces for MD03 Motor Driver - The MD03 is considered a medium power motor driver, succesfully handling continuous currents of 20 Ampares without damaging the board

Mode 2 operates by driving the SDA channel with an analog voltage between 0v and 5v to produce zero to full output drive voltage with separate direction control logic on the SCL channel. A TTL logic level 0 on the SCL channel represents "reverse" direction and logic level 1 represents "forward" direction. Quotation marks indicate that motor direction convention is application specific. The MD03 includes an internal Resistor-Capacitor, (RC) filter on the SDA channel, allowing PWM signals above 20 kHz to provide the same affect as an analog signal, thus allowing the MD03 to interface directly with digital control signals.

3.2.4 Embedded Control Framework and Odometric Implementation

A control framework was designed for the differential drive platform such that higher-level host software systems would have implicit access to two embedded PID velocity control loops, representing the active drive wheels, through a serial communication link into a BrainStem® network. This provided higher-level host software systems with a policy versus mechanism structure when utilising the differential drive platform for motion applications.

3.2.4.1 Control Loop Structure

Each drive unit was provided a dedicated BrainStem® Moto 1.0 module in order to implement a velocity PID loop around each active drive wheel. Although a single Moto 1.0 module is capable of providing the necessary I/O facilities to encapsulate both drive units in a control loop, two Moto 1.0 modules were utilised with one motion control channel disabled, on each module, to effectively double the bandwidth of the embedded controllers, Figure 3.10.

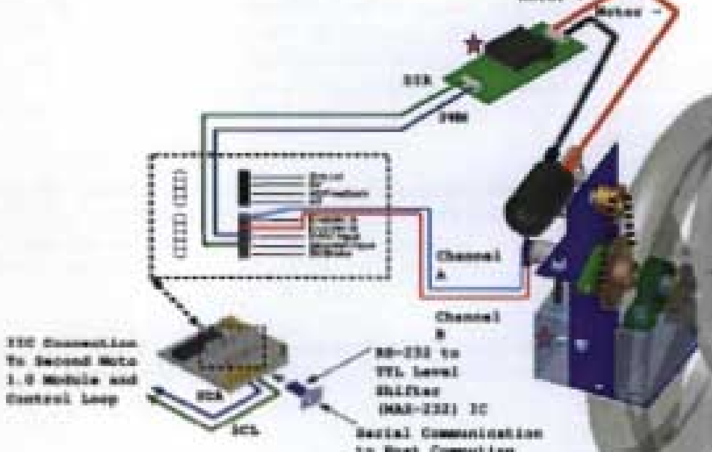

Motor +
Motor -
SDA
PWM
Channel A
Channel B
IIC Connection To Second Moto 1.0 Module and Control Loop
RS-232 to TTL Level Shifter (MAX-232) IC
SDA
SCL
Serial Communication to Host Computing Platform

Figure 3.10: Closed Loop Control Framework Around Each Drive Wheel - The level shifter converts RS-232 signal levels into CMOS/TTL levels required by the BrainStem® Moto 1.0 module

Two Moto 1.0 modules were networked on the same IIC bus by assigning master characteristics to one module, the router, and slave characteristics to the other. This is achieved through a built-in command set. Host computing platforms were provided serial communication access to the embedded BrainStem® network through the serial UART on the router module, Figure 3.11.

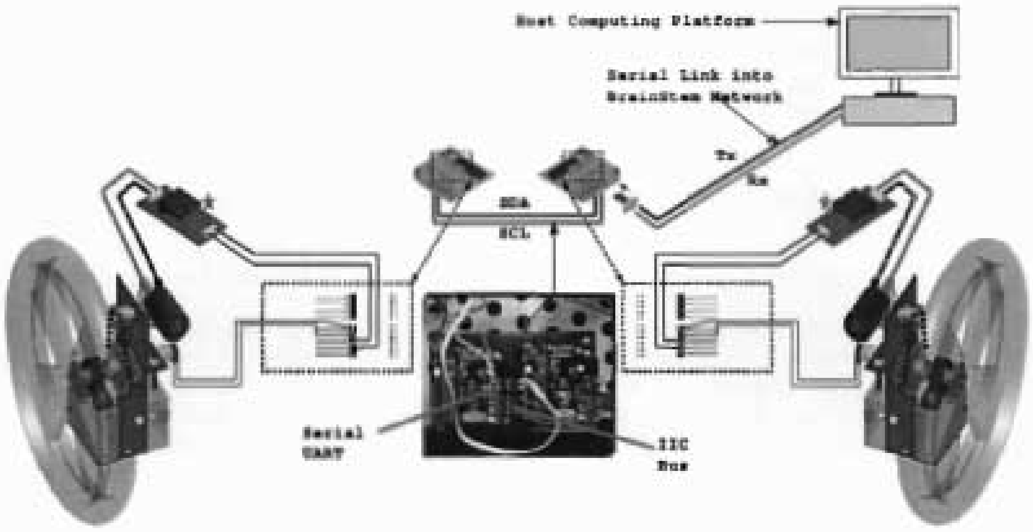


Figure 3.11: Control Loop Structure for the Differential Drive Platform - The serial link into the BrainStem® network runs at 9600 baud. This baudrate was selected as it provided the most stable link

3.2.4.2 Embedded Control Access Policy

Using the TEA language, embedded control code was written, compiled and loaded onto each Moto 1.0 module. The embedded code was designed to bootstrap at power-up and configure each module to perform closed loop velocity PID control on one channel while shutting down the other channel altogether. The control code runs continuously and provides host software with integer write access to selected memory locations on each modules scratchpad[1]. The embedded TEA program then utilises the host written scratchpad value as the reference value into the underlying velocity PID control loop.

The BrainStem® Moto 1.0 module can perform "velocity damping" when configured for velocity PID control implementation. In this setup, changes in velocity reference values into the PID control loop occur gradually over time. Velocity damping acts as an acceleration limiter and was used in the implementation of the control framework for the differential drive. The TEA code implementing the embedded control code is listed in Appendix B.2.

Once each control loop is active, higher-level host software systems write the selected scratchpad locations through the serial communication link into the BrainStem® network, to set the angular velocity of each drive wheel.

3.2.4.3 Odometric Implementation

The encoder feedback velocity metric used by the Moto 1.0 module is "encoder counts per PID period", which is the number of encoder pulses accumulated in the time between PID calculations. The PID period for each module is set to 20mS as part of the start-up routine of the embedded TEA programs.

As the counts-per-revolution of the HEDS-5701-F00 is known to be 256 cpr, along with knowledge of the quadrature decoding algorithm implemented on the Moto 1.0 module, the encoder velocity metric can be used to determine the angular velocity of each drive wheel and

[1]A scratchpad is the common terminology given to a globally shared area of memory in an embedded micro-controller that allows embedded processes to share data

thus through Eqns. 3.2 and 3.3 provides an estimate of the linear and angular velocity of the differential drive platform. Although this knowledge can provide input into odometry algorithms, such as discrete explicit Euler algorithmic implementations of Eq. 3.5, odometry was performed rather, through acquisition of accumulated encoder counts on the active control channel on each Moto 1.0 module. This form of odometry, based on accumulated encoder counts per drive wheel, is used often in mobile robotics and does not explicitly require the integration of velocities which creates a simpler algorithm implementation. One of the drawbacks associated with BrainStem® modules, however, is that they do not implement floating point arithmetic as part of their embedded program execution. Due to this drawback, odometric calculations had to be performed on the host computing platform after reading the memory mapped I/O port associated with storing the value of the accumulator on each Moto 1.0 module through the serial UART.

The metric used in implementing the odometry algorithm for the differential drive platform is "metres per encoder pulse" (mpep), and is formulated as follows.

$$\text{mpep} = \frac{2\,\pi\,r}{1024} = \frac{\text{Drive Wheel Circumference}}{\text{Encoder Pulses Per Revolution}}$$

Where r is the radius of the active drive wheels, 0.254 m or 10 inches, and 1024 are the number of electronic pulses per revolution for the HEDS-5701-F00 quadrature encoder under the 4x quadrature decoding algorithms implemented as part of the embedded firmware of the Moto 1.0 modules. The metric represents the minimum distance detectable by the embedded electronic control infrastructure. For this particular differential drive, this happens to be $\frac{2*0.254}{1024}$ or 0.0016 m (1.6 mm). With this metric, odometry can be performed by summing incremental changes in the linear displacement of each drive wheel and relating these linear distances to changes in the linear displacement and angular orientation of the differential drive platform.

Referring back to section 3.2.2.3, passive measures of minimising drift can be implemented through numerical methods in odometry algorithms. A particular form of odometry algorithm, based on the mpep metric described above, which is equivalent to performing 2^{nd} order Runge-Kutta integration of Eq. 3.5 [15] was implemented, Algorithm 1.
The odometry implementation used for the differential drive platform developed here implements this algorithm in an infinite loop in higher-level software systems on a host computing platform, this is covered more during the discussion of the higher-level software systems in chapter 4.

3.2.4.4 Preliminary Performance Testing

During preliminary testing, a laptop was utilised on-board the differential drive platform to act as its host computing system. A laptop running the Fedora Core 7 Linux distribution was selected. Access libraries, in the form of C source code, are available for communicating with the BrainStem® modules from Linux through a serial link. Utilising these access libraries, a simple C application was written that writes PID reference values into scratchpad locations on each Moto 1.0 in the network through a serial link, assigned through the structure of the embedded TEA programs, and reads the memory mapped I/O associated with velocity feedback inputs.

Short runs were performed by providing various velocity step inputs into the differential drives embedded control loops and recording the response in file structures created at program run-time, Figure 3.12.
As can be seen from the response, rise times are within specifications set in section 2.4

Algorithm 1: Runge-Kutta 2nd order Equivalent Odometry Algorithm

```
Input: The left and right wheel encoder accumulator values, $N_L$ and $N_R$
Output: The estimated configuration for the differential drive platform, $\vec{q}_r = [x_r\ y_r\ \theta_r]^T$
begin
    // Initialise Odometric Configuration $\vec{q}^-$
    if $\vec{q}^-$ not initialised then
        $N_L^- \leftarrow N_L$
        $N_R^- \leftarrow N_R$
        $x_r^- = y_r^- = \theta_r^- = 0$
        set odometry initialised flag
        return
    $\delta N_L \leftarrow (N_L - N_L^-)$
    $\delta N_R \leftarrow (N_R - N_R^-)$
    // Use mpep metric to determine linear displacement of each active drive wheel in
       metres
    $\delta L_w \leftarrow \delta N_L \times \text{mpep}$
    $\delta R_w \leftarrow \delta N_R \times \text{mpep}$
    // Determine incremental angular and linear displacement of the differential drive
       platform
    $\delta\theta_r \leftarrow (\delta R_w - \delta L_w)/A_L$          // $A_L$ = axle length between drive wheels
    $\delta d \leftarrow (\delta L_w + \delta R_w)/2$
    // Implement 2nd order equivalent Runge-Kutta numerical integration
    $\hat{\theta} \leftarrow \theta_r^- + (\delta\theta_r/2)$
    $x_r \leftarrow x_r^- + \delta d \cos\hat{\theta}$
    $y_r \leftarrow y_r^- + \delta d \sin\hat{\theta}$
    $\theta_r \leftarrow \theta_r^- + \delta\theta_r$
    // Normalise $\theta_r$ to an element $\in [-\pi, \pi]$
    $\theta_r \leftarrow \text{NORMALISE}(\theta_r)$
    // Update configuration
    $\vec{q}^- \leftarrow \vec{q}$
    return $\vec{q}$
end
```

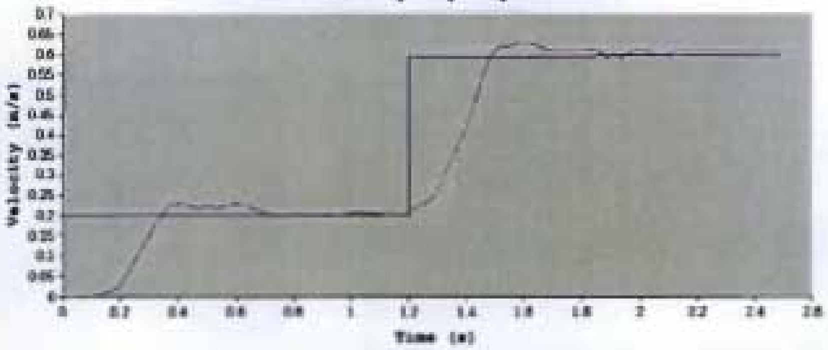


Figure 3.12: Sample Step Response - The error seen at 0.6 m/s is not to be confused with steady state error in the underlying PID loop. This is due to the quantisation errors of digital control hardware, an extremely important characteristic that affects the performance of posture stabilising controllers

3.2.5 Summary

The performance and implementation aspects of differential drive platforms has been covered to provide insight into the development of a physical implementation to facilitate the motion specifications of the Mobility Hardware sub-block of the Hardware Implementation Layer. The physical differential drive platform consists of two functional units that, under composition, provide both structural support and drive infrastructure to allow for robust motion and odometric performance of the differential drive platform. The drive infrastructure is in the form of integrated drive units. These drive units have been encapsulated in a closed loop control framework, enabled through a set of a PIC18C252 based embedded motor controllers, namely BrainStem® Moto 1.0 modules. The odometry algorithm uses a particular form of configuration update that makes it numerically equivalent to performing Runge-Kutta 2^{nd} order integration of Eq. 3.5. Unfortunately, due to the numerical execution environment provided by the embedded run-time kernel on the BrainStem® modules, this odometry algorithm had to be implemented on a host computer. This is covered in chapter 4.

The physical implementation of the differential drive platform is shown in Figure 3.13

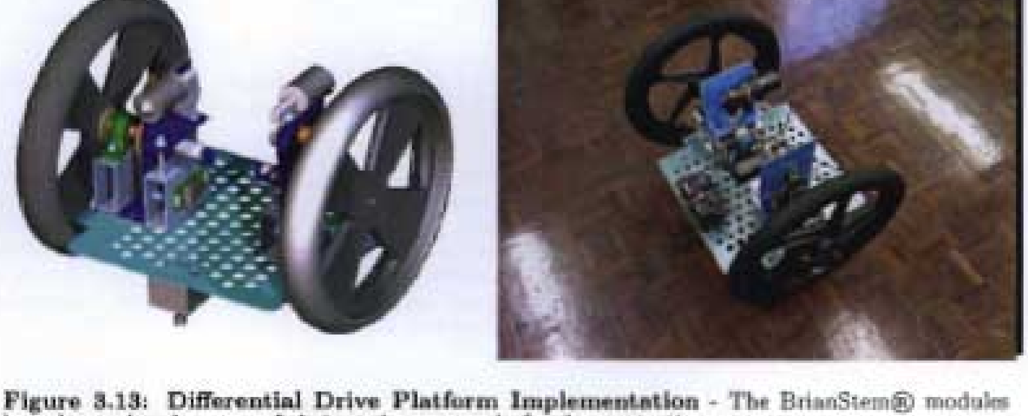


Figure 3.13: Differential Drive Platform Implementation - The BrianStem® modules have been placed on top of their enclosure purely for demonstration purposes

3.3 Sensory Infrastructure Sub-Block

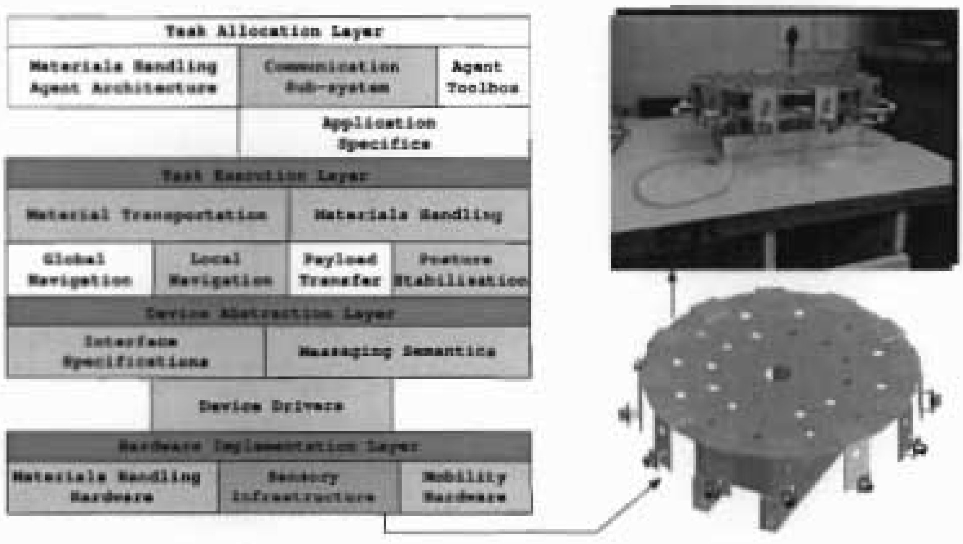


Figure 3.14: Sensory Infrastructure Sub-Block - As can be seen in the figure, ribbon cable has been used for all serial communications as this type of cable provides a sound physical medium for data transmission allowing higher baudrates to be used for communication purposes

3.3.1 Functional Specification

The Sensory Infrastructure sub-block specifies the active sensory infrastructures used by the mobile materials handling and routing robot platform to gather structural information on its local surrounding environment for perception purposes.

3.3.1.1 Output Specification

The output specification is such as to provide an even sensory envelope, independent of the orientation of the mobile materials handling and routing platform. This is to ensure that the mobile platform can react in a uniform manner as obstacles come into scope from all directions during operation.

- Output Specification:
  360^0 active sensory perception within a circular planar region at least three times the effective diameter of the mobile materials handling and routing platform.

Multiple active sensing technologies exist for robotic perception applications. These technologies are characterised by varying degrees of accuracy, power requirements, and cost. Technologies include Light Detection and Ranging (LIDAR) systems, active vision, infra-red systems and ultrasonic technologies. Of these technologies, ultrasonic sensing was selected to facilitate the output specification of the Sensory Infrastructure sub-block. Ultrasonic sensing technology has been implemented on multiple integrated sensors that are relatively cheap, have low power requirements and do not require powerful computational infrastructure to acquire range data. An overview of the characteristics of ultrasonic sensors is presented in section 3.3.2

3.3.2 Ultrasonic Sensors

Ultrasonic sensors have multiple operating characteristics that must be well understood when developing sensing systems.

3.3.2.1 Theory of Operation

Ultrasonic sensors detect distances, for range acquisition purposes, by emitting an ultrasonic sound wave into the surrounding environment and measuring the time taken for the echoed sound wave to be returned to the sensor, due to reflection off structural objects in the environment. Sound wave generation is provided by an integrated electronic device including a transceiver that both emits the sound wave, through electronic excitation of a disk membrane, and detects the signal when the sound wave echo excites the disk membrane on its return.

3.3.2.2 Capability Parameters

Due to physical bandwidth limitations of the membrane on the transceiver, there is a finite distance in front of the membrane face that can not effectively induce range readings through object detection, as the membrane is still vibrating under transmission excitation when the echo is returned. This is known as the blind zone and is a function of the physical bandwidth of the membrane, Figure 3.15.

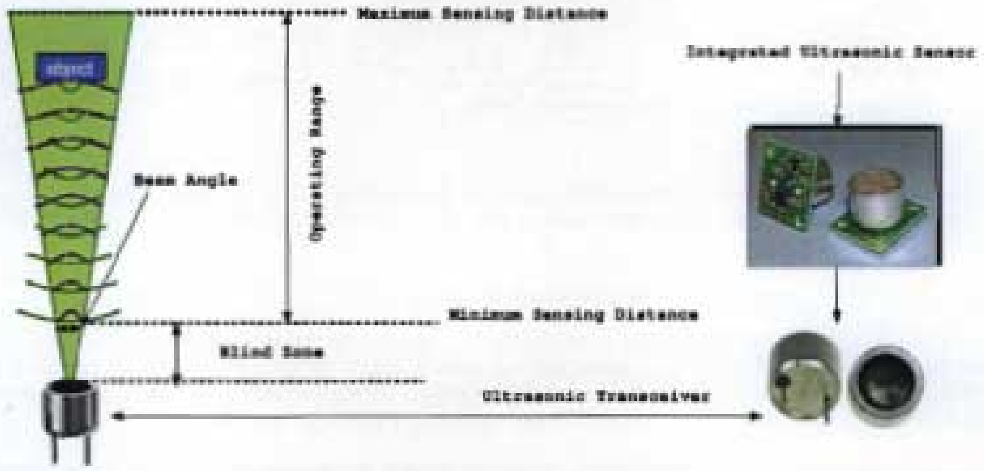


Figure 3.15: Ultrasonic Range Parameters - In modern times, it is uncommon for researchers to design and manufacture their own ultrasonic sensors, as cheap, highly integrated, and effective ultrasonic sensors are available on the market, designed specifically for robotic applications

The capability of an ultrasonic sensor, with regard to acquiring range readings of the objects in its local environment, is determined by multiple parameters. The maximum sensing distance is the largest reliable range reading detectable by the ultrasonic sensor.

The beam angle effectively determines the conical volume of region in front of the sensor in which an obstacle must reside in order to successfully return an echoed sound wave to the transceiver. This is known as the detection zone. Even if obstacles reside in the detection zone, it is not always guaranteed that the return echo will reach the transceiver. This is due to environmental characteristics discussed in section 3.3.2.4.

3.3.2.3 Physical Emission Characteristics

The ultrasonic sound wave emitted by the sensors transceiver, its beam, is not evenly distributed across the beam angle, Figure 3.16
This physical aspect of the signal gives ultrasonic sensors a directional bias in terms of range accuracy. Wide objects detected near the center of the beam provide more accurate range readings over objects detected near the beam periphery.

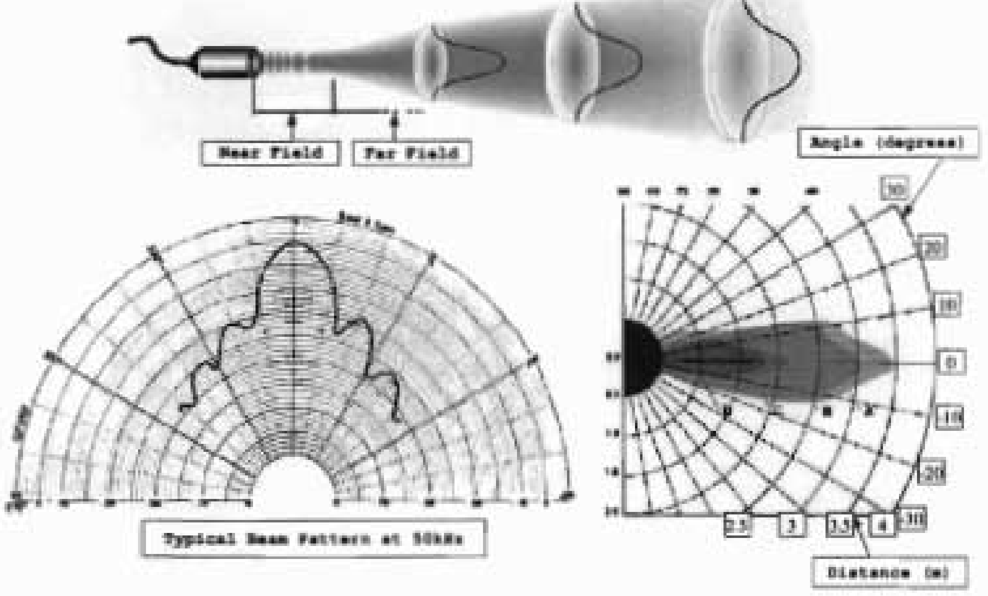


Figure 3.16: Ultrasonic Wave Propagation Geometry - The majority of modern ultrasonic sensors operate at 40KHz as this frequency provides good performance and propagation geometry

3.3.2.4 Environmental Performance Impacts

Ultrasonic sensors are extremely sensitive to the geometric properties of the objects in the detection region of the beam. Occasionally, an echo is refracted and not reflected off an object in the detection region. This produces random range readings that can affect the performance of mobile platforms utilising ultrasonic sensors for perception purposes.

In sensory systems utilising multiple fixed-position ultrasonic sensors, beam signals emitted by one sensor can occasionally be detected by another leading to the problem of sensor cross-talk, Figure 3.17.

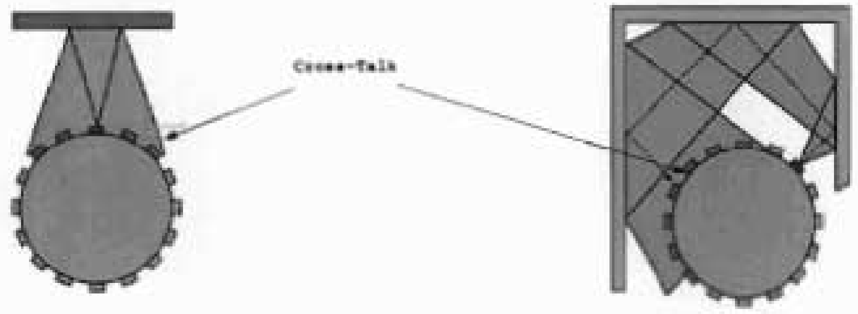


Figure 3.17: Sensor Cross-Talk - Mobile robot platforms experiencing cross-talk seldom travel in a smooth motion as data fusion algorithms start producing random outputs that degrade performance in directed motions

The negative effects of cross-talk can be minimised but never diminished absolutely. Researchers have developed methods of reducing cross-talk by providing unique beam signals to each ultrasonic sensor using pseudo-random binary signatures [41]. Other methods are based on firing sequences in which a sensor pair that are less likely to establish an instance of cross-talk, due to their relative position and orientation, are fired consecutively.

3.3.3 Hardware Implementation

12 SRF02 ultrasonic range finders were selected, based on insight provided by the capability parameters described under section 3.3.2.2 and integrated into an embedded sensor platform. An overview of the SRF02 ultrasonic range finder follows in section 3.3.3.1

3.3.3.1 The SRF02 Ultrasonic Range Finder

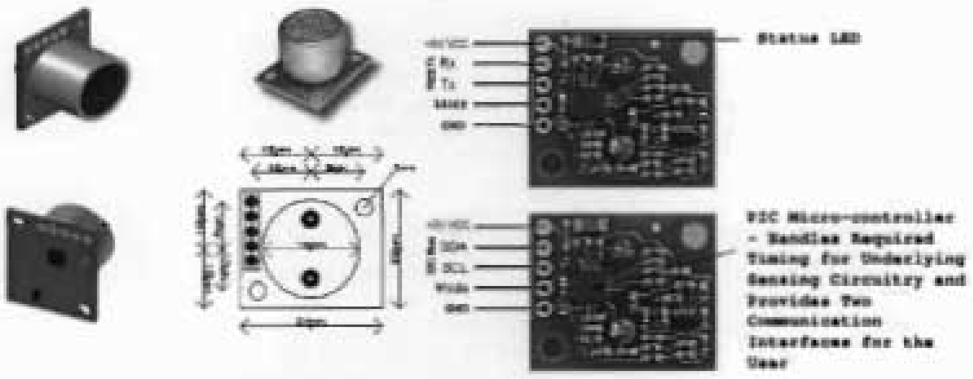

+5V VCC
Rx
Tx
Mode
GND
Status LED
+5V VCC
SDA
SCL
Mode
GND
PIC Micro-controller - Handles Required Timing for Underlying Sensing Circuitry and Provides Two Communication Interfaces for the User

Figure 3.18: SRF02 Ultrsonic Range Finder and Communication Pin-out - The status LED is used to indicate an emission beam burst, as well as indicate the current IIC address at power up

The SRF02 ultrasonic range finder is manufactured by Devantech [43] and is a fully integrated sensor with embedded PIC micro-controller, which handles timing of ultrasonic beam transmission and reception, Figure 3.18. The embedded PIC also acts as an interface between the user and the actual ultrasonic sensing circuitry, providing two communication interfaces for the user. For the development of the embedded sensor platform, the IIC interface was selected for communication.

In the IIC communication mode, the embedded PIC abstracts the sensing circuitry behind 6 registers allowing the user to initiate ranging primitives through standard IIC communication protocols with the SRF02. Register 0 is the command register. Writes to this register initiate a ranging session. Reads on the command register returns the SRF02 firmware revision. Register 2 and 3 hold the 16 bit unsigned result from the last ranging session. The meaning of the value stored in these registers depends on the value written to the command register, that initiated the associated ranging session. Depending on the ranging command written to the command register, the result stored in registers 2 and 3 can represent the last range in inches, centimetres or the beams flight time in micro seconds (uS).

After a ranging command is written to the command register, a wait period of 70 milliseconds must occur in order to allow the range to complete and the beam energy to dissipate. An IIC read sequence on registers 2 and 3 then returns the range result to the embedded system requesting range data, e.g. an embedded control module or micro-controller.

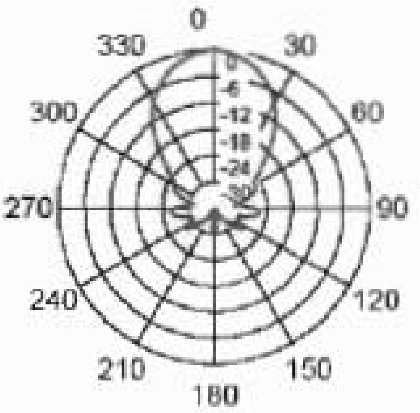


Figure 3.19: Beam Propagation Geometry for the SRF02 Ultrasonic Range Finder - As can be seen in the figure, the SRF02 has a relatively even beam propagation geometry. This makes the SRF02 a well suited ultrasonic sensor for mobile robotics

3.3.3.2 Physical Emission Characteristics of the SRF02

The SRF02 has been designed to function effectively over its operating range and has the beam propagation geometry shown in Figure 3.19 [37].
The SRF02's transceiver membrane is capable of producing beam angles approaching 60 degrees. In order to facilitate the specifications set by the Sensory Infrastructure sub-block, 6 SRF02 sensors would need to be placed 60 degrees apart. This configuration however would be prone to range accuracy errors during detections near the periphery of the beam, induced through the uneven propagation geometry of ultrasonic sensors, recall section 3.3.2.3. To passively overcome these accuracy bias characteristics, 12 SRF02 sensors were placed 30 degrees apart in order to maximise the accuracy of the range data. This has a negative affect on cross-talk however, although by implementing structured firing sequences, these negative affects can be minimised.

3.3.3.3 Sensor Platform Implementation

In order to position the SRF02 sensors in a 360 degree array, an aluminium bracket was designed to hold each sensor. Each bracket housed a connector hub to allow easy attachment of the SRF02 onto the IIC bus. An aluminium disk plate provided structural support for the 12 aluminium brackets which were positioned 30 degrees apart from one another around the periphery of the disk. Standard aluminium extrusions were stacked together to create a beam upon which the disk was fastened, Figure 3.20.

3.3.3.4 Embedded Control Framework

An embedded control framework was implemented to provide host computing systems with serial access so sensory range data from the sensor platform. For this purpose, a BrainStem® GP 1.0 module was used to integrate each SRF02 sensor onto an IIC bus and handle the firing of each sensor, utilising an embedded TEA program, Appendix A.

SRF02 ultrasonic range finders are shipped with an IIC address of 0xE0. In order to interface multiple sensors on the same bus, each consecutive sensor needed to be assigned a different IIC address. Documentation on how to do this is provided with the SRF02 and consists of writing three consecutive commands to the command register. During address changes, only one SRF02 must reside on the IIC bus and three commands written to the command register, the current IIC address followed by the command 0xA5 and lastly the new requested IIC address.

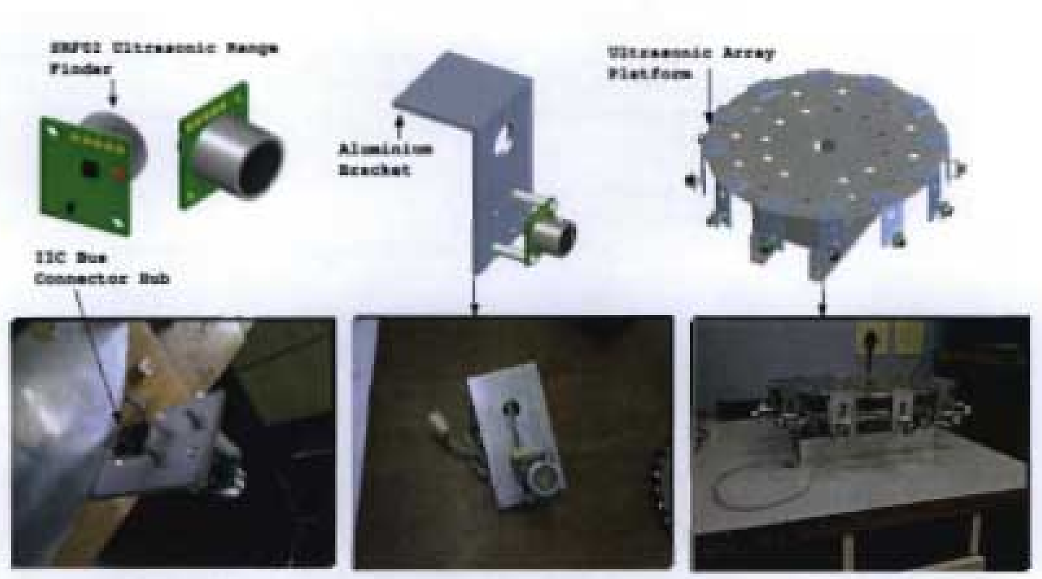


Figure 3.20: Ultrasonic Sensor and Sensor Platform - The Inter Integrated Circuit (IIC) bus has become the standard communication interface for the majority of commercial off the shelf sensors for robotic applications

This was done on 11 of the 12 SRF02 sensors purchased and provided IIC addresses ranging from 0xE0 to 0xF6.

The embedded TEA program implemented on the GP 1.0 module executed IIC read and write routines to fire each SRF02 ultrasonic sensor. Sensors were fired in a sequence in which consecutive firing pairs were opposite one another on the sensor platform. This aided in minimising the probability of cross-talk, Figure 3.21.

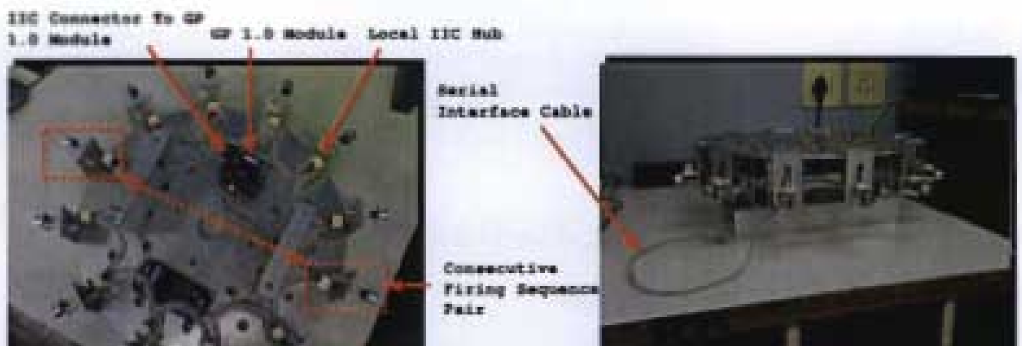

IIC Connector To GP 1.0 Module
GP 1.0 Module
Local IIC Hub
Serial Interface Cable
Consecutive Firing Sequence Pair

Figure 3.21: Ultrasonic Sensor Platform and Embedded Control Interface - Having IIC connector hubs allows for easy removal of the sensors from the platform, if required

The TEA program was configured to bootstrap at power up and continuously fire the SRF02's in the centimeter ranging mode. The return values were stored in an array in the scratchpad to allow host software to read the scratchpad array through a serial communication link to retrieve the range readings, see Appendix B.1.

In order to interface the sensor platform with the underlying differential drive platform, developed in section 3.2.3, Standard aluminium extrusions were utilised. Large rectangular extrusions were selected as they provided mounting points for peripheral hardware such as DC-DC converters, Figure 3.22.

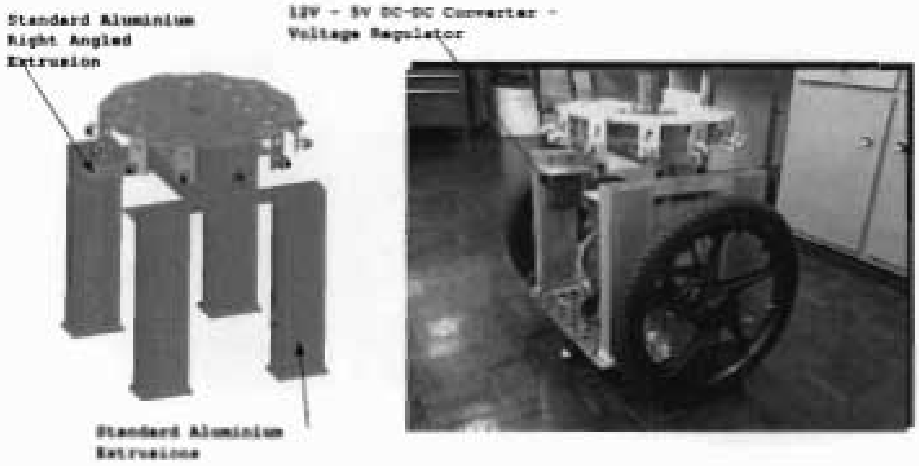

Standard Aluminium Right Angled Extrusion
12V - 5V DC-DC Converter - Voltage Regulator
Standard Aluminium Extrusions

Figure 3.22: Interface Support For Sensor Platform Integration - Industry standard voltage regulators were used to provide the platform's electronic infrastructure with constant power. This prevents any unwanted power glitches from resetting parameters or destroying hardware

3.3.4 Summary

Ultrasonic sensors were selected as the active sensory infrastructure to facilitate the specifications set by the Sensory Infrastructure sub-block of the Hardware Implementation Layer.

An embedded system implementation makes the ultrasonic range data available to host computing and software systems through a serial communication link. The sensors are fired in consecutive pairs positioned opposite one another on the sensor platform, in order to minimise cross-talk. Range data in stored locally on the embedded control module in a scratchpad array which host computing and software systems have read access through a serial communication link.

3.4 Materials Handling Hardware Sub-Block

3.4.1 Functional Specification

The Materials Handling Hardware sub-block is a specification on the capabilities of the hardware device providing the physical interface to a material payload for handling purposes.

3.4.1.1 Output Specification

The output specification has been based on the payload manipulation required in order to perform a material payload transfer operation, recall section 2.3.2.3.

- Output Specification:
  Hardware device must provide the material payload at least one extra degree of motion freedom above that provided by the underlying mobility device.

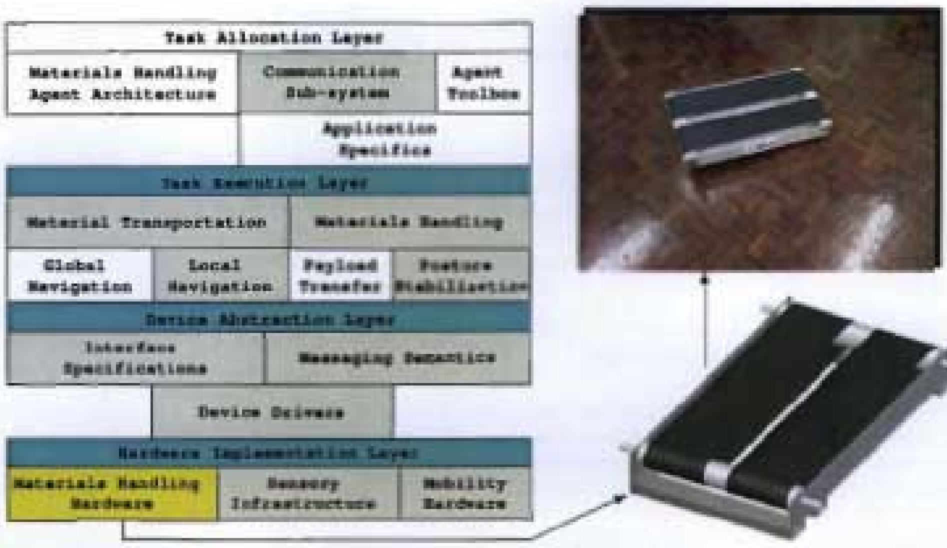


Figure 3.23: Materials Handling Hardware Sub-Block - Although not implemented yet, sensing system on the conveyor, such as infra-red distance detectors can be used to detect payload instances

The hardware devices used for materials handling purposes are extremely application specific, and so design of a materials handling device to facilitate the above specification was based on creating infrastructure to interface with the existing conveyor systems of the Computer Integrated Manufacturing (CIM) cell in the research laboratory, recall the specifications of section 2.4.

3.4.2 Hardware Implementation

The materials handling device designed to facilitate the output specification consists of two functional components, Figure 3.24.

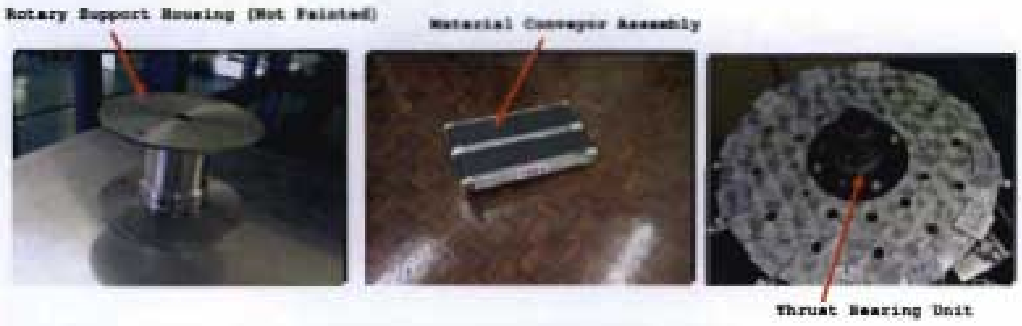

Rotary Support Housing (Not Painted)
Material Conveyor Assembly
Thrust Bearing Unit

Figure 3.24: Materials Handling Device Implementation - The thrust bearing unit facilitates smooth motions under high load conditions, although for research purposes this does not take effect

A material conveyor assembly provides material translation motion through a belt drive around two nylon rollers. The cambered profile machined into the rollers keeps the belts tracking in a straight line. A rotary support housing and thrust bearing unit integrates the material conveyor assembly with the underlying sensor platform. This provides the material conveyor with an added degree of freedom thus satisfying the output specification.

3.4.3 Summary

Due to the application specifics of materials handling hardware devices, the Materials Handling hardware sub-block only places output specification on the additional degrees of freedom provided by the materials handling hardware for material payload manipulation and transfer. These extra degrees of freedom are required in order to increase the overall accuracy and, more importantly, the repeatability of the materials handling hardware and associated transfer operations.

3.5 Chapter Summary

This chapter presented the physical prototype implementations developed to facilitate the output specifications of the underlying hardware sub-blocks of the Hardware Implementation Layer. The physical prototypes are encapsulated in embedded system frameworks, making the devices available through a serial communication link for host computing platforms to gain mediated access to their functionality.

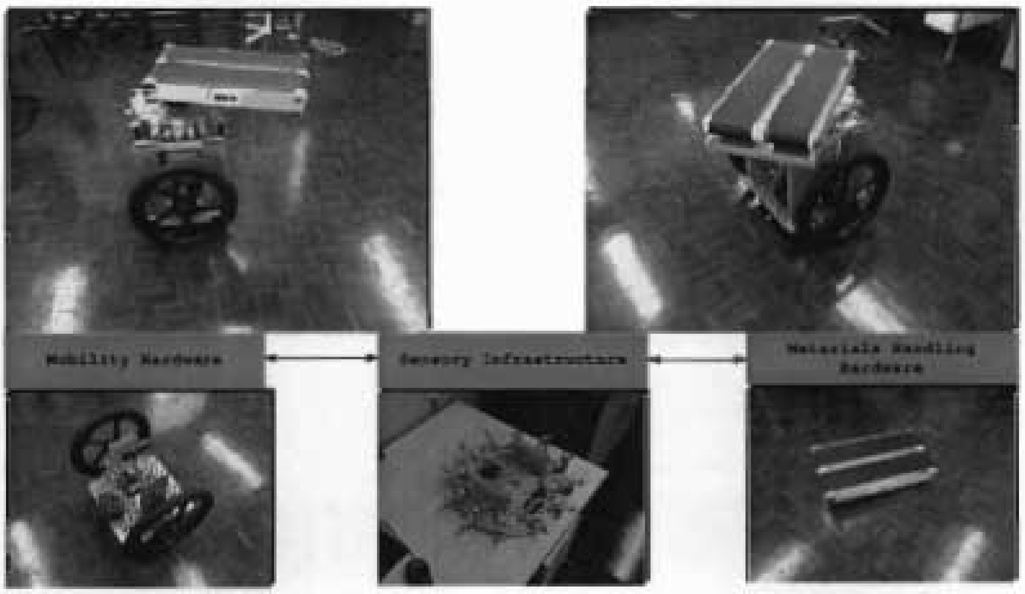


Figure 3.25: HIL Physical Implementation - One can see the mobile robot platform in this sense as an integrated collection of devices rather that a monolithic platform

The AMTS orientates the physical implementation of the HIL away from monolithic mobile robots, and onto a collection of functionally correlated, yet operationally disjoint devices. This aspect of the architecture can be interpreted as an integrated axiomatic framework where there is a one to one mapping between design parameters and functional requirements, Figure 3.25.

The physical robot implementation, as a collective unit of three integrated prototype devices, has been named in order to follow in the norm of robotic system development as well as allow for easy reference to the mobile robot platform. The mobile device has been named "RollerMHP" after its wheeled mobility and application scope.

Chapter 4

Device Abstraction Layer

''Design and programming are human activities; forget that and all is lost.''
- Bjarne Stroustrup

The previous chapter covered the development of RollerMHP, in alignment with the underlying specifications encapsulated in the Hardware Implementation Layer of the AMTS. This chapter presents the Open Source software system used in implementing device abstractions above the Hardware Implementation Layer. This was required in order to allow for control software scalability by implementing generic control applications that establish homogeneous message transfer constructs between abstract robotic device classes. Robotic device abstractions are provided by software drivers which drive the robotic hardware and perform the required message translation between standardised "interfaces", Figure 4.1.

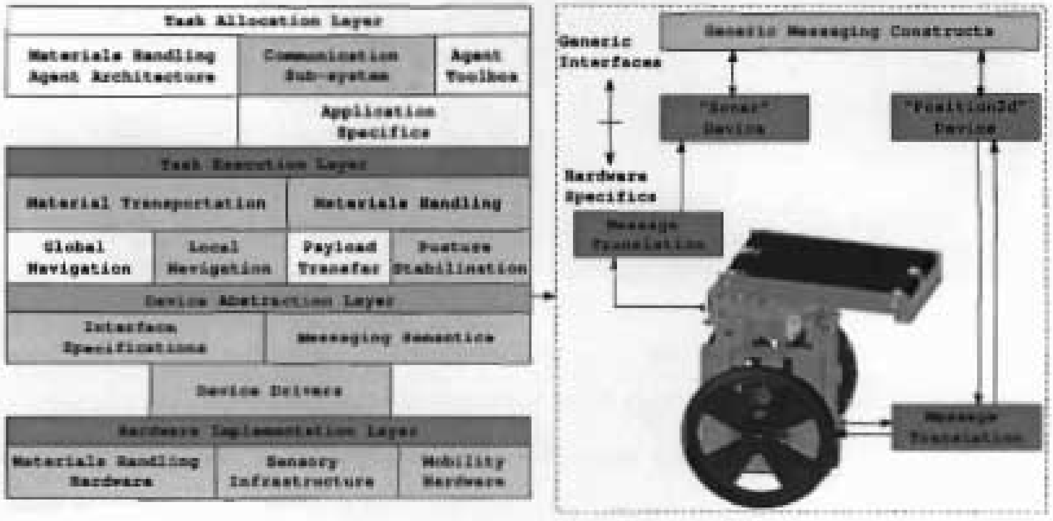


Figure 4.1: Device Abstraction Layer - This layer of the AMTS aligns itself with the concepts of Operating System theory, where the Hardware Abstraction Layer (HAL) works in conjunction with underlying device drivers to provide a homogeneous execution environment for application programs

Description of the software system used in implementing the Device Abstraction Layer has been deferred to section 4.2 in order to present the computing infrastructure selected to operate as a "localhost" computer on-board RollerMHP.

4.1 RollerMHP's Onboard Host Computer

RollerMHP required an on-board computer to integrate and operate its underlying embedded systems, as well as hold software and communication systems to provide higher-level management frameworks with a "task sink interface"[1].

4.1.1 Selected Computer System

An x86 based computing platform was selected to function as RollerMHP's on-board computer. An x86 architecture was selected over more embedded architectures such as ARM and PowerPC as it allowed for a smooth installation of standard Operating Systems.

4.1.1.1 Mainboard and CPU

RollerMHP's on-board host computer consists of a VIA C7VCM Mini-ITX form factor mainboard with integrated VIA C7 CPU running at 1.5 GHz [30]. An 80 Gigabyte hard drive functions as the on-board storage device and is attached to the mainboard through its IDE interface. For memory support, the mainboard has one SO-DIMM socket to hold a DDR400/333 SDRAM memory module. The board can support up to 1GB of RAM although only 256MB was used in this application.

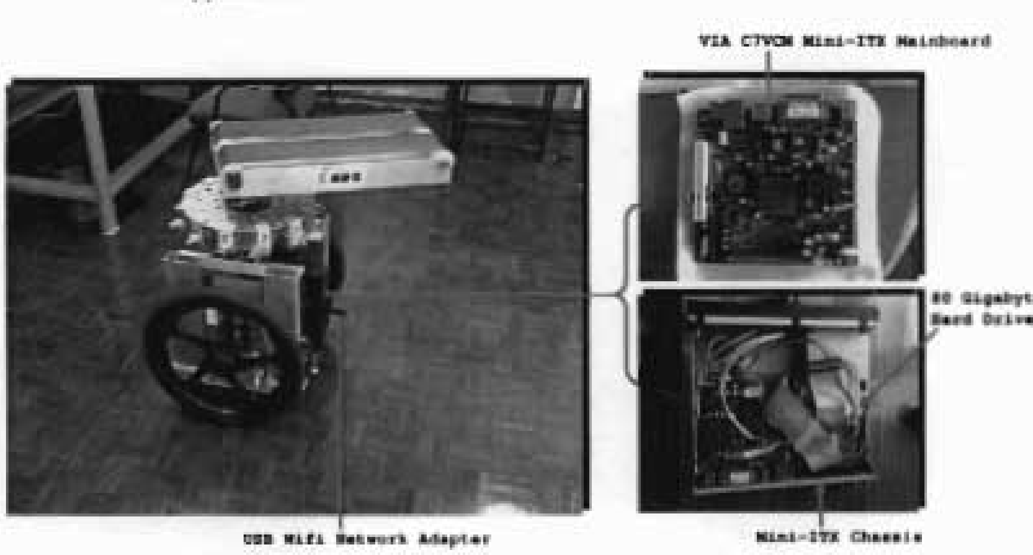


Figure 4.2: Computational Infrastructure On-board RollerMHP - The Mini-ITX form factor has become the preferred form-factor for high power computing in mobile robotic applications, due to its small size and standard bus interfaces such as PCI

A built-in ATX DC-DC converter means that the motherboard only needs a 12 Volt power supply input. All other voltages, including those needed by the hard drive are created and provided on-board the C7VCM mainboard. The mainboard and all peripheral hardware is housed in a chassis designed to facilitate the Mini-ITX form factor, Figure 4.2.

4.1.1.2 Operating System Implementation

Open Source Operating System platforms, such as those based on the Linux kernel, have become widely used in the academic research community due to their stability and the growing

[1]The author defines a "task sink interface" as a software and communication system that can accept and interpret FMRP task assignments

number of Open Source software developers.

RollerMHP's onboard computer runs the Fedora Core 7 Linux distribution. Fedora Core, or more recently, just "Fedora", is considered as one of the more technical Linux distributions allowing the user to configure and tune the Operating System to suit their specific application, whether it be for desktop use or an embedded Real-Time system that utilises Real-Time extensions of the base Linux kernel.

4.1.1.3 Global Communication Infrastructure

Although the Task Allocation Layer of the AMTS has not been implemented yet, a communication sub-block was required in order to assign motion tasks to RollerMHP, Figure 4.3.

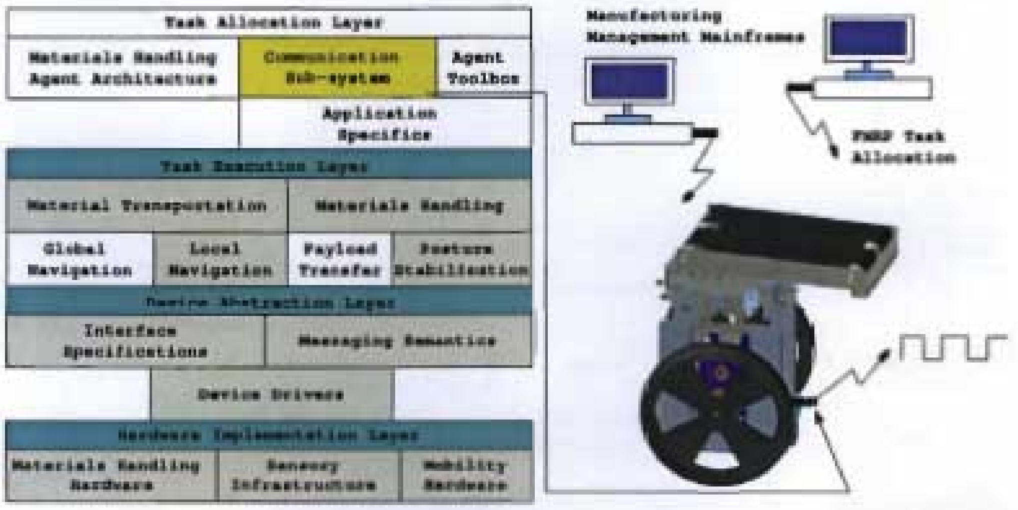


Figure 4.3: Communication Sub-System of the AMTS - For research purposes in a laboratory environment, IEEE 802.11b/g provides a relatively high bandwidth communication channel although in manufacturing environments, the 2.4GHz frequency range is often utilised along with Wireless Area Networks (WAN) which could impede the use of this frequency range for future materials handling systems

RollerMHP is accessible, in terms of global communication[1], through the IEEE 802.11b/g communication specification, commonly known as "Wifi", via a USB wireless network adapter on the on-board computer, Figure 4.2. The network scoped communication interface is a requirement in modern manufacturing environments where all manufacturing execution and supervision based data transfer occurs over Real-Time Ethernet (RTE) systems.

4.1.1.4 Functional Scope

RollerMHP's on-board computer serves two main purposes.

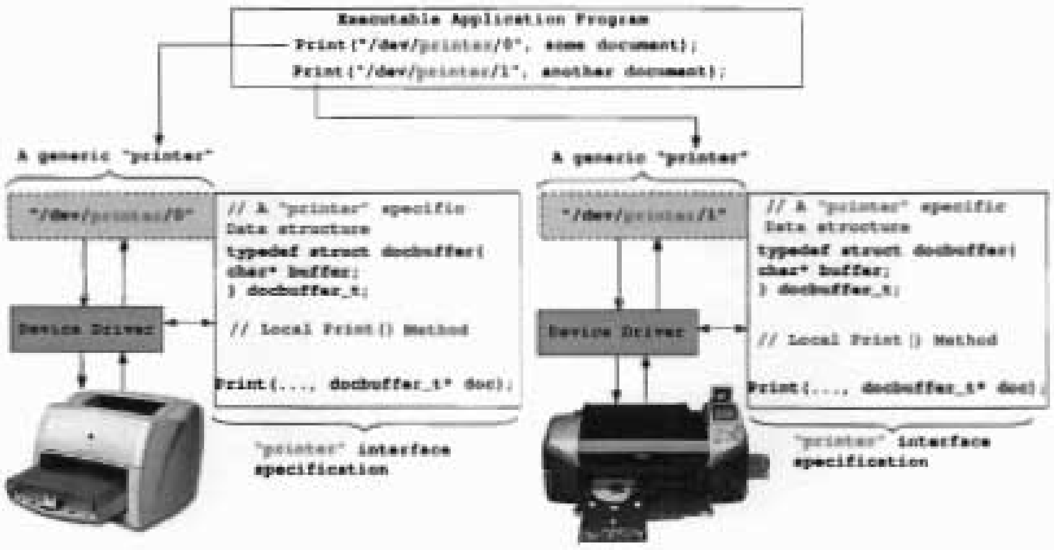
1. To operate and integrate RollerMHP's embedded systems, which form the low-level control framework for the differential drive and sensor systems, through serial communication streams into BrainStem® networks.
2. To house and operate an Open Source software system that allows the hardware specifics of RollerMHP to be abstracted onto generic network device interfaces, in a client-server access model. In this regard, the on-board computer acts as a "localhost" computer.

The software system used in implementing this functionality is introduced in section 4.2.

[1]Global communication is considered here as communication between external systems, such as manufacturing management mainframe computers, and the mobile materials handling and routing robot platforms

4.2 The Player Robot Device Interface

A Hardware Abstraction Layer (HAL) is a software construct used in Operating Systems to separate, or "hide", the hardware specifics of a computer system from the higher-level execution environment of software applications. HAL's perform this separation by utilising well defined generic software constructs, such as "mouse" or "printer", each having a standard interface. Higher-level software executables interact with these generic interfaces to access hardware through common messaging semantics and communication specifications. This allows application programs to avoid having to know the hardware details of a particular mouse, printer or any other hardware device, as long as each device adheres to the messaging constructs associated with the appropriate standard interface. The details of making a particular printer support the standard "printer" interface is handled by a software object called a device driver, Figure 4.4.



```
Executable Application Program
Print("/dev/printer/0", some document);
Print("/dev/printer/1", another document);
```

```
// A "printer" specific
Data structure
typedef struct docbuffer{
char* buffer;
} docbuffer_t;

// Local Print() Method

Print(..., docbuffer_t* doc);
```

```
// A "printer" specific
Data structure
typedef struct docbuffer{
char* buffer;
} docbuffer_t;

// Local Print() Method

Print(..., docbuffer_t* doc);
```

Figure 4.4: Hardware Abstraction Layer - A HAL forms part of the core functionality of all modern Operating Systems and is vital in providing homogeneity in the complex integrated environment of modern computing infrastructure

Through the implementation of HAL's, application programs can call methods under the facilities of, for example,

```
Print(''printer/0'', some document)
```

as opposed to,

```
Print_In_Some_Hewlett_Packard_Specific_Way(some document)
```

It is up to device drivers to translate generic methods and associated data structures, such as the hypothetical `Print(..., some document)` method and `docbuffer_t` data structure, into device specific low-level I/O communication to actually carry out the requested operation.

The Player Robot Device Interface, "Player", is a C/C++ based software implementation analogous to the HAL in an Operating System and forms part of the Player/Stage/Gazebo project [44]. Player defines a set of interfaces, each being a specification on the way in which user based software applications can interact with a particular class of robotic device. Player has been designed and developed to run on Operating Systems based on the POSIX specification[1], such as Linux, both PC and embedded versions, Solaris and BSD systems.

[1]Portable Operating System Based On Unix

4.2.1 Fundamental Concepts

Player operates in terms of three fundamental concepts in establishing its HAL functionality. In the most common form of implementation instance, these three elements, as a collective unit, are integrated in a client server model. An overview of each fundamental concept follows.

4.2.1.1 Interface

An interface is a software specification on how to interact with a certain class of robotic sensor, actuator, or algorithm. An interface defines the syntax and semantics associated with all messages that can be exchanged with entities in the same class. For example, the position2d interface is a specification on the message syntax and semantics associated with the class of robotic mobility devices that are capable of producing planar two dimensional motions, such as differential drive platforms. Each interface specification has been implemented as a set of C data structures and message codes, specific to the data management requirements and message semantics associated with a certain class of robotic devices, Figure 4.5.

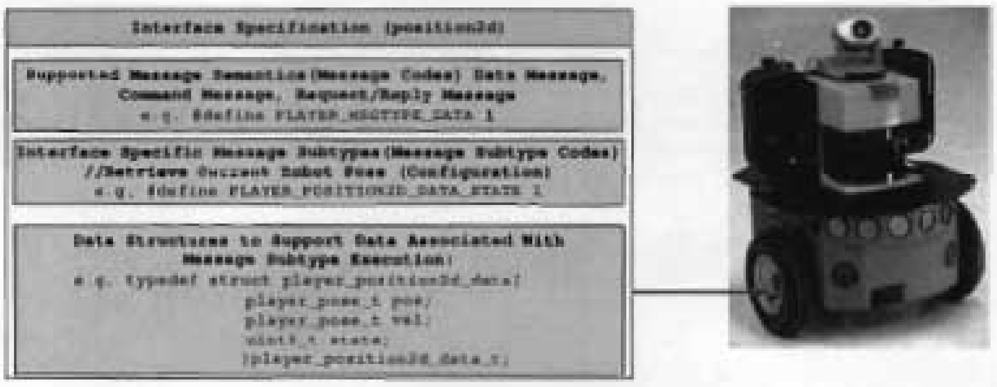

Interface Specification (position2d)
Supported Message Semantics(Message Codes) Data Message, Command Message, Request/Reply Message
e.g. #define PLAYER_MSGTYPE_DATA 1
Interface Specific Message Subtypes(Message Subtype Codes)
//Retrieve Current Robot Pose (Configuration)
e.g. #define PLAYER_POSITION2D_DATA_STATE 1
Data Structures to Support Data Associated With Message Subtype Execution:
e.g. typedef struct player_position2d_data{
player_pose_t pos;
player_pose_t vel;
uint8_t state;
}player_position2d_data_t;

Figure 4.5: Interface Specification - only a subset of the supported message semantic subtypes has been shown for clarity. For full insight into Player's interface specification, see [44]

Player specific message structures have been developed to facilitate message transfer between robotic devices operating under Players interface specification. An overview of the message concepts and Player specific data structures associated with their management is provided in section 4.2.3.

4.2.1.2 Driver

In the context of Player, a driver is a piece of software, usually written in C++, that communicates with a robotic sensor, actuator, or algorithm in such a way as to translate its inputs and outputs to conform with one or more interfaces. The driver, much like any other in the context of modern computing, hides the specifics of any given entity by making it appear to be the same as any other entity in its class, Figure 4.6.

4.2.1.3 Device

Player considers a device to consist of a driver bound to an interface, and provided a fully qualified address[1]. All message based communication occurs among devices, via interfaces. The drivers, while performing most of the work, are never accessed directly.

[1]The address structures are introduced during the introduction to Players client-server access model in section 4.2.1.4

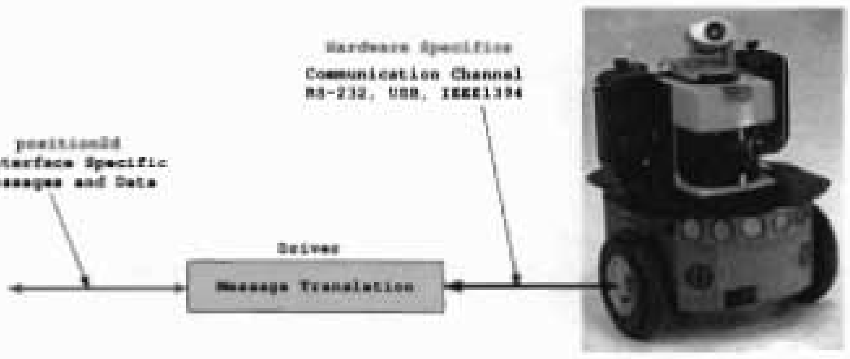


Figure 4.6: Driver Translation Functionality - Drivers are not always written for single interface support. Drivers can be written to support multiple interfaces. It is up to the driver designer to decide on the best driver implementation based on flexibility and or code efficiency

4.2.1.4 Player's Client-Server Model for Message Transfer

It has been established that Player provides HAL functionality for robotic system development, which allows higher-level software applications to utilise generic messaging constructs to communicate with and manipulate certain classes of robotic devices. The facilities providing the message transfer is the topic of this section, Figure 4.7.

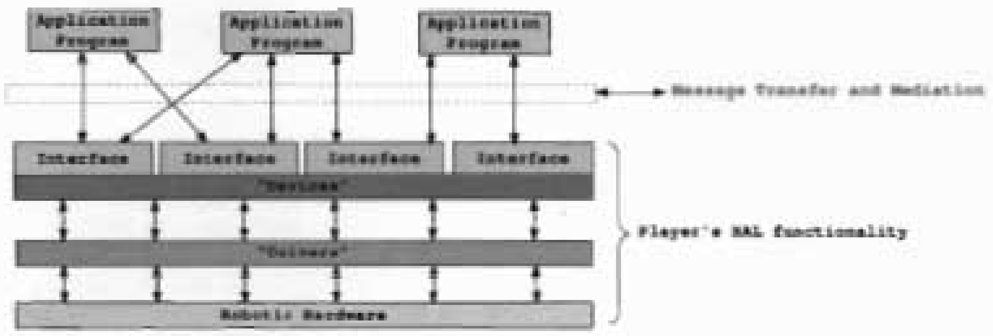


Figure 4.7: System Placement of Player's Message Transfer and Mediation Mechanism - Under Player's architecture, multiple application programs can communicate with the same device interface

To maximise message transfer distribution, Player has been designed to operate as a networked device server, and is often referred to as the Player server in this regard. Running on a host computer, the Player server mediates message transfer to and from devices. Client programs communicate with the Player server over a TCP socket and initiate a message transfer session through a subscription request protocol. Client programs use local proxy's on device instances in the Player server in order to send and receive messages. This completes the HAL functionality provided by Player. Drivers run inside the Player server, under device scope, often in multiple threads, and control their associated robotic device. Devices are assigned network addresses for reference and communication purposes which is implemented in Player as a 12 byte address field encapsulated in a C struct type definition, Table 4.1.
The Player server mediates all message transfer between client applications and address referenced devices, as well as inter-device subscriptions, Figure 4.8.
Client control applications are written in terms of generic device interfaces, therefore, control code written for one robotic device will, within reason, work on another as long as it supports

Players Device Address Structure

```
typedef struct player_devaddr {
        uint32_t host; // server host IP
        uint32_t robot; // Associated TCP socket
        uint16_t interf; // Interface code e.g. #define PLAYER_POSITION2D_CODE 4
        uint16_t index; // Specific device index
        } player_devaddr_t;
```

Table 4.1: Players 12 byte Address Structure - The index field is used to specify a particular device instance

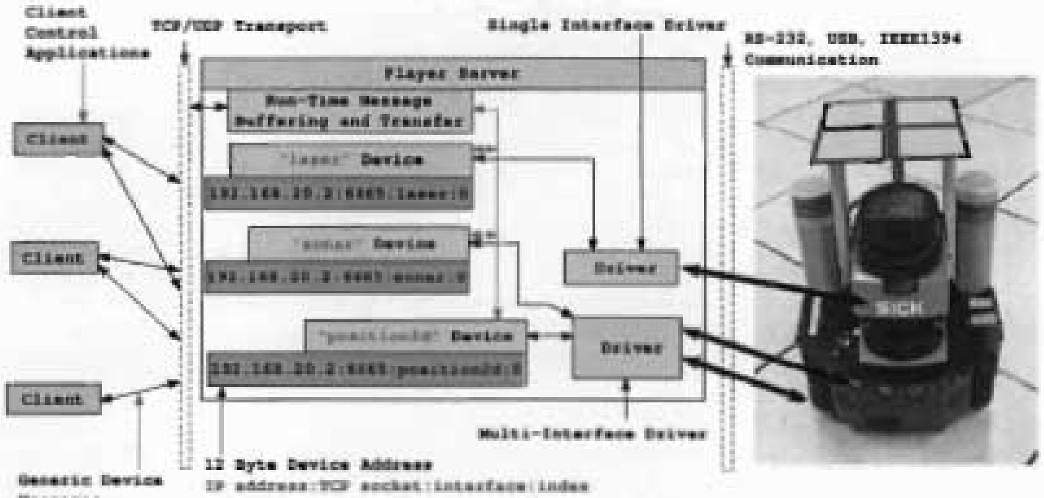

Client Control Applications
TCP/UDP Transport
Single Interface Driver
RS-232, USB, IEEE1394 Communication
Player Server
Run-Time Message Buffering and Transfer
"laser" Device
192.168.20.2:6665:laser:0
"sonar" Device
192.168.20.2:6665:sonar:0
"position2d" Device
192.168.20.2:6665:position2d:0
Client
Client
Client
Driver
Driver
SICK
Multi-Interface Driver
12 Byte Device Address
IP address:TCP socket:interface:index
Generic Device Messages

Figure 4.8: Client Server Architecture of the Player Robot Device Interface - Player was originally designed for implementation on Pioneer mobile robots [1]. The PatrolDX Pioneer robot is shown in the Figure

the same interface specifications. This code re-use and scalability makes Player the worlds leading Robot Device Interface [13].

4.2.2 Software Architecture

Player is composed of 5 main software libraries.

- **libplayererror**
  `libplayererror` is a C library that provides generic error reporting facilities such that application developers can utilise a common error message format for debugging purposes.
- **libplayercore**
  `libplayercore` is a C++ library that forms the main server kernel and messaging constructs. The library provides basic messaging and queuing functionality, a driver API to allow users to write drivers for their specific hardware needs, as well as support for loading plug-in drivers at run time.
- **libplayerdrivers**
  Driver support for some common hardware devices used in robotics research, such as the SICK LMS200 laser range finder, have been included as part of Player's source code. The `libplayerdrivers` library stores all such drivers, which can be included in the server at compile time. This is the static software context for drivers. A dynamic context exists as well in which driver code is loaded into the server as a shared object[1] at run time.
- **libplayertcp**
  The TCP transport mechanism associated with Players client-server model has been encapsulated in a separate C++ class object to separate the functionality of the Player server from explicit network transport particulars. This allows the server to focus on message mediation between device interfaces. Support for TCP client-server transport is provided by the `libplayertcp` library. The functionality of this library is covered in section 4.2.3.3.
- **libplayerxdr**
  The `libplayerxdr` library provides support for XDR data marshaling. Raw data messages sent over the TCP/IP network are formatted in accordance with the eXternel Data Representation (XDR) encoding specification [2]. This specification details the encoding of network data that is independent of the word size, byte-order (endian-ness) or other details of any particular computer architecture. XDR specifies a set of data types, e.g. int, float, char, and encodings for them. `libplayerxdr` is a C library that has been designed to facilitate this encoding and decoding specification, such that application programmers can avoid the inevitable software bugs associated with having to write "marshaling/demarshaling" code. Player specific C structures are transformed into their XDR representations automatically at run-time by the functions provided in the `libplayerxdr` library.

Note: From this point on, reference will be made to Player's C++ class constructs in order to provide scope on the servers run-time functionality and driver API. Without reference to the functionality provided by each C++ class object, comprehension of Player's main operating structure may be to a lesser degree.

4.2.3 Messaging Basics

The transfer of data, encapsulated in Player specific message structures, between client applications and devices housed in the server occurs over a TCP/IP network. The messaging constructs used in this context is the topic of the following sections.

[1] Other names for these file executable formats include shared library or dynamic link library (DLL) in the context of the windows operating system

4.2.3.1 Raw Player Messages

Player developers have included a particular message semantic and packaging structure that requires the use of a "message header". All raw Player messages sent over the TCP/IP network are preceded by a message header. The message header itself has been implemented as a C struct type definition, Table 4.2.

Players Message Header Structure

```
typedef struct player_msghdr {
        player_devaddr_t addr; //Destined device address, host:port:interf:index
        uint8_t type; //Message type (Data, Command, Request/Reply
        uint8_t subtype; //Interface specific message code
        double timestamp; //seconds since epoch
        uint32_t seq; //message book keeping
        uint32_t size; //size in bytes of the message payload
        } player_msghdr_t;
```

Table 4.2: Players Raw Message Data Structure - Under Players current implementation, message payloads are limited to 8MB in length which is a considerably large message

A message header keeps track of a destined device address and acts as an indicator, as to the message type and subtype, that follows the particular message header instance, such that the message processing facilities in the Player server can efficiently distribute the message to the correct device.

4.2.3.2 Server Scoped Message Encapsulations and Queuing Facilities

Inside the Player server, all raw messages received through a TCP socket over an IP network are automatically encapsulated in `Message` objects[1]. All Player messages are transferred between devices, and their associated drivers[2], as pointers to `Message` objects. Reference semantics on data transfer has been used so that messages can be delivered to multiple recipients with minimal memory overhead. `Message` objects are delivered by being pushed on and popped off `MessageQueue` objects. Each actual low-level driver code executable[3] running in the server has its own integrated `MessageQueue` object, which receives any messages sent to its associated device interface/s.

`MessageQueue` objects support configurable message replacement. This functionality is useful when a driver requires new incoming messages of the same semantic, i.e. type and subtype, to overwrite old ones, e.g. the message queue of a driver supporting the `position2d` interface must behave in this way under velocity command messages. Other driver implementations, such as those controlling robotic manipulators would configure their `MessageQueue` object to queue incoming messages of the same semantic in a FIFO manner, in order to correctly perform forward and inverse kinematics for manipulation purposes. Messages are stored in an associated queue through the use of a `MessageQueueElement` object, which provides a linked list encapsulation of a `Message` object, Figure 4.9.
There are two `MessageQueue` operating contexts in Player's server implementation. One queue context is associated with `Driver` object implementations and acts as a drivers local message buffer, and the other with client server subscriptions. The creation of client scoped message queues is discussed in section 4.2.3.3.

[1]This raw message packaging facility is provided by the `PlayerTCP` object and is covered in section 4.2.3.3
[2]The device-driver collective is discussed under section 4.2.4
[3]Generally in the form of shared library

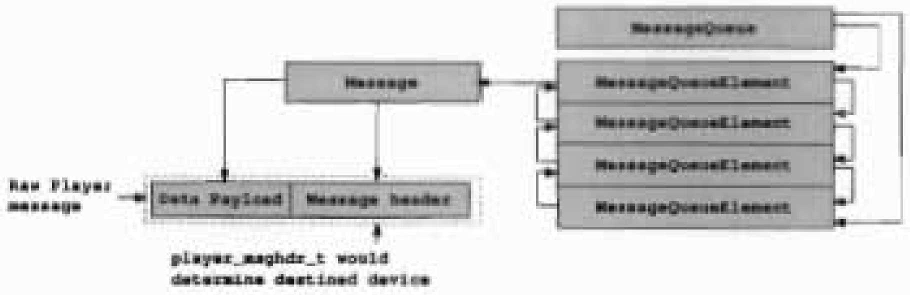


Figure 4.9: Graphical Representation of Message and MessageQueue Software Object - In the figure, a raw Player message is a newly XDR-demarshaled network representation of C data structures representing the interface specifications associated with a certain class of robotic device preceded by an instance of the Player message header data structure

4.2.3.3 Client-Server Message Transfer

This section covers the functionality provided by the `libplayertcp` library in providing message transfer between a TCP socket interface and Player's server implementation, recall section 4.2.2. Player has been designed such that the network transport layer associated with the client-server model is completely hidden from the user. This means that driver authors do not have to explicitly take TCP communication specifics into account when developing drivers. The `PlayerTCP` object forms part of the run-time infrastructure of Player's server implementation. The server keeps reference to a `PlayerTCP` object which handles the run-time creation of client associated message queues in the server, and wraps raw network packets into `Message` objects for pushing onto and popping off of message queues.

When a client application subscribes to the server, on a particular TCP socket, the `PlayerTCP` object handles the creation of the `MessageQueue` object in the server, which acts as a queue reference for that particular client-server message passing session. The server then establishes synchronisation messages with the client to keep data transfer temporally strict, Figure 4.10.

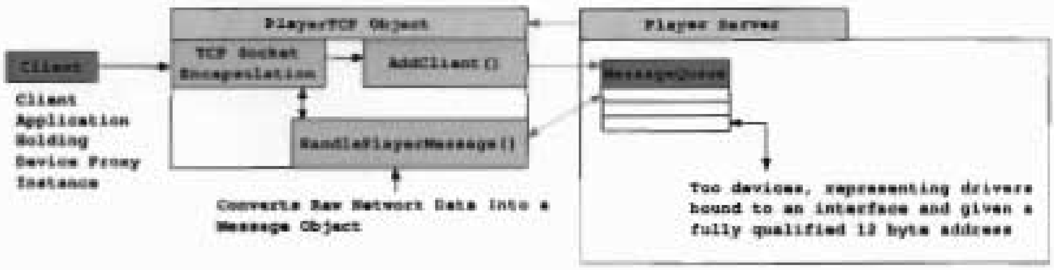

PlayerTCP Object
Player Server
Client
TCP Socket Encapsulation
AddClient()
MessageQueue
Client Application Holding Device Proxy Instance
HandlePlayerMessage()
Converts Raw Network Data into a Message Object
Too devices, representing drivers bound to an interface and given a fully qualified 12 byte address

Figure 4.10: Run-Time Instance of a Client Server Subscription - The `AddClient()` method actually handles the creation of the clients `MessageQueue` object and performs the required book keeping on the number of subscribed client applications

The `PlayerTCP` object in this sense acts as a black-box between Player server specific `Message` objects and network scoped, XDR encapsulated C data structures, representing Player's message headers and interface specifics. Each client application, currently subscribed to the Player server, has an associated `MessageQueue` object in the server, which the server uses to send messages between devices currently subscribed on the clients message queue.

4.2.3.4 Server-Driver Message Transfer

When a client application creates a proxy on a particular device instance, the server will try to subscribe the clients `MessageQueue` object to an associated `Device` object's client queue reference. `Device` objects encapsulate the servers implementation of devices, which represent an interface-driver collective with an associated 12 byte device address, recall Table 4.1. Recall that drivers are never accessed directly in the operating context of the Player server, even though they are explicitly involved in processing all messages sent to their associated devices.

As far as message transfer is concerned, `Device` objects act as forwarding mechanisms for messages sent to them by one of their subscribed client message queues. `Device` objects have reference to their underlying `Driver` object's `MessageQueue` object, and forward all messages sent to them onto the `Driver` object's input `MessageQueue` object, Figure 4.11.

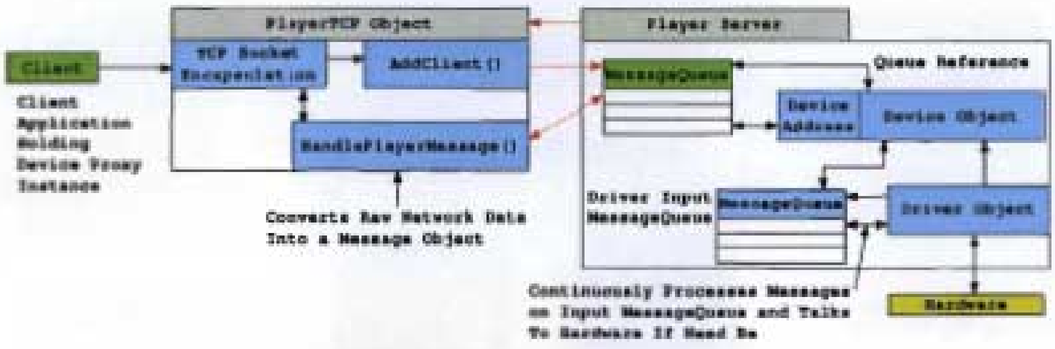

PlayerTCP Object
Player Server
Client
TCP Socket Encapsulation
AddClient()
Client Application Holding Device Proxy Instance
HandlePlayerMessage()
Converts Raw Network Data Into a Message Object
MessageQueue
Queue Reference
Device Address
Device Object
Driver Input MessageQueue
MessageQueue
Driver Object
Continuously Processes Messages on Input MessageQueue and Talks To Hardware If Need Be
Hardware

Figure 4.11: Device Specific Message Forwarding -

A `Driver` object's input `MessageQueue` object is the most important element with respect to driver code implementation. The driver API is described in section 4.2.4.

4.2.4 Players Driver API

For Player non-developers, i.e. the user community, the driver API is the most important aspect of Player. The driver API allows users to write C++ drivers to provide device support for their own specific hardware needs. User developed drivers are compiled into shared libraries to act as plug-in software modules, which the server loads during run-time through the use of configuration files[1] and a `ConfigFile` object. A `ConfigFile` object is included as part of Player's `libplayercore` library in order to handle the parsing of configuration files, passed into the server executable at run-time, in order to find and load user developed plug-in drivers.

In the operating context of the Player server, a driver must perform two main tasks.

1. Abstract the utilisation specifics of a hardware device behind one or more of Player's interface specifications.
2. Actively process Player specific messages that arrive on its associated input `MessageQueue` object.

Player developers have included the `Driver` class as part of Players driver API in order to allow users to readily create drivers to support their specific hardware needs.

The `Driver` class includes all the base functionality to interface with the message passing run-time environment of the Player server. The main thread of a driver sits in an infinite loop and continuously processes messages that arrive on its input message queue. Driver authors,

[1] Configuration files are used to tell the server executable and associated `ConfigFile` object where to find the plug-in driver. The Configuration file used for RollerMHP is shown in Appendix D.2

i.e. users, implement their specific driver through inheritance with the base `Driver` object and re-implement key methods in order to support their specific hardware device. The Player distribution and official website includes documentation with multiple examples on how to go about writing plug-in drivers [44].

4.3 Driver Development for RollerMHP

Using Players driver API and the BrainStem® C API access libraries for Linux, drivers were written to support RollerMHP's hardware specifics under Players interface specifications in order to facilitate the Device Abstraction Layer of the AMTS. Child `Driver` classes[1] were written that wrapped the necessary routines of the BrianStem® C API in C++ member functions, Figure 4.12.

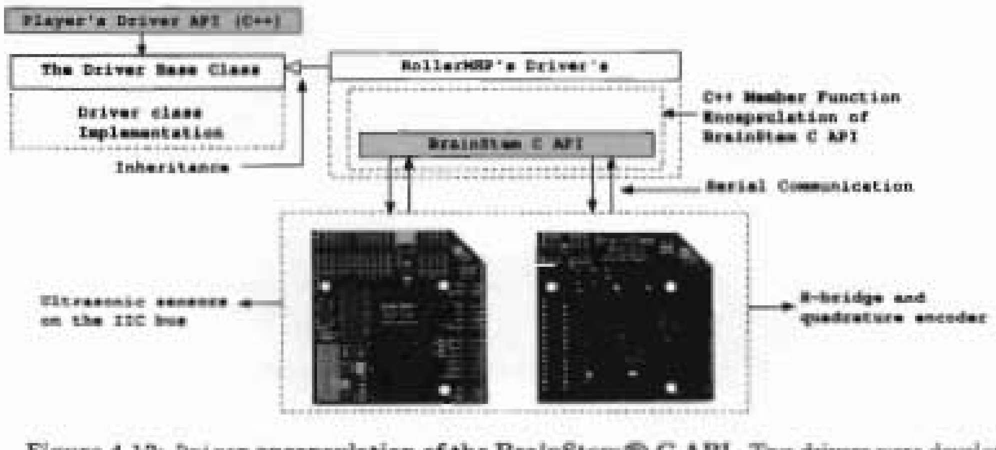

Player's Driver API (C++)
The Driver Base Class
Driver class Implementation
Inheritance
RollerMHP's Driver's
C++ Member Function Encapsulation of BrainStem C API
BrainStem C API
Serial Communication
Ultrasonic sensors on the IIC bus
H-bridge and quadrature encoder

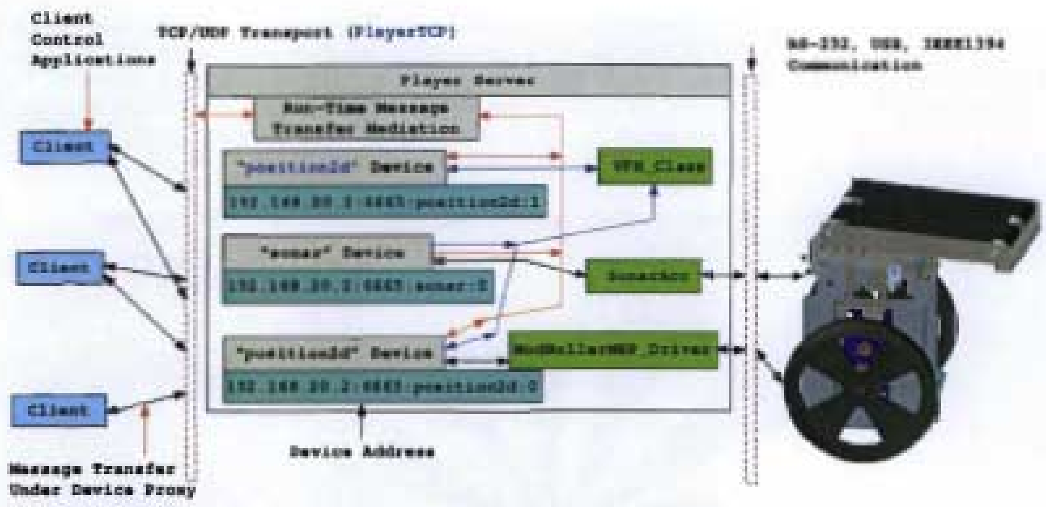
Figure 4.12: `Driver` encapsulation of the BrainStem® C API - Two drivers were developed to support RollerMHP in Player's operating framework

4.3.1 Supported Interface Specifications

Two main drivers were written in C++ and compiled into shared libraries under Player's compilation specifics in order to act as plug-in objects for dynamic loading into Player's server implementation at run-time. With the current driver implementations, RollerMHP supports the `position2d` and `sonar` interfaces[2]. Two separate drivers were written for RollerMHP's hardware as this software practice provides a one to one mapping between the bulk driver code and the supported interface, which produces cleaner and more maintainable code.

The `ModRollerMHP_Driver` driver and the `SonarAcc` driver, see Appendix C, provide the necessary software infrastructure to get RollerMHP up and running in the context of Player's server implementation, Figure 4.13.
The `ModRollerMHP_Driver` driver supports the `position2d` interface specification and allows RollerMHP's differential drive platform to appear as a generic `position2d` device over an IP network. Among the vast functionality provided by the `ModRollerMHP_Driver` driver, it allows client applications to control RollerMHP's translational and rotational velocity and receive configuration updates according to the Runge-Kutta odometry implementation, recall Algorithm 1 of section 3.2.4.3. The `SonarAcc` driver supports the `sonar` interface specification and abstracts RollerMHP's embedded active sensor system behind an array of generic sonars over an IP network. The `SonarAcc` driver is currently being extended to support the `actarray` interface, [5], to allow the materials handling conveyor to be abstracted behind an array of generic linear and

[1]In C++, a child class is a class which inherits from another class
[2]For full insight into Player's interface specifications, see [44]

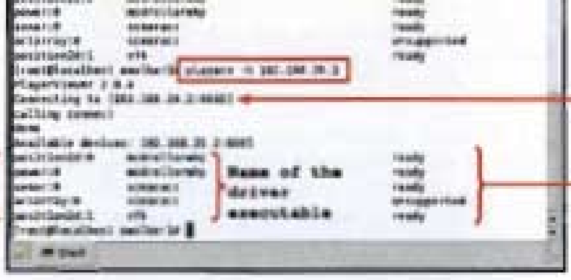


Client Control Applications
TCP/UDP Transport (PlayerTCP)
RS-232, USB, IEEE1394 Communication
Player Server
Run-Time Message Transfer Mediation
Client
"position2d" Device
192.168.20.2:6665:position2d:1
VFH_Class
Client
"sonar" Device
192.168.20.2:6665:sonar:0
SonarAcc
"position2d" Device
ModRollerMHP_Driver
192.168.20.2:6665:position2d:0
Client
Device Address
Message Transfer Under Device Proxy

Figure 4.13: Player's Server Implementation for RollerMHP - The position2d device at index 1 forms the device interface of an "abstract" driver which uses other drivers as sources of data and sinks for commands instead of actual hardware. This is covered in chapter 5

rotary actuators. This will allow for the development of generic forward and inverse kinematic code for the materials handling infrastructure on RollerMHP.

Player comes with a few general purpose client applications for testing and debugging driver code. PlayerViewer is one such client application, which provides a GUI interface and proxy manifold to a range of generic devices over an IP network. During preliminary testing and debugging of the ModRollerMHP_Driver and SonarAcc driver code, the PlayerViewer client application was used to check the functionality of the drivers and debug the code if need be[1]. A screen shot of PlayerViewer's shell output is shown in Figure 4.14.



Session Edit View Bookmarks Settings Help
Server subscription request
Available devices in the server
Name of the driver executable
Status of the driver executable
Shell

```
position2d:1      vfh                                  ready
[root@localhost [illegible]]# playerv -h 192.168.20.2
PlayerViewer 2.0.4
Connecting to [192.168.20.2:6665]
calling connect
done
Available devices: 192.168.20.2:6665
position2d:0      [illegible]                          ready
power:0           [illegible]                          ready
sonar:0           [illegible]                          ready
actarray:0        [illegible]                          unsupported
position2d:1      vfh                                  ready
[root@localhost [illegible]]# playerv -h 192.168.20.2
PlayerViewer 2.0.4
Connecting to [192.168.20.2:6665]
calling connect
done
Available devices: 192.168.20.2:6665
position2d:0      [illegible]                          ready
power:0           [illegible]                          ready
sonar:0           [illegible]                          ready
actarray:0        [illegible]                          unsupported
position2d:1      vfh                                  ready
[root@localhost [illegible]]#
```

Figure 4.14: PlayerViewer's Shell Output - The shell output messages show the available devices in the server, the name of the driver providing the message translations and the status of the device

[1]If need be is an under-statement, any code of considerable complexity is bound to have a few bugs. The many books on debugging source code written by highly respected software developers is evidence of this fact

In the shell output, the PlayerViewer client application has been executed under subscription to a server at location 192.168.20.2 on the TCP socket 6665, which represents the IP address of RollerMHP's on-board computer, from one of the spare Linux computers in the laboratory. The actual communication occurs over a wireless Ethernet link, facilitated by two IEEE 802.11b/g USB network adapters. As far as client applications are concerned, such as PlayerViewer, RollerMHP is seen as a combination of `position2d` and `sonar` devices, Figure 4.15.

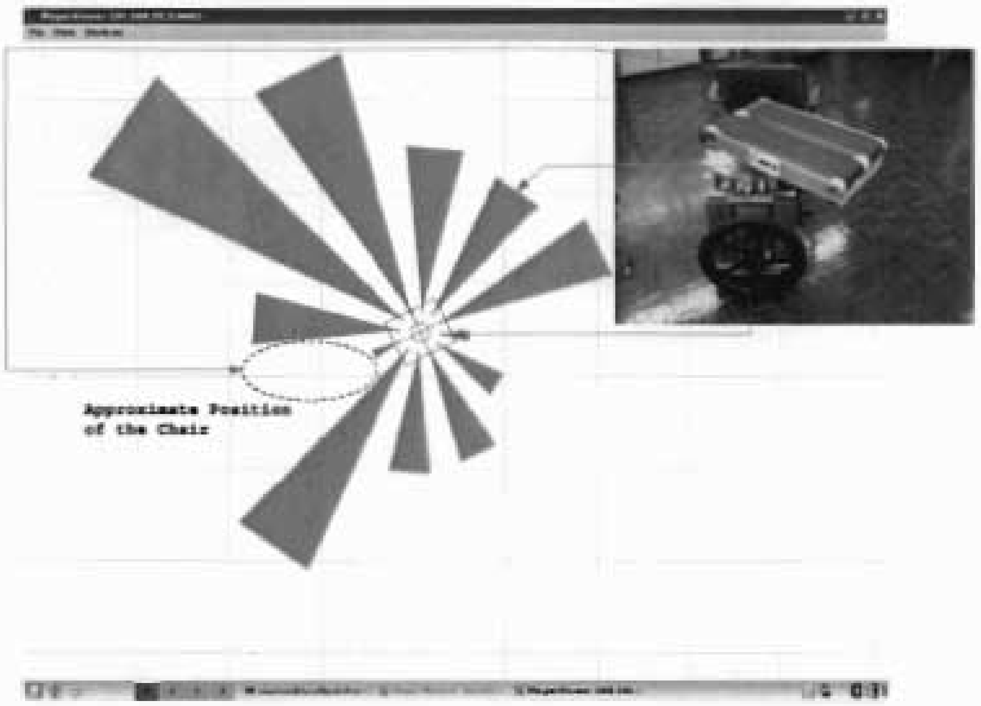


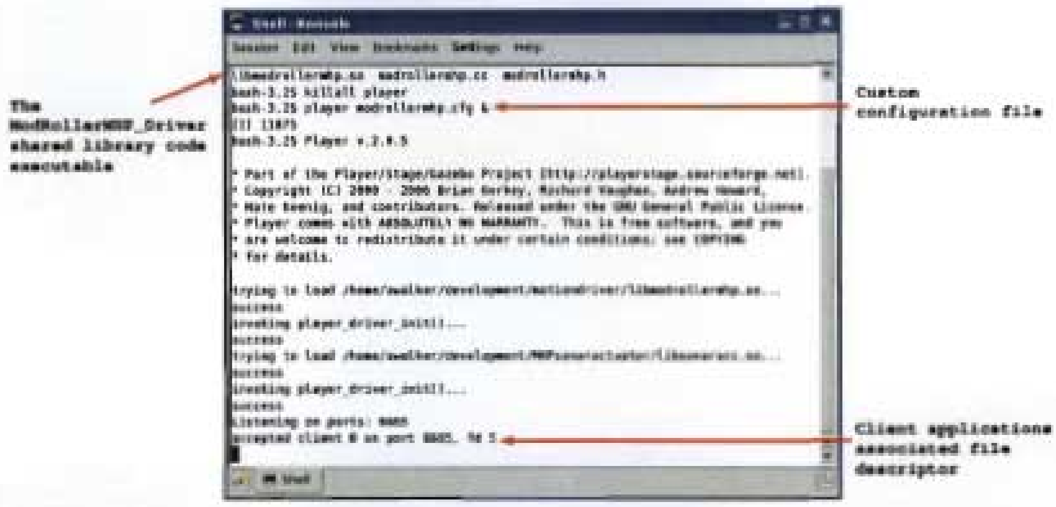
Figure 4.15: PlayerViewer Screen-shot of RollerMHP's Device Instances - The sonar sensor model used in PlayerViewer models the beam of an ultrasonic sensor as a conical geometry, which is not a true representation of the actual beam geometry. Under most operating conditions though, this is a reasonably accurate assumption

To show what this client application subscription looks like from the server side of message transfer session, RollerMHP's onboard computer was connected to a computer screen and the server instantiated on a custom configuration file, which essentially tells the server and associated `ConfigFile` object where to find the plug-in driver code in Fedora's file system, and which interfaces the driver supports, Figure 4.16.

As can be seen in Figure 4.16, after the server is up and listening on TCP socket 6665 it accepts its first client, client 0 representing PlayerViewer, which can then act through sending messages with device proxy's to the server to manipulate RollerMHP via its exposed `position2d` device and receive RollerMHP's active sensor data via its exposed `sonar` device.

4.4 Chapter Summary

HAL's are vital components in providing a homogeneous operating platform for application programs. These HAL's allow application programs to communicate with and operate on generic devices without having to know detailed knowledge about the specifics associated with a device's underlying hardware, such as the hardware implementation of desktop printers. HAL's,


The ModRollerMHP_Driver shared library code executable
Custom configuration file
Client applications associated file descriptor

```
libmodrollermhp.so  modrollermhp.cc  modrollermhp.h
bash-3.2$ killall player
bash-3.2$ player modrollermhp.cfg &
[1] 11875
bash-3.2$ Player v.2.0.5

* Part of the Player/Stage/Gazebo Project [http://playerstage.sourceforge.net].
* Copyright (C) 2000 - 2006 Brian Gerkey, Richard Vaughan, Andrew Howard,
* Nate Koenig, and contributors. Released under the GNU General Public License.
* Player comes with ABSOLUTELY NO WARRANTY.  This is free software, and you
* are welcome to redistribute it under certain conditions; see COPYING
* for details.

trying to load /home/swalker/development/MotionDriver/libmodrollermhp.so...
success
invoking player_driver_init()...
success
trying to load /home/swalker/development/MHPsonaractuator/libsonaracc.so...
success
invoking player_driver_init()...
success
Listening on ports: 6665
accepted client 0 on port 6665, fd 5
```

Figure 4.16: Server's Shell Output for PlayerViewer Client Subscription - The `PlayerTCP` object keeps track of subscribed clients using file descriptors. The PlayerViewer client is seen as `fd 5` by the server's `PlayerTCP` object

as a software construct, provide a generic interface between physical hardware and the application programs requesting resources from them. This is achieved through the construction of interface specifications which represent the software abstraction framework for a particular class of computer hardware devices, such as "mouse" or "printer". Device drivers provide the mapping between the hardware specifics of a device and the associated interface specification.

Player provides this functionality for robotic devices, thus allowing the development of generic control code, which provides software scalability and promotes code re-use, a vital requirement in any modern software system. Player's driver API was used, along with Player's interface specifications, to abstract the hardware specifics of RollerMHP, i.e. the BrainStem® modules and connected hardware devices, behind generic interfaces.

Player has been designed to facilitate message transfer and communication between application programs and device abstractions, through a client-server model. This provides devices with network scope and makes them accessible over a TCP/IP network. Application programs act as clients and subscribe to Player's server implementation in order to initiate a message transfer session. These client applications hold local device proxy's to the server in order to send and receive messages to and from devices held in the server. Physical communication with RollerMHP occurs over a wireless Ethernet connection facilitated by IEEE 802.11b/g USB network adapters.

Two drivers were written to provide Player specific HAL functionality and support for RollerMHP in Player's client-server operating model. The `ModRollerMHP_Driver` driver supports the `position2d` interface and allows RollerMHP's underlying differential drive platform and embedded system to appear as a generic `position2d` device capable of configurations on $\vec{q} = \mathbb{R}^2 \times SO^1$. The `SonarAcc` driver supports the `sonar` interface and abstracts RollerMHP's ultrasonic sensor platform and embedded system behind an array of generic `sonar` devices.

Chapter 5

Task Execution Layer

''To isolate mathematics from the practical demands of the sciences is to invite the sterility of a cow shut away from the bulls.'' -Pafnuty Lvovich Chebyshev

Chapter 4 covered the software system, namely Player, used in providing the infrastructure to develop generic robotic control frameworks by abstracting hardware specifics onto well defined software interfaces and messaging constructs. Player provided client-server mediated control of the abstract "devices" constituting RollerMHP's physical configuration, over an IP network.

This chapter covers the development, testing, and validation of the motion control algorithms, designed in the scope of Player's interface specifications, for RollerMHP, to facilitate the motion requirements underlying the Task Execution Layer of the AMTS.

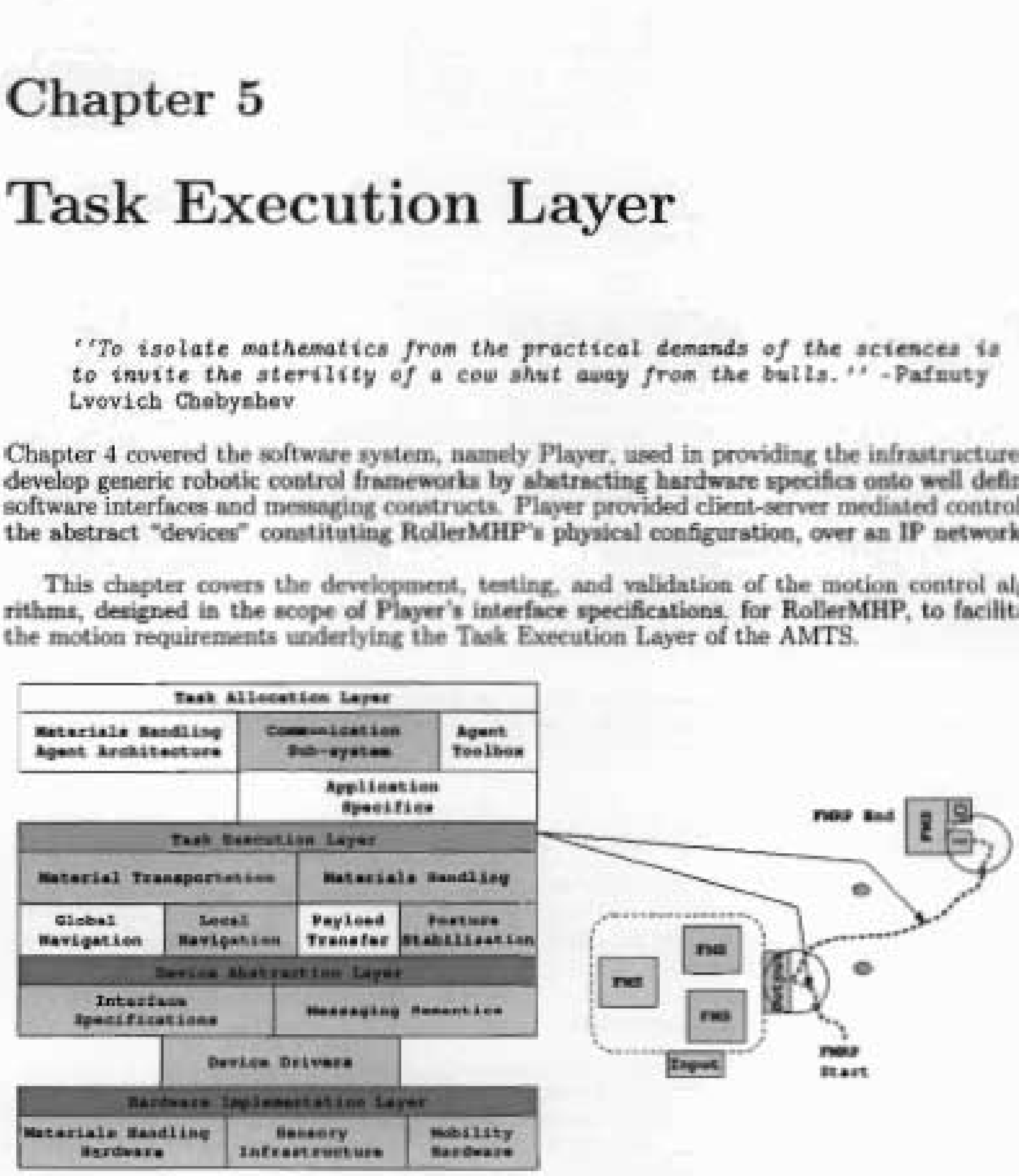


Figure 5.1: Task Execution Layer - This chapter only covers generic aspects of the Task Execution Layer. This is due to the highly application specific domains of modern production environments

Facilitation of a FMRP task requires study into global and local navigation, as well as posture stabilising motion control of the physical robot platforms executing FMRP task instances. It is therefore only natural that much effort is spent in researching, developing, and testing the motion control aspects associated with material payload transportation. To this extent, this chapter holds the majority of this dissertations research scope.

5.1 Material Transportation

Depending on the complexity of the payload, materials handling, i.e loading and unloading operations, can form the majority of the materials handling and routing task duration. Although this aspect would suggest placing emphasis on optimising materials handling operations, in modern dynamic production environments with highly distributed manufacturing infrastructure. gross material transportation forms a major aspect in the efficiency and real-time task tracking ability of mobile materials handling systems performing FMRP tasks, Figure 5.2.

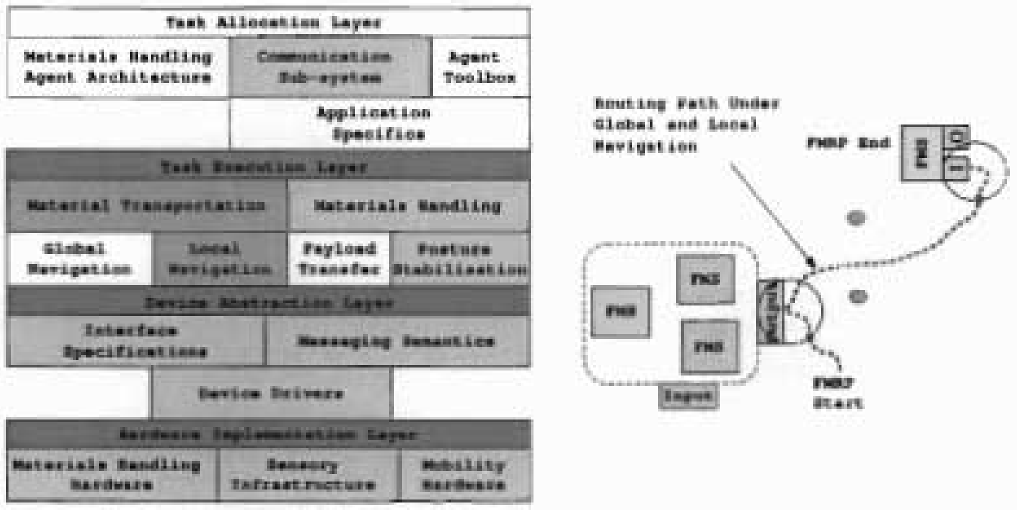


Figure 5.2: Material Transportation Aspect of FMRP Task Execution - Global navigation is essentially a path planning problem and is highly dependent on the environment in which the mobile robot platform is operating in. It is therefore not explicitly covered here

5.1.1 Navigation Preliminaries

The definition of a FMRP provided in section 2.3.2.3 defined a Transportation Primitive as a global navigation operation between two distributed, unconnected manufacturing infrastructure subsets[1], while avoiding obstacles in a real-time manner. An brief overview of navigation operations follows in section 5.1.1.1.

5.1.1.1 Global Navigation Overview

The performance of global navigation algorithms, implemented on mobile robot platforms, is highly dependent on the amount of structural information, regarding the mobile platforms operating environment, available for concurrent analysis and decision making. Decision making is meant in the context of finding optimal paths between navigation endpoints under analysis of the structural layout of the operating environment.

Intuitively, in order for a mobile robot platform to autonomously perform global navigation operations, it must be provided, or create, a map of its working environment as well as know its location within the map. Mobile robot platforms perform localisation to determine their position, or more precisely their pose or configuration in a map. This is achieved by correlating structural information about their surrounding environment with structural information, such as distinct landmarks or waypoints in the map provided, through sensory perception, data

[1]Unconnected in the sense that no standard materials handling infrastructure joins the manufacturing infrastructure subsets

fusion, and data analysis. Map representations can be in many forms, from complex three dimensional structural models to minimal two dimensional bitmap representations of the environment. Once localised, a mobile robot has the base knowledge to perform path planning and global navigation operations.

In recent years gracious research efforts have been applied in solving the Simultaneous Localisation and Mapping (SLAM) problem [17]. The SLAM problem addresses whether it is possible for a mobile robot platform, placed in an unknown environment, to build a map of its surrounding environment through structural geometric development of sensory perception outputs, while concurrently localising in the map. Although multiple solutions to the SLAM problem have been developed, the "loop closure" problem still makes an appearance due to the ever present uncertainty in sensor performance [29]. Due to the structured layout of production environments, it would seem unorthodox to expect mobile materials handling robots to perform SLAM. In production environments, the loop closure problem would not be tolerated and production rates would surely drop. Maps provided a priori accompanied by active measures of localisation under the use of active beacon technologies [42] would suffice in providing the necessary information and knowledge to enable mobile materials handling and routing robot platforms to perform global navigation operations.

Due to the highly application specific nature of global navigation operations, research efforts during this project, were focused on local navigation and obstacle avoidance in facilitating a subset of the material transportation requirements of FRMP task execution.

5.1.1.2 Local Navigation Overview

Local navigation includes both obstacle avoidance as well as local decision making. In order to perform local navigation, a mobile robot platform must utilise sensory perception outputs and perform data fusion to extract enough structural information about the surrounding obstacles in order to, both avoid them, and move in the most feasible direction possible if more than one obstacle is present in its local surrounding environment. The aspect of decision making in cases of multiple obstacles must not be overlooked and can determine the overall performance of a local navigation algorithms and prevent the introduction of "local minimums" where the mobile robot gets trapped in areas with many surrounding obstacles during operation.

5.1.2 Player as a Code Repository

Although most drivers that run in the Player server directly control hardware, more recently, a number of "abstract" drivers have been developed which use other drivers as sources of data and sinks for commands. Many algorithms have been developed using these measures and accompany the Player source code. The "amcl" driver is one such abstract driver implementation and performs Adaptive Monte Carlo Localisation, which is a very popular particle filter based localisation algorithm used by modern mobile robots [16]. Many other abstract drivers accompany the Player source code including local and global navigation algorithms. RollerMHP uses a well known local navigation algorithm that comes with Player in the form of an abstract driver namely the Vector Field histogram (VFH) algorithm, and was compiled into the Player server as a static object at compile time. The Vector Field Histogram algorithm is a local navigation algorithm pioneered by Johann Borenstein [10]. The algorithm provides mobile robots with real-time obstacle avoidance, and was developed using mobile robots equipped with ultra-sonic sensor arrays for sensory perception, much like RollerMHP. The algorithm has many favourable characteristics that make it highly applicable in dynamic manufacturing environments.

A complete overview of the operating characteristics of the VFH algorithm can be found in [8][7][9]. A video of RollerMHP performing a local navigation operation can be seen on the CD accompanying this dissertation.

5.2 Materials Handling

From the definition provided in section 2.3.2.3. A Materials Handling Primitive requires the mobile materials handling and routing robot platform to perform posture stabilisation, in which the mobile robot platform converges onto a goal pose in the vicinity of an I/O port from an initial configuration on the boundary of a Region of Convergence (RoC).

Material transfer operations are application specific whereas posture stabilisation requirements are generic enough to warrant credible research. This section explicitly covers the research, development, and testing of the motion controllers implemented on RollerMHP in order to perform posture stabilisation within a RoC, Figure 5.3.

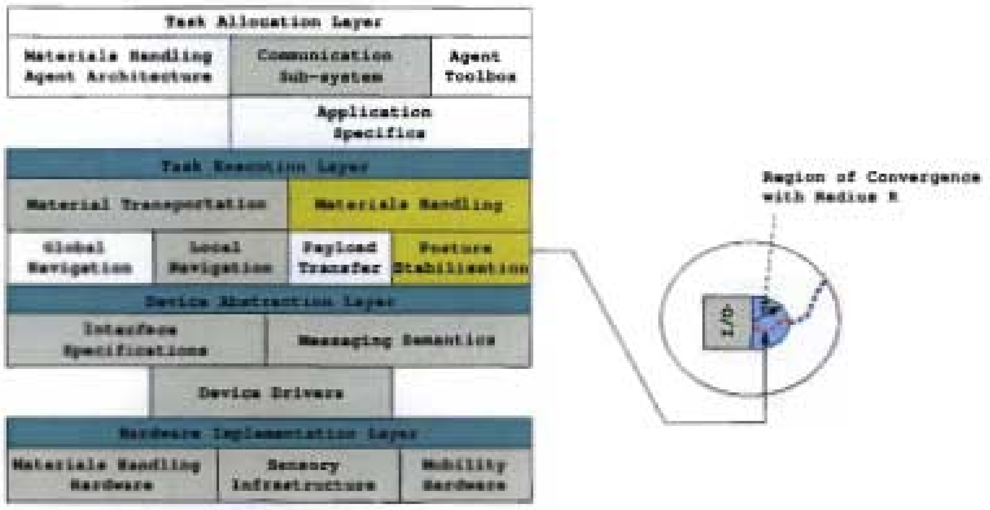


Figure 5.3: Materials Handling Aspect of FMRP Task Execution - The characteristics of, and requirements for, the payload transfer block, are sensitive to the application environment and are therefore not covered. This is done to focus research effort in more useful domains

5.2.1 Preliminary Control Concepts

Before the introduction of the posture stabilisation motion control problem, two fundamental concepts in control theory, stability and controllability, are introduced in order to make the motion controllers, introduced in following sections, more comprehensive. The following stability discussion is in terms of state-space concepts and it is assumed that all state vectors and control inputs are elements in $\mathbb{R}^n$ and $\mathbb{R}^m$ respectively, the space of all real numbers.

5.2.1.1 Stability

The notion of stability in control engineering is usually associated with the response characteristics of systems implementing feedback control. Given the vector valued state differential equation,

$$\dot{\vec{x}}(t) = f(\vec{x}(t), \vec{u}(t)), \qquad f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$$

representing the vector field of a control system model with state vector $\vec{x}(t) \in \mathbb{R}^n$ and control inputs $\vec{u}(t) \in \mathbb{R}^m$. The feedback control problem in this regard is to find some control law $\vec{u}(t) = g(\vec{x}(t))$, $g : \mathbb{R}^n \to \mathbb{R}^m$, such that the state of the system or "plant", $\vec{x}(t)$, converges onto some desired goal state $\vec{x}_g$, from an arbitrary initial state $\vec{x}(t_0) = \vec{x}_i$, as $t \to \infty$. The control engineering term given to the task of designing $g(\vec{x})$ is "feedback stabilisation" [40].

Under the application of $g(\vec{x})$, a fully qualified vector field[1] is produced. This fully qualified vector field, representing the image of $\dot{\vec{x}}(t) = f(\vec{x}(t), g(\vec{x}(t)))$, can be considered autonomous, in that its solutions to initial value problems in the form of contour integrals, representing state trajectories, evolve through time without further dependencies on $\vec{u}$.

In linear state-space control theory, i.e. systems models taking the form $\dot{\vec{x}}(t) = f(\vec{x}(t), \vec{u}(t)) = A(t)\,\vec{x}(t) + B(t)\,\vec{u}(t)$, with A and B being $n \times n$ and $n \times m$ matrices respectively, feedback stabilisation characterisation involves the application of eigenvalue analysis on $n \times n$ matrices incorporating linear operators[2], which is a well understood theory and method of feedback stability characterisation [35]. Nonlinear control theory however, is not nearly as well established as its linear counterpart. The feedback stabilisation of nonlinear control systems is usually a much more complex task than that associated with linear systems, along with the methods of stability characterisation. In nonlinear control theory, and applied mathematics for that matter, a particular type of stability theory exists termed Lyapunov Stability, named after the Russian mathematician, Aleksandr Mikhailovich Lyapunov. This is covered in section 5.2.1.2

5.2.1.2 Lyapunov Stability and Equilibrium Points

Lyapunov Stability theory characterises how the state trajectory of a dynamic system behaves in the local vicinity of a goal state, which is considered to be an equilibrium point of the state space, in that $\dot{\vec{x}}(t) = f(\vec{x}_g, g(\vec{x}_g)) = 0$, for the cases involving feedback stabilisation. Lyapunov Stability theory is an involved subject and this is only a brief overview of its associated concepts.

A goal state, $\vec{x}_g$ is said to be Lyapunov stable if for any open neighbourhood of $\vec{x}_g$, $\mathcal{N}_1 \subseteq \mathbb{R}^n$, i.e. an open region of state-space which includes $\vec{x}_g$, there exists another open neighbourhood, $\mathcal{N}_2 \subseteq \mathcal{N}_1$ of $\vec{x}_g$ such that for any initial state $\vec{x}(t_0) = \vec{x}_i \in \mathcal{N}_2$, the state trajectory[3] stays within $\mathcal{N}_1$, $\forall t > t_0$ [40]. Lyapunov stability does not specify weather the state trajectory eventually reaches the goal point as time approaches infinity. In order for this to occur, the dynamic system requires asymptotic stability characteristics.

A goal state $\vec{x}_g$ is asymptotically stable if it is Lyapunov stable, and further more $||\vec{x}(t) - \vec{x}_g|| \to 0$ as $t \to \infty$ [40]. Furthermore, a goal state $\vec{x}_g$ is exponentially stable if it is asymptotically stable and there exists constants α and β such that [40],

$$||\vec{x}(t) - \vec{x}_g|| \leq \alpha\, e^{-\beta t}\, ||\vec{x}(t_0) - \vec{x}_g||, \qquad \forall t > t_0$$

The full state feedback posture stabilising controllers developed and tested, to provide Roller-MHP with feedback stabilisability to a goal point it its configuration space $\vec{q}_g = [x_g, y_g, \theta_g]^T$, from an initial configuration $\vec{q}_i = [x_i, y_i, \theta_i]^T$, are all asymptotically stable.

5.2.1.3 Controllability

Consider once again, all systems described by the vector valued state differential equation in the form,

$$\dot{\vec{x}}(t) = f(\vec{x}(t), \vec{u}(t)), \qquad f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$$

Systems of this form are termed controllable if for all $\vec{x}_i, \vec{x}_g \in \mathbb{R}^n$, there exists a control input trajectory $\mathcal{U} = \vec{u}(\tau)$ for $\tau \in [t_0, t]$, such that in integrating the vector field, $\dot{\vec{x}}(\tau) = f(\vec{x}(\tau), \mathcal{U})$ from $\vec{x}(t_0) = \vec{x}_i$, the value of the integral curve, or state trajectory, is $\vec{x}(t) = \vec{x}_g$. Simply put, a system is controllable if there exists a state trajectory between an initial state, $\vec{x}_i$ and a goal state x_g. Controllability is returned to when discussing the control properties associated with posture stabilising differential drive platforms in section 5.2.2.4.

[1] Fully qualified in the sense that the vector field is independent of $\vec{u}$

[2] Such as the complex variable $s = \sigma + j\omega$

[3] In the case considered here, the solution in the form of a contour integral for an initial value problem of a vector valued state differential equation

5.2.1.4 Posture Stabilisation

Posture stabilisation of practical physical systems is a control engineering problem, and in the case of differential drive platforms, a nonlinear control problem. It is important to understand the required control efforts in implementing posture stabilisation, through feedback stabilisation, on differential drive platforms. In this regard, posture stabilisation consists of regulating relatively small configuration errors[1] in the differential drive's configuration space. Posture stabilisation of differential drives requires the development of, either feed-forward control laws, or feedback control laws. Feed-forward control implementation is strictly a trajectory planning problem, in which a control input trajectory, $\mathcal{U}$, is developed, representing temporally dispersed translational and rotational velocity commands, and applied to the differential drive platforms physical control infrastructure in order to move the platform along a pre-determined trajectory that joins an initial configuration and goal configuration. Feedback control laws are, by their very nature, more robust, but face serious mathematical obstructions. Posture stabilising motion controllers designed to overcome these obstructions can produce unrealistic and unsatisfactory transient responses [6]. An overview of the unique control properties of differential drives follows in section 5.2.2.

5.2.2 Modeling and Control Properties

In this section, the kinematic model of a differential drive platform is implicitly exposed through explicit coverage of the constraints on its configuration velocity. Eq. 3.4. Deriving the kinematic model in this way provides greater insight into the feedback stabilisation properties of differential drive platforms.

5.2.2.1 The Configuration Space

It is important to visualise motion trajectories in terms of configuration spaces[2] in order to appreciate the constraints imposed by the wheel-floor interface of differential drive platforms. Selecting the same generalised coordinates as those of section 3.2.2.3 to describe a differential drive platform's configuration, its configuration space can be visualised as a topological space, Figure 5.4

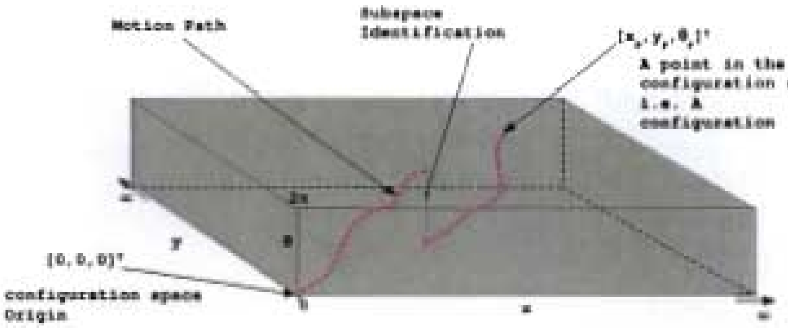


Figure 5.4: Differential Drive Configuration Space - This topological space has an identity on the orientation coordinate. Identities are useful in path planning and can provide optimal solutions to path planning problems

The topological identity allows configuration trajectories to "wrap around" the configuration space and are useful in motion planning. Posture stabilisation implementation relies on understanding the constraints imposed on generalised configuration velocities, such as Eq. 3.4. and its associated controllability. An overview of under-actuated systems is presented in section 5.2.2.2.

[1] Relatively small in that the configuration error is within some bounded region around the origin of the differential drives configuration space. The configuration error is infact a relative term, as the goal point, which is essentially arbitrary, is considered the origin of the configuration space

[2] State space concepts are analogous to the configuration space

5.2.2.2 Under-actuated Mechanical Systems

An under-actuated mechanical system is one in which there are a smaller number of available control inputs than independent generalised configuration coordinates. Under-actuated systems arise due to some form of non-integrable motion constraint. In such systems, it is impossible to choose a set of generalised configuration coordinates equal to the number of degrees-of-freedom (dof) for the mechanical system. The number of generalised configuration coordinates, also known as Lagrangian coordinates, exceeds the number of dof by the number of non-integrable motion constraints [28]. Such mechanical systems are called nonholonomic.

Differential drive platforms are nonholonomic as they have $n = 3$ generalised configuration coordinates $\vec{q} = [x, y, \theta]^T \in \mathbb{R}^2 \times SO^1$ but only $m = 2$ available control inputs $\vec{u} = [v, \omega]^T$, which are practically realised through the control of the left and right drive wheel angular velocities. The non-integrable differential constraint imposed on a differential drive platform's generalised configuration velocity $\dot{\vec{q}}$ is due to the rolling without slipping condition exhibited by the active drive wheels, section 5.2.2.3.

5.2.2.3 Differential Constraints

All Pfaffian[1] nonholonomic systems are characterised by $n - m$ non-integrable differential constraints on their generalised configuration velocity, Eq. 5.1 [15].

$$A(\vec{q})\,\dot{\vec{q}} = 0 \tag{5.1}$$

All feasible instantaneous generalised configuration velocities can be found by determining the kernel solution to Eq. 5.1. This can be achieved through the use of standard techniques from linear algebra, and for the cases where the number of constraints is less than the number of generalised configuration coordinates, produces a continuum of solutions, Eq. 5.2 [15].

$$\dot{\vec{q}} = G(\vec{q})\,\vec{u}, \quad \vec{u} \in \mathbb{R}^m \tag{5.2}$$

Where the column space of the $n \times m$ matrix $G(\vec{q})$ is chosen as to span the null space of matrix $A(\vec{q})$.

Differential drive platforms have $(n - m) = (3 - 2) = 1$ differential constraint on their generalised configuration velocity that arises as a consequence of the drive wheels not being able to slip in the lateral direction[2], Figure 5.5.
If the active drive wheels do not slip in the lateral direction then the translational velocity remains distributed between its underlying components in the x and y coordinates of its generalised configuration space, Eq. 5.3.

$$\frac{\dot{x}}{\cos\theta} = v = \frac{\dot{y}}{\sin\theta} \tag{5.3}$$

The resulting differential constraint is shown in Pfaffian form in Eqns. 5.4 and 5.5.

$$\dot{x}\sin\theta - \dot{y}\cos\theta \quad = \quad 0 \tag{5.4}$$

$$\overbrace{[\sin\theta \ -\cos\theta \ 0]}^{A(\vec{q})} \overbrace{\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}}^{\dot{\vec{q}}} \quad = \quad 0 \tag{5.5}$$

[1]Pfaffian forms of constraint equations are parametric representations that specify all allowable instantaneous configuration differentials as opposed to implicit representations which specify all prohibited instantaneous configuration differentials [40]

[2]Under normal operating conditions

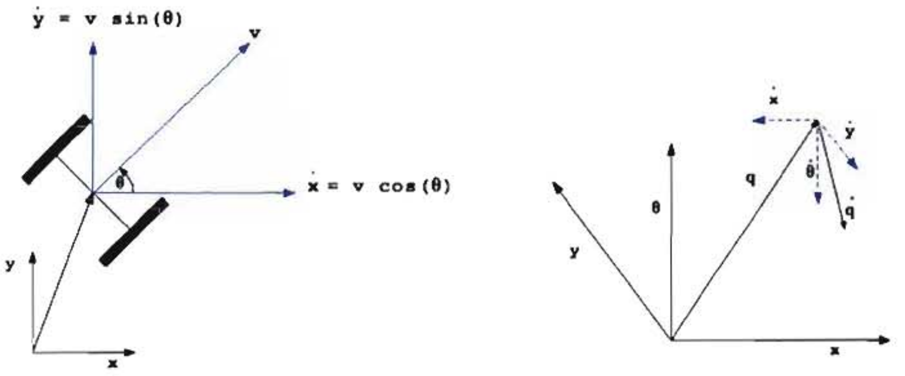


Figure 5.5: Rolling Without Slipping Condition - The configuration space is also shown in order to provide visualisation on configuration trajectories

The Pfaffian constraint equation, Eq. 5.1 is satisfied for $\dot{x} = \cos\theta$ and $\dot{y} = \sin\theta$. Scalar multiples of this solution is also a solution of Eq. 5.1, this scaling represents the magnitude of the differential drives translational velocity v, i.e. $\dot{x} = v\cos\theta$, $\dot{y} = v\sin\theta$. The Pfaffian constraint does not prohibit the remaining generalised configuration velocity, i.e. $\dot{\theta}$, which is scaled over $\mathbb{R}$ in a one to one mapping through the application of ω. Collecting all feasible solutions for the instantaneous generalised configuration velocity $\dot{\vec{q}}$ into matrix form, results in the first order kinematic model of a differential drive platform, which is a vector field f over $\mathbb{R}^n$, $f : \mathbb{R}^2 \times \mathrm{SO}^1 \times \mathbb{R}^2 \to \mathbb{R}^3$, Eq. 5.6.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{5.6}$$

The vector field associated with the first order kinematic model of a differential drive platform is best interpreted as a combination of two vector fields, Eq. 5.7.

$$\overbrace{\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}}^{f} = \overbrace{\begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix} v}^{f_1} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega}^{f_2} \tag{5.7}$$

Therefore, the translational and rotational velocity control inputs, v and ω act as weights which determine how much each vector field contributes to the resulting generalised configuration velocity. Eq. 5.7 classifies a differential drive platform, in the context of control application, as a control-affine system [40]. Control-affine systems are linear in terms of control input application but nonlinear in terms of state trajectories. Differential drive platforms are in fact a special kind of control-affine system called a drift-less control-affine system, in which the generalised configuration velocities collapse under zero control input[1].

[1]Generalised configuration velocities collapse under this condition in terms of kinematics, although on real robots like RollerMHP, the configuration velocities converge to zero under the natural dynamics of the robot and associated differential drive platform

5.2.2.4 Controllability

By taking the tangent linearisation of Eq. 5.7 about some point in the configuration space $\vec{q}_p$ a linearised system is formed, Eq. 5.8 [15].

$$\dot{\tilde{q}} = \begin{bmatrix} \cos\theta_p \\ \sin\theta_p \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega, \quad \tilde{q} = \vec{q} - \vec{q}_p \tag{5.8}$$

As far as posture stabilisation is concerned, this system is not controllable [15]. This implies that a linear controller will never be able to achieve posture stabilisation of a differential drive platform, not even in the local sense where the system is linearised continuously around current states through the use of Jacobians. In order to provide a differential drive platform with a posture stabilising motion controller, one must delve into the tools provided by nonlinear control theory. This insight does not break down the overall controllability however. Controllability of a differential drive can be shown constructively by explicitly providing a sequence of maneuvers to bring a differential drive platform from an initial configuration $[x_i, y_i, \theta_i]^T$ to any desired goal configuration $[x_g, y_g, \theta_g]^T$. Since a differential drive, configured in the sense shown in section 3.2.2.3, can rotate on itself, the posture stabilising task consists of a pure rotation on the point $[x_i, y_i]^T$, to align the orientation of the differential drive platform with the goal point, followed by a pure translation to the goal point, $[x_g, y_g]^T$, and lastly a pure rotation operation on the goal point until the platform is at the correct orientation $\theta = \theta_g$. Systems of this kind are known as locally null controllable, and as far as posture stabilisation is concerned, require special attention, section 5.2.2.5.

5.2.2.5 Feedback Stabilisability and Brockett's Condition

The vector field of a differential drive's generalised kinematic model, representing all feasible instantaneous configuration velocities, Eq. 5.6, can be generalised into the form shown in Eq. 5.9.

$$\dot{\vec{x}} = f(\vec{x}, \vec{u}), \qquad f : \mathbb{R}^2 \times SO^1 \times \mathbb{R}^2 \to \mathbb{R}^3 \tag{5.9}$$

Feedback stability of systems of the form in Eq. 5.9, through the development and implementation of a *smooth time-invariant feedback control laws*, $\vec{u} = g(\vec{x})$, requires the resulting, fully qualified vector field, $\dot{\vec{x}} = f(\vec{x}, g(\vec{x}))$, to have certain properties. Brockett [39] exposed a condition that all fully qualified vector fields must uphold, in order to make a certain goal configuration in their configuration space a "smoothly" stable equilibrium point, under full state feedback control. The condition is termed Brockett's Condition and is developed below [21].

Brockett's Condition. Given the system,

$$\dot{\vec{x}} = f(\vec{x}, \vec{u}), \qquad \vec{x}(t_0) = \vec{x}_0, \qquad f(0,0) = 0$$

with $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$, continuously differentiable. If the system is smoothly stabilisable, i.e. there exists a continuously differentiable function $g : \mathbb{R}^n \to \mathbb{R}^m$, such that the origin is an asymptotically stable equilibrium point of $\dot{\vec{x}} = f(\vec{x}, g(\vec{x}))$, with stability defined in the Lyapunov sense[1], then the image of f must contain an open neighbourhood of the origin.

For those not familiar with set theory, this states that in order to stabilise such systems, with smooth time-invariant feedback control laws, the resulting fully qualified vector field must produce integral curves, as solutions to initial value problems, that are able to approach the origin of the state-space from any direction. This can also be interpreted as a continuity condition over the vector fields around the origin. In practical applications, this condition is not limited to the origin of the state-space and, through linear transformations, is applicable to arbitrary goal points in the state space of the system. This is shown when introducing the

[1]recall section 5.2.1.2

posture stabilising controllers developed and tested on RollerMHP.

During his explanation, Brockett used the so called "nonholonomic integrator", as an example of a system that is not smoothly stabilisable onto the origin of its state space by full state feedback control [39]. The nonholonomic integrator, known in the control community as Brockett's system, is also known as the Heisenberg system as it arises in quantum mechanics [40]. It is represented as follows, Eq. 5.10.

$$\dot{x}_1 = u_1, \qquad \dot{x}_2 = u_2, \qquad \dot{x}_3 = x_1 u_2 - x_2 u_1 \tag{5.10}$$

Where $\vec{x} \triangleq [x_1, x_2, x_3]^T \in \mathbb{R}^3$ and $\vec{u} \triangleq [u_1, u_2]^T \in \mathbb{R}^2$. Since the image of the map $[\vec{x}^T, \vec{u}^T]^T \mapsto [u_1, u_2, x_1 u_2 - x_2 u_1]^T$, i.e. the vector field, does not contain the point $[0, 0, \epsilon]^T$ for any $\epsilon \neq 0$, Brockett's condition states that there is no time-invariant continuously differentiable full state feedback control law, i.e. $g(\vec{x})$ that makes the origin of the state space an asymptotically stable equilibrium point. This is best interpreted by looking at the vector fields, Figure 5.6

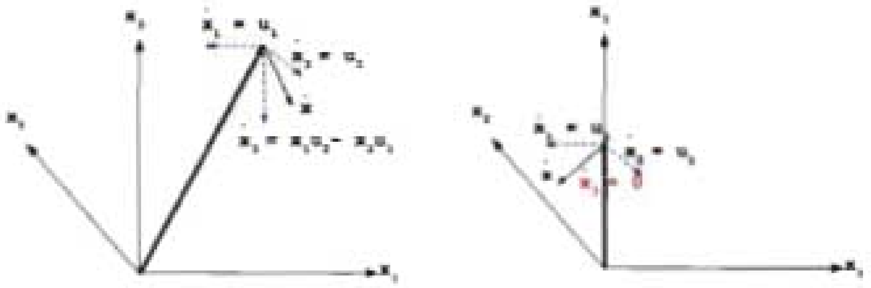


Figure 5.6: Vector Field Properties of the Nonholonomic Integrator - The state is seen to "lock" on the x_3 axis and prevent further stabilisation under the condition that $x_1 = x_2 = 0$

Whenever x_1 and x_2 are both zero, $\dot{x}_3 = 0$ and x_3 remains constant, thus destroying any chance of convergence of a configuration onto the origin. The difficulties implied by Brockett's condition can be overcome with gracious efforts into the development of time-varying periodic controllers, sliding mode control laws, stochastic control laws, as well as a number of nonlinear control law implementations [48] [31].

It is a known fact, any completely nonholonomic system with three configuration coordinates, or states, and two control inputs, can be converted into Brockett's system, i.e. a nonholonomic integrator, by a local co-ordinate transformation [34]. This transformation is known as a diffeomorphism and is used often in differential manifold theory. Since differential drive platforms meet this criteria, they are equivalent to Brockett's system through a coordinate diffeomorphism and therefore fail to meet Brockett's condition for smooth time-invariant feedback stabilisability.

From the aforementioned insights gained into the feedback stabilisability of differential drive platforms, two posture stabilising controllers were developed for RollerMHP using two different feedback control laws. The first controller implemented pivots off the vector field characteristics exposed through Brocketts nonholonomic integrator and is presented in section 5.2.3. The second controller implemented is based on a polar coordinate transformation that produces a resulting system that satisfies Brockett's condition for smooth stabilisability, however, the transformation produces a singularity at the origin of the transformed configuration space and requires special attention, section 5.2.4.

5.2.3 Logic Based Switching Control Law Implementation

In order to provide RollerMHP with a posture stabilising motion controller, a logic based switching controller was implemented. The control law is a slight modification to that in [21].

The logic based switching controller implemented on RollerMHP is based on stabilising Brockett's original nonholonomic integrator and is an example of a hybrid control law, employing both continuous dynamics and discrete logic. This makes for some interesting feedback vector fields.

5.2.3.1 Preliminary Insights

Recall the nonholonomic integrator, Eq 5.11.

$$\dot{x}_1 = u_1, \qquad \dot{x}_2 = u_2, \qquad \dot{x}_3 = x_1 u_2 - x_2 u_1 \tag{5.11}$$

It has already been established that when x_1 and x_2 are both zero, $\dot{x}_3 = 0$ and x_3 remains constant. Perhaps more importantly, whenever x_1 and x_2 are "small", only "large" control inputs will produce credible changes in x_3. One control strategy to make the origin of the configuration space a stable equilibrium point of the full state feedback control system, is to use control inputs $\vec{u} \triangleq [u_1, u_2]^T$ to keep configurations away from the x_3 axis when x_3 is large and let x_1 and x_2 become small as x_3 diminishes towards zero. This is precisely the method performed by the logic based switching controller implemented on RollerMHP. Before an introduction to the control law however, the coordinate diffeomorphism used to convert the configuration space of a differential drive into that of Brockett's nonholonomic integrator is introduced in section 5.2.3.2.

5.2.3.2 Coordinate Diffeomorphism

The coordinate diffeomorphism used for the purposes of transforming RollerMHP's configuration velocity into Brockett's nonholonomic integrator is shown in Eqns. 5.12 through 5.16 [22].

$$\begin{aligned}
x_1 &= x\cos\theta + y\sin\theta && (5.12)\\
x_2 &= \theta && (5.13)\\
x_3 &= 2\left(x\sin\theta - y\cos\theta\right) - \theta\left(x\cos\theta + y\sin\theta\right) && (5.14)\\
u_1 &= v - \omega\left(x\sin\theta - y\cos\theta\right) && (5.15)\\
u_2 &= \omega && (5.16)
\end{aligned}$$

With these coordinate transformations, RollerMHP's configuration exists in the configuration space[1] of Brockett's nonholonomic integrator and is susceptible to the configuration "locking" notions exposed in previous sections. For example, to expose the first Brockett vector field coordinate:

$$\begin{aligned}
\dot{x}_1 &= \dot{x}\cos\theta + \dot{y}\sin\theta - x\sin\theta\,\dot{\theta} + y\cos\theta\,\dot{\theta}\\
&= \dot{x}\cos\theta + \dot{y}\sin\theta + \dot{\theta}\left(y\cos\theta - x\sin\theta\right)\\
&= v\cos^2\theta + v\sin^2\theta - \dot{\theta}\left(x\sin\theta - y\cos\theta\right)\\
&= v - \omega\left(x\sin\theta - y\cos\theta\right)\\
&= u_1
\end{aligned}$$

By performing the same operations on all configuration coordinates, one will see that this diffeomorphism does indeed produce Brockett's nonholonomic integrator.

An overview of the constituent elements and operating structure of the logic based switching controller implemented to stabilise the nonholonomic integrator form of RollerMHP's configuration space is presented in section 5.2.3.3.

[1]State space is perhaps a better term in control contexts

5.2.3.3 Control Law Construction

From this point on, configuration will be synonymous with state in order to utilise readily understood control terminology.

The logic based switching controller operates by constructively switching between predetermined control laws in order to stabilise Brockett's non-holonomic integrator, and thus RollerMHP's configuration, from an arbitrary initial condition. The switching logic is rather complex and is dependent on how the Brockett state evolves through time within functionally bound and overlapping regions in $\mathbb{R}^3$, i.e. the Brockett state space. The logic based switching controller builds from two base mathematical elements [21].

1. Four, monotone nondecreasing[1], functions $\pi_j : [0, +\infty) \rightarrow \mathbb{R}, \quad j \in \mathcal{S} \triangleq \{1, 2, 3, 4\}$, with the following properties:

   (i) $\pi_j(0) = 0$ for each $j \in \mathcal{S}$, and $0 < \pi_1(w) < \pi_2(w) < \pi_3(w) < \pi_4(w)$ for every $w > 0$.

   (ii) π_1 and π_2 are bounded.

   (iii) π_1 is such that if $w \rightarrow 0$ exponentially fast, then $w/\pi_1(w) \rightarrow 0$ exponentially fast.

   (iv) π_4 is smooth on some non-empty interval $(0, c]$, and

   $$\frac{\mathrm{d}}{\mathrm{d}w}\pi_4(w) < \frac{\pi_4(w)}{w}, \qquad w \in (0, c]$$

   As well, if $w \rightarrow 0$ exponentially fast then $\pi_4(w) \rightarrow 0$ exponentially fast.

   The last condition, condition (iv), is a specification on lipschitz continuity, which can be used to determine whether a specific initial value problem, much like the nonholonomic stabilisation problem considered here, has a unique solution on a vector field.

2. Four overlapping regions in $\mathbb{R}^3$:

$$\begin{aligned}
\mathcal{R}_1 &\triangleq \{\vec{x} \in \mathbb{R}^3 : 0 \leq x_1^2 + x_2^2 < \pi_2(x_3^2)\}, \\
\mathcal{R}_2 &\triangleq \{\vec{x} \in \mathbb{R}^3 : \pi_1(x_3^2) < x_1^2 + x_2^2 < \pi_4(x_3^2)\}, \\
\mathcal{R}_3 &\triangleq \{\vec{x} \in \mathbb{R}^3 : \pi_3(x_3^2) < x_1^2 + x_2^2\}, \\
\mathcal{R}_4 &\triangleq \{0\}
\end{aligned}$$

Utilising these two base elements, the switching logic based control law takes on the form show in Eq. 5.17 [21].

$$\vec{u} = g_\sigma(\vec{x}), \qquad t \geq t_0, \tag{5.17}$$

Where σ is a piecewise, continuous from the right at every point, switching signal taking on values in the set $\mathcal{S} \triangleq \{1, 2, 3, 4\}$ for each element $\vec{x} \in \mathbb{R}^3$ under the following control structure [21].

$$g_1(\vec{x}) \triangleq \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \; g_2(\vec{x}) \triangleq \begin{bmatrix} x_1 + \frac{x_2 x_3}{x_1^2 + x_2^2} \\ x_2 - \frac{x_1 x_3}{x_1^2 + x_2^2} \end{bmatrix}, \; g_3(\vec{x}) \triangleq \begin{bmatrix} -x_1 + \frac{x_2 x_3}{x_1^2 + x_2^2} \\ -x_2 - \frac{x_1 x_3}{x_1^2 + x_2^2} \end{bmatrix}, \; g_4(\vec{x}) \triangleq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.18}$$

The switching signal σ is determined recursively, Eq. 5.19 [21].

$$\sigma = \phi(\vec{x}, \sigma^-), \qquad t \geq t_0 \tag{5.19}$$

Where, for $t \geq t_0$, $\sigma^-(t)$ denotes the limit from the left of $\sigma(\tau)$ as $\tau \rightarrow t$. $\sigma^-(t_0)$ is an element in $\mathcal{S}$ that effectively initialises Eq.5.19. This is an important aspect in the practical application

[1] Non-decreasing functions are required to prevent the state "locking" discussed in previous sections

of this control law and is covered when presenting the algorithms developed to implement the controller.

The switching signal transition function $\phi : \mathbb{R}^3 \times \mathcal{S} \rightarrow \mathcal{S}$ is a pivotal mechanism in the logic based switching controllers performance as $t \rightarrow \infty$ and is defined as shown in Eq.5.20 [21].

$$\phi(\vec{x}, j) = \begin{cases} j & \text{if } \vec{x} \in \mathcal{R}_j \\ \max\{i \in \mathcal{S} : \vec{x} \in \mathcal{R}_j\} & \text{if } \vec{x} \notin \mathcal{R}_j \end{cases} \qquad \vec{x} \in \mathbb{R}^3, j \in \mathcal{S} \tag{5.20}$$

Although it may not seem apparent, ϕ is implemented in such a way as to provide hysteresis in the switching signal when the Brockett state transitions between the various control regions $\mathcal{R}_k$ that partition the state space. This can be seen by analysing a typical set of monotone, non-decreasing functions π_j composed with the various control regions $\mathcal{R}_k$. A typical choice for a set of π_j is $\pi_1(w) = 1 - e^{-\sqrt{w}}$, $\pi_2 = 2\pi_1$, $\pi_3 = 3\pi_1$ and $\pi_4 = 4\pi_1$ [21]. For the explanation of hysteresis, the various control regions have been projected into (w_1, w_2)-space, where $w_1 = x_3^2$ and $w_2 = x_1^2 + x_2^2$, Figure 5.7.

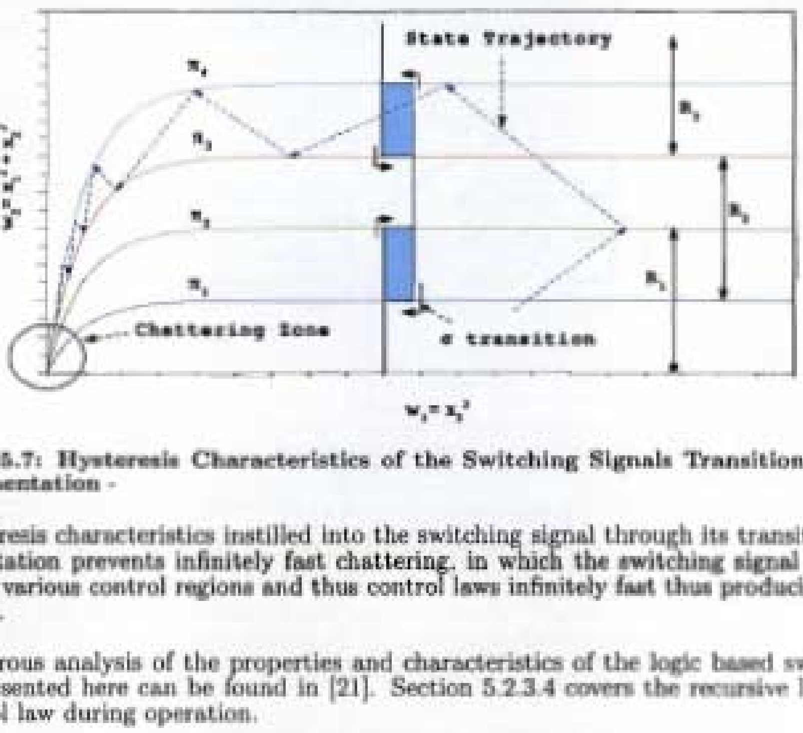


Figure 5.7: Hysteresis Characteristics of the Switching Signals Transition Function Implementation -

The hysteresis characteristics instilled into the switching signal through its transition function implementation prevents infinitely fast chattering, in which the switching signal switches between the various control regions and thus control laws infinitely fast thus producing controller instability.

A rigorous analysis of the properties and characteristics of the logic based switching controller presented here can be found in [21]. Section 5.2.3.4 covers the recursive logic used by the control law during operation.

5.2.3.4 Hybrid Dynamics and Operating Structure

The logic based switching controller, implementing full state feedback control, effectively produces a hybrid dynamic system, which has both continuous dynamics and discrete logic, Eq. 5.21.

$$\dot{\vec{x}} = f_\sigma(\vec{x}) \tag{5.21}$$

Each vector field, $\dot{\vec{x}}_i = f_i(\vec{x})$, representing the resulting fully qualified state velocity of Brockett's nonholonomic integrator under application of the i^{th} control law, Eq. 5.22, is dependent on the value of a switching signal σ with hysteresis characteristics instilled through the implementation of its transition function, Eq. 5.23,

$$\vec{u} = g_i(\vec{x}) \tag{5.22}$$

$$\sigma = \phi(\bar{x}, \sigma^-), \qquad \sigma^- = \lim_{\tau \to t^-} \sigma(\tau) \tag{5.23}$$

There are multiple tasks required in the implementation of the logic based switching controller.

1. Determine the current state of the nonholonomic integrator, i.e. determine $\bar{x} \in \mathbb{R}^3$.
2. Based on the current state, update the current control region, i.e. determine the current $\mathcal{R}_k$.
3. Based on a current state, $\bar{x}$, and associated control region $\mathcal{R}_k$ as well as σ^-, which has been effectively initialised[1] implement the hysteresis based switching signals transition function, Eq. 5.23.
4. Apply the control law $\bar{u} = g_\sigma(\bar{x})$, based on the current value of σ output from its transition function.

The controller implementation on RollerMHP runs these four operations recursively to provide posture stabilisation. The control system structure of the above logic based switching controller can be more readily understood by looking at the feedback control loop associated with its implementation, Figure 5.8.

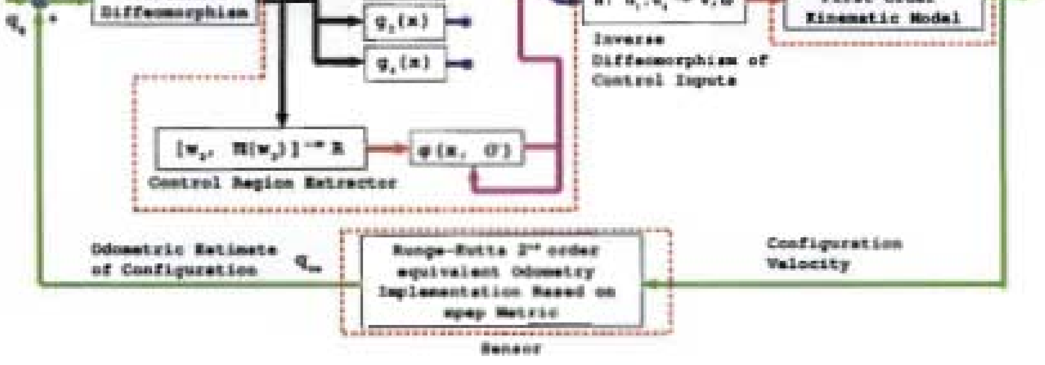

Requested Goal Configuration
Goal-Relative Configuration Error
Controller
Plant
Coordinate Diffeomorphism
x
$g_1(x)$
$g_2(x)$
$g_3(x)$
$g_4(x)$
u
$H: u_1, u_2 \to v, \omega$
Inverse Diffeomorphism of Control Inputs
RollerMHP's First Order Kinematic Model
q_c
q_e
$[w_2, H(w_2)] \to R$
$\varphi(x, \sigma)$
Control Region Extractor
Odometric Estimate of Configuration
q_m
Runge-Kutta 2nd order equivalent Odometry Implementation Based on epsp Metric
Sensor
Configuration Velocity

Figure 5.8: Control Loop Construct of the Logic Based Switching Controller - Although the "sensor", i.e. the odometry implementation, acts to integrate the configuration velocity, this is not an actual representation of the algorithm structure, Algorithm 1, which performs odometry based on acquiring accumulated encoder pulses

An overview of the algorithms implemented to perform the required control follows in sections 5.2.3.5 through 5.2.3.8.

5.2.3.5 Nonholonomic Integrator State Constructor Algorithm

The first algorithm is more of a procedure that converts the configuration error between RollerMHP's initial configuration and a requested goal configuration, with respect to the local co-ordinate system of the goal configuration, into its equivalent state error in the state space of Brockett's nonholonomic integrator and is shown in Procedure 2 with reference to Figure 5.9.

[1]This is covered during the discussion of the Algorithm 3

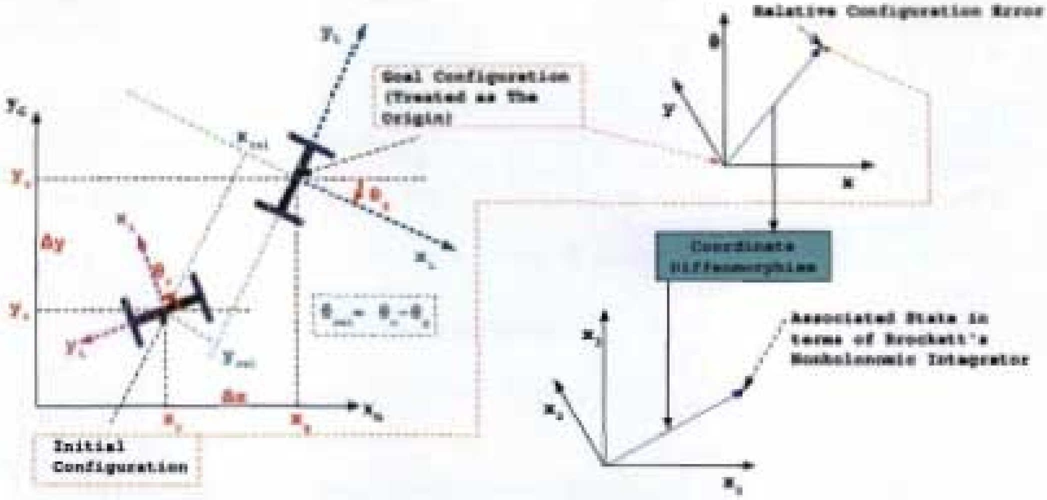

Relative Configuration Error
Goal Configuration (Treated as The Origin)
Coordinate Diffeomorphism
Associated State in terms of Brockett's Nonholonomic Integrator
Initial Configuration
Δy
Δx
$\theta_{rel} = \theta_c - \theta_g$
x_{rel}
y_{rel}
x_1
x_2
x_3

Figure 5.9: Configuration Error Calculation and Coordinate Transformation - The goal configuration must be viewed as the origin of the configuration space

```
Procedure 2: Update_Current_State
Input: Current configuration $\vec{q}_c \in \mathbb{R}^2 \times SO^1$ and requested goal configuration
       $\vec{q}_g \in \mathbb{R}^2 \times SO^1$
Output: Equivalent state of Brockett's "nonholonomic integrator" $\vec{x} \in \mathbb{R}^3$
begin
    // get $\vec{q}_c - \vec{q}_g$
    $\Delta x = x_c - x_g$
    $\Delta y = y_c - y_g$
    $\theta_{rel} = \theta_c - \theta_g$
    // rotate relative position error by $\theta_{rel}$, anti-clockwise
    $x_{rel} = \Delta x \cos\theta_{rel} + \Delta y \sin\theta_{rel}$
    $y_{rel} = -\Delta x \sin\theta_{rel} + \Delta y \cos\theta_{rel}$
    // apply diffeomorphism defined under section 5.2.3.2 to relative configuration error
    $x_1 = x_{rel} \cos\theta_{rel} + y_{rel} \sin\theta_{rel}$
    $x_2 = \theta_{rel}$
    $x_3 = 2\left(x_{rel} \sin\theta_{rel} - y_{rel} \cos\theta_{rel}\right) - \theta_{rel}\left(x_{rel} \cos\theta_{rel} + y_{rel} \sin\theta_{rel}\right)$
end
```

5.2.3.6 Control Region Extractor Algorithm (With Hysteresis)

This algorithm determines which control region $\mathcal{R}_k$ is currently occupied, based on an input state vector $\vec{x} \in \mathbb{R}^3$, such that $\phi : \mathbb{R}^3 \times \mathcal{S} \to \mathcal{S}$ can operate so as to perform the necessary switching logic, while the hysteresis characteristics prevent chattering of the switching signal, σ, as $t \to \infty$, Algorithm 3.

Algorithm 3: Control Region Extractor

```
Input: The current state vector $\vec{x} \triangleq [x_1, x_2, x_3] \in \mathbb{R}^3$
Data: Switching signal discrete set elements $s_i \in \mathcal{S} \triangleq \{1,2,3,4\}$; bounding function
      images $\pi_j$ for $j \in \{1,2,3,4\}$
Output: The current control region $\mathcal{R}_k$ for $k \in \{1,2,3,4\}$
begin
    $w_1 \leftarrow x_3^2$,    $w_2 \leftarrow x_1^2 + x_2^2$    // coordinate containers
    foreach $j \in \{1,2,3,4\}$ do
        $\pi_j \leftarrow$ tuning constant$_j$ $\times (1 - e^{\sqrt{w_1}})$    // create $\pi$ function images
    if $\sigma^-$ not initialised then
        $[w_2, \pi(w_1)]? \mapsto \mathcal{R}_k$    // determine initial control region, see section 5.2.3.3
        $\sigma^- \leftarrow s_k$    // set $\sigma^-(t_0)$ to an initial set index, equivalent to $k$
        return $\mathcal{R}_k$    // and set $\sigma^-$ initialised flag
    // implement hysteresis based region selection based on $\sigma^-$
    if $0 \leq w_2 < \pi_2$ then
        if $(w_2 > \pi_1)$ and $(\sigma^- = s_2)$ then
            return $\mathcal{R}_2$
        else
            return $\mathcal{R}_1$
    else if $\pi_2 \leq w_2 < \pi_4$ then
        if $(w_2 > \pi_3)$ and $(\sigma^- = s_3)$ then
            return $\mathcal{R}_3$
        else
            return $\mathcal{R}_2$
    else if $w_2 \geq \pi_4$ then
        return $\mathcal{R}_3$
    else
        return $\mathcal{R}_4$
end
```

The initialisation logic, $[w_2, \pi(w_1)]? \mapsto \mathcal{R}_k$, has not been explicitly shown here, as it is of secondary importance to the hysteresis logic implementation. For those who seek greater insight into this algorithm, a C++ implementation is shown in the `Update_Control_Region()` method in Appendix D.1.4.

5.2.3.7 Switching Signal Transition Algorithm

This algorithm implements a simple selection process which pivots off the work performed by Algorithm 3, in order to change the set element of the σ switching signal, Algorithm 4. This simple algorithm is applicable provided that previous work on implementing the hysteresis switching logic on the control region transitions has been taken into account.

```
Algorithm 4: Switching Signal Selection Algorithm
Input: Current control region $\mathcal{R}_k$ based on $\vec{x} \in \mathbb{R}^3$ and the switching signals left limit
       value $s_i \in \mathbb{S} \triangleq \{1, 2, 3, 4\}$
Output: The correct set element value for $\sigma$
begin
    if $i \neq k$ then
        $\sigma \leftarrow s_k$
        $\sigma^- \leftarrow \sigma$
    else
        $\sigma \leftarrow s_i$
        $\sigma^- \leftarrow \sigma$
end
```

5.2.3.8 Control Input Algorithm

This algorithm implements the control law based on the value of the current switching signal σ. The diffeomorphism is also taken into account, Algorithm 5.

```
Algorithm 5: Selectable Control Law Application
Input: The current value of the switching signal $\sigma = s_i \in \mathbb{S} \triangleq \{1, 2, 3, 4\}$
Output: A control law application $\vec{u} = g_\sigma(\vec{x})$
begin
    switch $\sigma$ do
        case $s_1$
            $\vec{u} = g_1(\vec{x})$
        case $s_2$
            $\vec{u} = g_2(\vec{x})$
        case $s_3$
            $\vec{u} = g_3(\vec{x})$
        case $s_4$
            $\vec{u} = g_4(\vec{x})$
    // convert back into required translational and rotational velocity
    $v = u_1 + u_2 (x_{rel} \sin\theta_{rel} - y_{rel} \cos\theta_{rel})$
    $\omega = u_2$
end
```

5.2.3.9 C++ Class Implementation

The logic based switching controller was implemented in C++ as a `Switcher` object. The `Switcher` object was integrated into RollerMHP's `ModRollerMHP_Driver` driver which provided the `Switcher` object with the necessary data to perform the control laws, i.e. provided access to RollerMHP's configuration data, and allowed the controller to set required translational and rotational velocities see Appendix D. The `Switcher` object is integrated directly into `ModRollerMHP_Driver` code, this was done purely for testing purposes. Currently, the code is being ported to an abstract driver, allowing research colleagues to benefit from the control law and further enhance its properties within the scope of the Player Robot Device Interface.

5.2.3.10 Testing and Response Characteristics

The logic based switching controller has very interesting response characteristics that, unfortunately, make it extremely difficult to tune. Tuning the algorithm consists of selecting the gains on the π_j functions, in order to achieve reasonable, rather than required, state trajectories in the state space of Brockett's nonholonomic integrator, recall Figure 5.7. A poor selection of bounding function gains, represented as "tuning constants" in Algorithm 3, can have a detrimental affect on posture stabilisation performance. This is a characteristic of the hybrid nature of the logic based switching controller, in that the temporally continuous switching between predetermined control laws produces feedback vector fields that reasonably stabilise the *transformed* state vector onto the origin. Thus the diffeomorphism's pre-image, i.e. the real world configuration trajectory of the differential drive platform, may not behave quite as expected. This is one of the major drawbacks of this type of posture stabilising controller implementation.

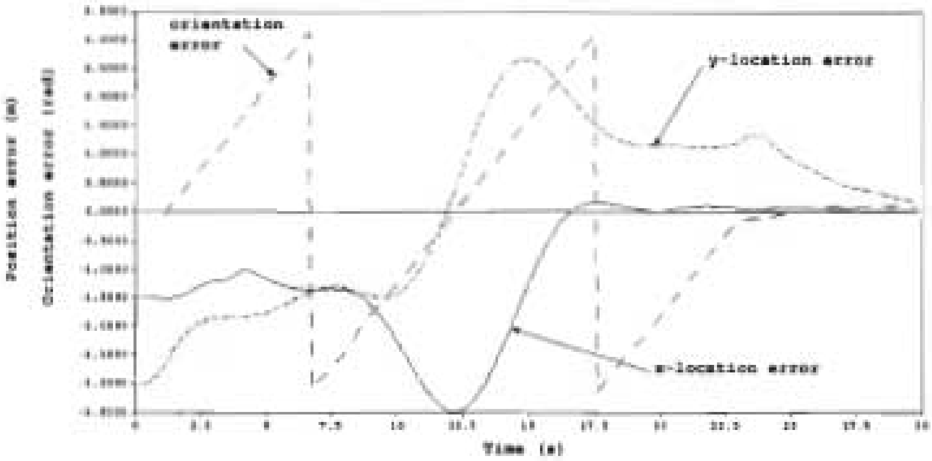


Figure 5.10: Posture Stabilisation Response for Logic Based Switching Controller - The orientation co-ordinate wraps around due to the identification of the configuration space $\mathbb{R}^2 \times SO^1$

After rigorous tuning, a response from an initial configuration of $[-1.50, -3.00, 0.00]^T$ showed convergence to the origin of the configuration space as $t \to \infty$, Figure 5.10.

As can be seen from the response of the orientation coordinate, RollerMHP did spin around a few times. This characteristic only appears from large initial configuration errors in which multiple switches in the control laws are required in order to stabilise the configuration onto the origin. The author feels that it is the time invariance of each control law implementation that causes this behaviour, and time varying control law implementations would perhaps suit this kind of application better. It is to the authors knowledge that this is the first time any algorithm of this kind has been implemented on a real mobile robot platform of considerable size and dynamic influence.

Greater insight into this controller implementation comes from analysis of the position trajectory, Figure 5.11.
The error in the final configuration is due to a number of reasons. On physical systems when the vector fields become small near the origin of the configuration space, the magnitude of the input commands fall below that required to produce further motion of the physical platform, and as such contributes to the steady state error of the controller. This behaviour is further enhanced through the quantisation of the underlying digital feedback infrastructure, i.e. the quadrature encoders providing angular velocity feedback on the drive wheels, as lower velocities are more sensitive to the quantisation effects of digital feedback.

The run-time architecture of the software implementations used in executing the controller on computing infrastructure also plays a major role in performance. Multi-threaded implementations that run the control application algorithm, Algorithm 5 in a separate thread from that of the state extraction and switching signal transition algorithms, Procedure 2 and Algorithms

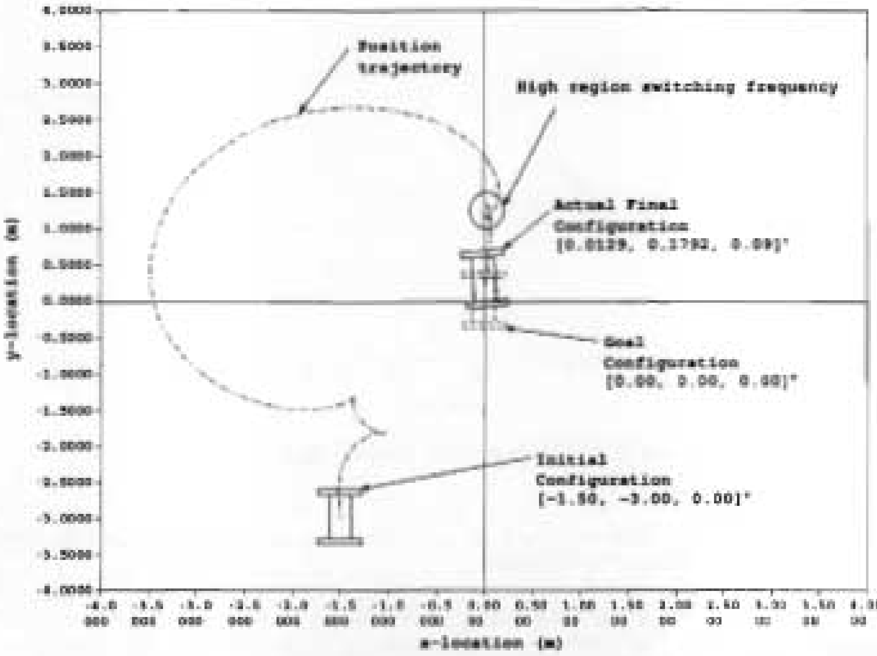


Figure 5.11: RollerMHP's Position Trajectory Response -

3 and 4, are exposed to the negative effects of random scheduling. This can produce unwanted characteristics that are not actually part of the controllers fundamental mathematical properties. These "race conditions" usually contribute to headaches, rather than performance, and should be eliminated through the correct mutex[1] locking of shared data, such as the image of the switching signal σ.

After testing the controller with multi-threaded based code implementations, the author is convinced that a single threaded code implementation of the controller can only better the performance of the switching signal and thus the controller, in stabilising Brockett's nonholonomic integrator and thus, through diffeomorphic transformations, mobile robot platforms.

To show how bad things can get with the hybrid dynamic system produced through the associated switching logic of the controller, RollerMHP's first posture stabilisation response is shown in Figure 5.12. RollerMHP was placed on a stand during this initial test, this was great insight as can be seen in the response.
The response shown in Figure 5.12 is a true indication of Lyapunov stability, and pays tribute to the truly nonlinear nature of the feedback stabilisation of differential drive platforms, and the associated Brockett system through diffeomorphic transformation. Although Lyapunov stability is considered a weak form of stability, it is a very real quantity that should not be overlooked in the discussion of system performance.

5.2.4 Control Law Implementation Through Polar Coordinate Transformation

The previous controller utilised the control characteristics of logic based switching between time-invariant control laws to produce resultant vector fields capable of producing, locally lipschitz and piecewise continuous, integral curves, or state trajectories, that converge to the origin

[1]Mutual Exclusions are used in computer science to represent the semantics associated with mutually exclusive access to shared data in a multi-threaded software application

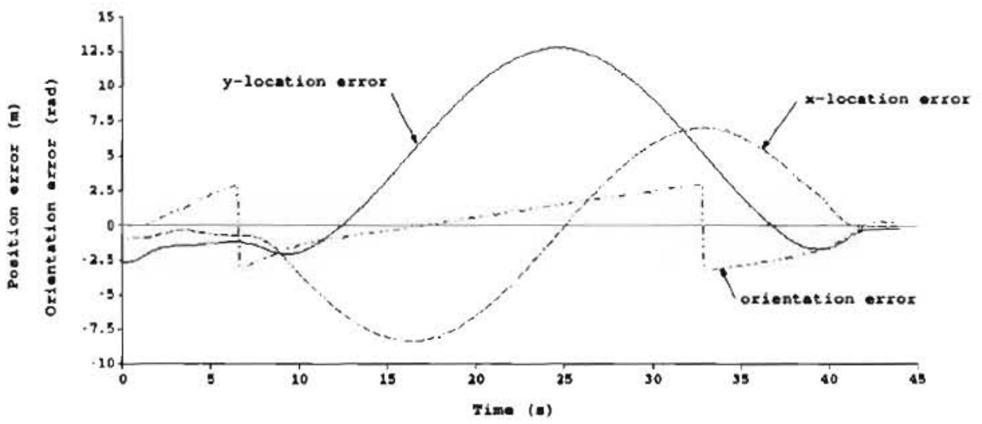


Figure 5.12: RollerMHP's First Response -

of the state space as $t \to \infty$. This was done so as to overcome the obstructions and conditions imposed by the insights provided by Brockett.

Another method of overcoming the control obstructions placed on systems not satisfying Brockett's Condition for smooth feedback stabilisability, is to remove the Cartesian coordinates from the differential drives state space and replace them with ones of a polar coordinate nature. This was done so for the second controller implementation which is based on the control law developed in [12].

5.2.4.1 Polar Coordinate Transformation

With reference to Figure 5.13, the following polar coordinate transformation was applied to the configuration of a differential drive platform to produce a control system which overcomes Brockett's Condition for smooth time-invariant feedback stabilisation, Eq. 5.24 through 5.26 [15]. For clarity purposes in the following description, relative configurations will be denoted as if they are absolute, i.e. $q_{rel} = q$, although all configurations are relative to the goal point, which is denoted the origin during control application, and as such, one can treat q_{rel} as q.

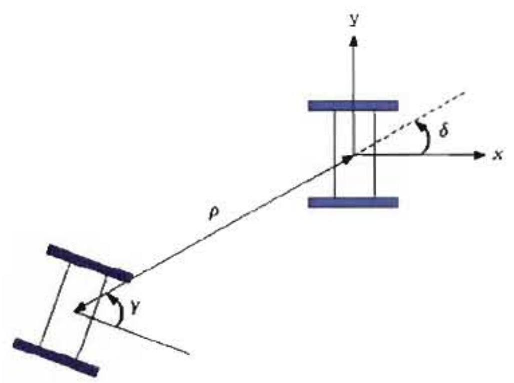


Figure 5.13: Polar Coordinate Transformation -

$$\rho = \sqrt{x^2 + y^2} \qquad (5.24)$$

$$\begin{aligned} \gamma &= \tan^{-1}\left(\frac{y}{x}\right) - \theta + \pi && (5.25) \\ \delta &= \gamma + \theta && (5.26) \end{aligned}$$

Using this new coordinate system, the system model, in the form of state differential equations, for a differential drive becomes characterised by a singularity at the origin of the state space, Eq. 5.27 through 5.29 [15].

$$\begin{aligned} \dot{\rho} &= -\cos\gamma\, v && (5.27) \\ \dot{\gamma} &= \frac{\sin\gamma}{\rho} v - \omega && (5.28) \\ \dot{\delta} &= \frac{\sin\gamma}{\rho} v && (5.29) \end{aligned}$$

This introduces some practical restrictions on algorithm implementation and results in a lower bound on accuracy. However, through careful algorithm implementation, this lower bound can be made insignificant.

5.2.4.2 Control Law Implementation

The following non-linear control law was used in setting up the necessary feedback vector fields to produce a globally asymptotic and stable equilibrium point at the origin of the state space, Eq. 5.30 and 5.31 [15].

$$\begin{aligned} v &= k_1\, \rho \cos\gamma && (5.30) \\ \omega &= k_2\, \gamma + k_1 \frac{\sin\gamma\cos\gamma}{\gamma}\left(\gamma + k_3\, \delta\right) && (5.31) \end{aligned}$$

Where, k_1, k_2, and k_3 are the tuning parameters.

5.2.4.3 Tuning Properties

The pivotal properties of the control law can be exposed by transforming back into the configuration space incorporating Cartesian coordinates and analysing the resulting input vector fields along the x and y axes of the local goal configuration, Eq. 5.32 and 5.33.

$$\begin{aligned} v &= k_1 \sqrt{x^2+y^2}\cos\left(\tan^{-1}\left(\frac{y}{x}\right) - \theta + \pi\right) && (5.32) \\ \omega &= k_2\left(\tan^{-1}\left(\frac{y}{x}\right) - \theta + \pi\right) + \\ & \quad k_1\left[\frac{\sin\left(\tan^{-1}\left(\frac{z}{x}\right) - \theta + \pi\right)\cos\left(\tan^{-1}\left(\frac{y}{x}\right) - \theta + \pi\right)}{\left(\tan^{-1}\left(\frac{y}{x}\right) - \theta + \pi\right)}\right] \\ & \quad \times\left[\tan^{-1}\left(\frac{y}{x}\right) - \theta + \pi + k_3\left(\tan^{-1}\left(\frac{y}{x}\right) + \pi\right)\right] && (5.33) \end{aligned}$$

By analysing the control law in the "native" configuration space of the differential drive platform allows insight into tuning the controller in order to optimise performance.

Along the x and y axes of the goal configuration, input translation vector fields v vary sinusoidally, according to the relative orientation error between the current configuration and goal configuration, Figure 5.14.

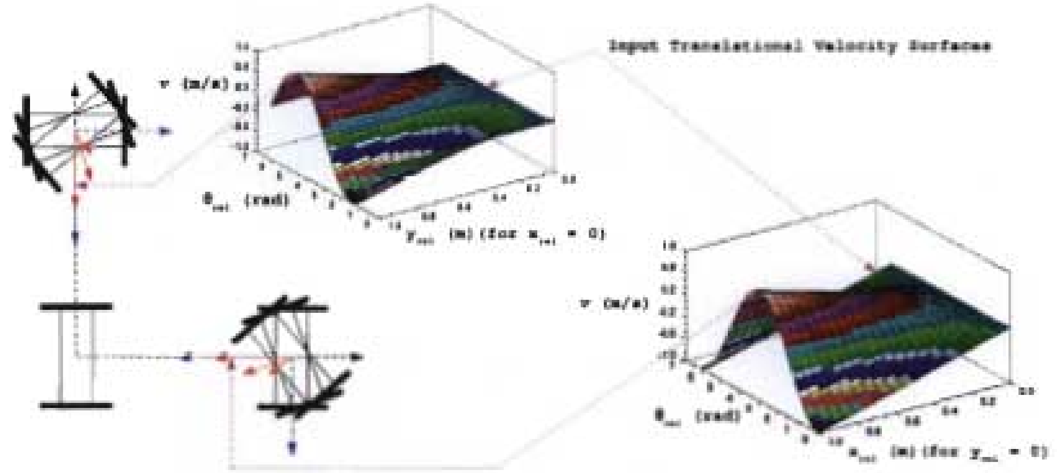


Figure 5.14: Input Translational Velocity Variation - As can e seen by the input translational velocity variations, shortest path convergence properties have been incorporated into this controller

For current configurations anywhere along the goal configurations local y-axis, i.e. $\forall x = 0$, from Eq. 5.32, $\tan^{-1}(y/x) = \pi/2 - n\,\pi, n \in \{0,1\}$, and $v = k_1\,|y|\,\cos(\pi/2 + (1-n)\,\pi - \theta_{rel})$.

Along the positive y-axis, $n = 0$ and $v = k_1\,y\,\cos(\frac{3\pi}{2} - \theta_{rel})$, which is maximised at $k_1\,y$ for $\theta_{rel} = \frac{3\pi}{2}$ and minimised at $-k_1\,y$ for $\theta_{rel} = \frac{\pi}{2}$. Along the negative y-axis, $n = 1$ and $v = k_1\,|y|\,\cos(\pi/2 - \theta_{rel})$, which is maximised at $k_1\,|y|$ for $\theta_{rel} = \frac{\pi}{2}$ and minimised at $-k_1\,|y|$ for $\theta_{rel} = \frac{3\pi}{2}$.

For current configurations anywhere along the goal configurations local x-axis, i.e. $\forall y = 0$, from Eq. 5.32, $\tan^{-1}(y/x) = n\,\pi, n \in \{0,1\}$, and $v = k_1\,|x|\,\cos((n+1)\,\pi - \theta_{rel})$.

Along the positive x-axis, $n = 0$ and $v = k_1\,x\,\cos(\pi - \theta_{rel})$, which is maximised at $k_1\,x$ for $\theta_{rel} = \pi$ and minimised at $-k_1\,x$ for $\theta_{rel} = 0$. Along the negative x-axis, $n = 1$ and $v = k_1\,|x|\,\cos(2\pi - \theta_{rel})$, which is maximised at $k_1\,|x|$ for $\theta_{rel} = 0$ and minimised at $-k_1\,|x|$ for $\theta_{rel} = \pi$

For other configurations involving non-zero x and y values, the $\tan^{-1}(y/x)$ term of the input translational velocity, Eq. 5.32 contributes to the phase shift in the sinusoidal velocity variations around relative orientation errors, according to the current approach quadrant. However, the magnitude of the translational velocity input v is still tuned entirely by the constant k_1 and thus, must be set to a value that optimises performance over the operating range of the posture stabilisation controller. By the definition of a FMRP, recall section 2.3.2.3, the euclidean distance between a configuration on the boarder of a Region of Convergence and the I/O port of the associated manufacturing infrastructure subset represents the maximum "$\rho = \sqrt{x^2 + y^2}$" seen by the posture stabilising controller. In order to prevent unnecessary saturation of control inputs in the face of physical systems like RollerMHP, the maximum translational velocity input must potentially occur[1] on the boarder of the RoC, i.e.

$$|v|_{max} = k_1\,\rho_{max}$$

[1] i.e. If the relative orientation in the configuration error between a configuration on the boarder of a RoC and the I/O port is such as to produce an extreme value of Eq. 5.32

The DC motor and chain and sprocket power transmission unit powering each active drive wheel provides RollerMHP with a maximum translational velocity of approximately 2 m/s. RollerMHP has been governed to a maximum translational velocity of 1.5 m/s in order to prevent excessive wear on the drive motors. Utilising a maximum radius for a RoC of 3 m, equivalent to the maximum ρ seen by the control law in Equation 5.30, a gain of $k_1 = 0.5$ provides the necessary mapping of maximum translational velocity to suitable configurations on the boarder of a RoC as required.

Through much the same analysis of Eq. 5.33, the tuning parameters, k_2 and k_3 can be chosen so as to provide maximum angular velocity inputs within the capability of the physical system in question, i.e. RollerMHP.

Although this analysis may provide slight insight into tuning the controller, it is always best to optimise the performance through on-line tuning. On-line tuning produced $k_2 = 0.4$ and $k_3 = 0.6$ to provide convergence from arbitrary configurations within a 3m circular region around the goal configuration, i.e. $[0.0, 0.0, 0.0]^T$ in times less than 10 seconds.

5.2.4.4 Testing and Response Characteristics

For testing purposes, the polar coordinate based posture stabilising control law was integrated directly into the `ModRollerMHP_Driver` driver, see the `PolarControlAlgorithm()` method in Appendix C.1.

The polar coordinate transformation produces trajectories that seem "natural" in the sense that the trajectories mimic those of feed-forward control. This is beneficial as one can determine the maximum area occupied by the mobile platform during posture stabilisation.

Multiple tests were preformed during the on-line tuning of the controller. Tasks were assigned a difficulty rating, depending on the degree in which the differential constraints were "stimulated" by the requested motion. For example, a parallel parking operation is the most difficult posture stabilisation operation for a differential drive, while a straight line motion is the easiest.

Preliminary tests involved the on-line tuning of the controller and were all performed from the same initial configuration of $[-3.00, -1.50, 0.0]$. The first response was selected at $k_1 = 0.4$, $k_2 = 0.3$ and $k_3 = 0.7$, Figure 5.15.

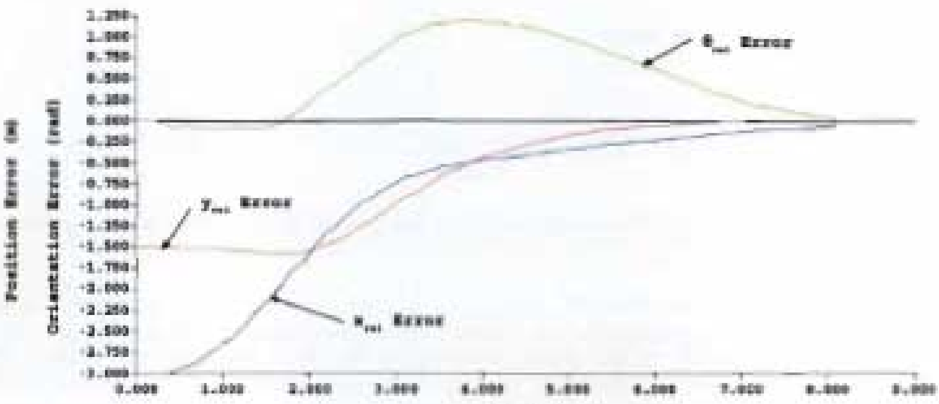


Figure 5.15: Posture Stabilisation -

The "natural" response characteristics of the controller can be seen in the $x_{rel} - y_{rel}$ trajectory, Figure 5.16. The response produced a steady state error in position of around 1.6% of the initial configuration error with an absolute value of $\rho = 5.32\text{cm}$. This can be considered reasonable in the face of the digitised low-level feedback infrastructure and frictional effects of RollerMHP's drive infrastructure.

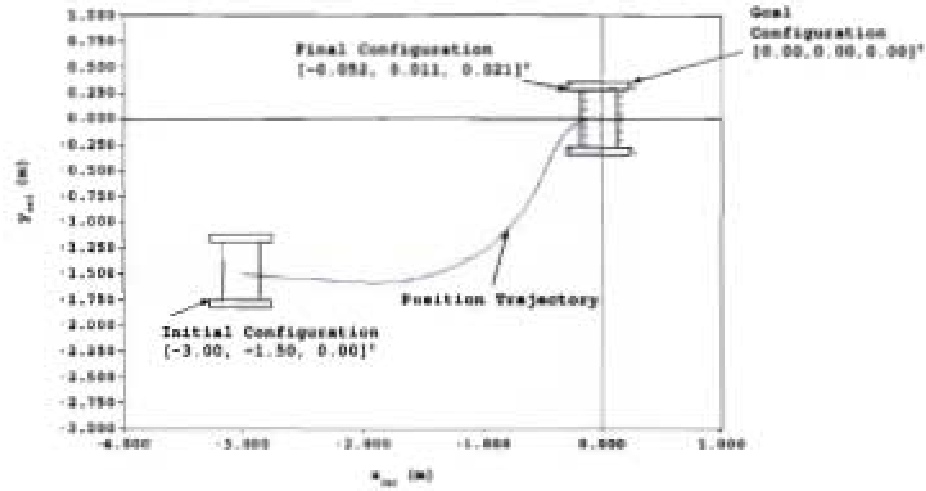


Figure 5.16: Position Trajectory -

The control inputs fall well below saturation levels and show good convergence properties, Figure 5.17.

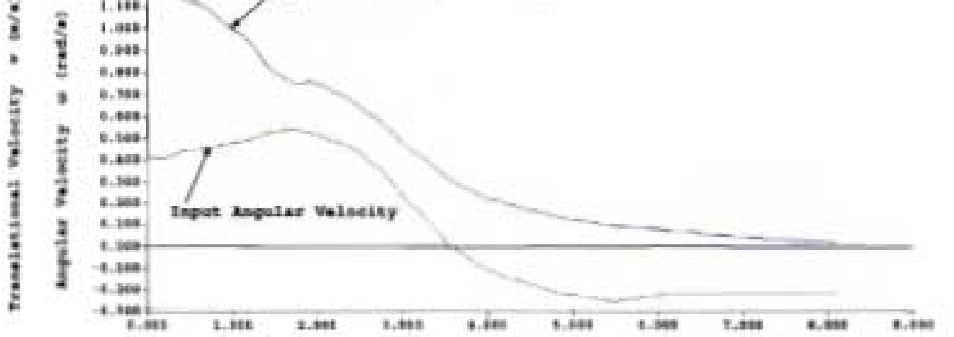


Figure 5.17: Control Input Convergence - The fixed lower limit on the angular velocity input ω as $t \to \infty$ and $\rho \to 0$ is a practical requirement during application of this controller

The angles γ and δ are undefined for $x = y = 0$ and as such, the practical application of this controller requires slight modification to the control law implementation. As $\rho \to 0$, the values for γ and δ must be "locked" on the values assumed in the final approaching phase. This practical obstruction partially contributes to the steady state error of the controller and can be seen in the angular velocity response in Figure 5.17.

From insight into the extremums of the translational and angular velocity inputs over the configuration error-space representing an empty RoC, and the physical limitations of RollerMHP, the controller was tuned to as to attain minimum convergence times while avoiding control input saturation. With this setup, $k_1 = 0.5$, $k_2 = 0.4$ and $k_3 = 0.6$, Figure 5.18.

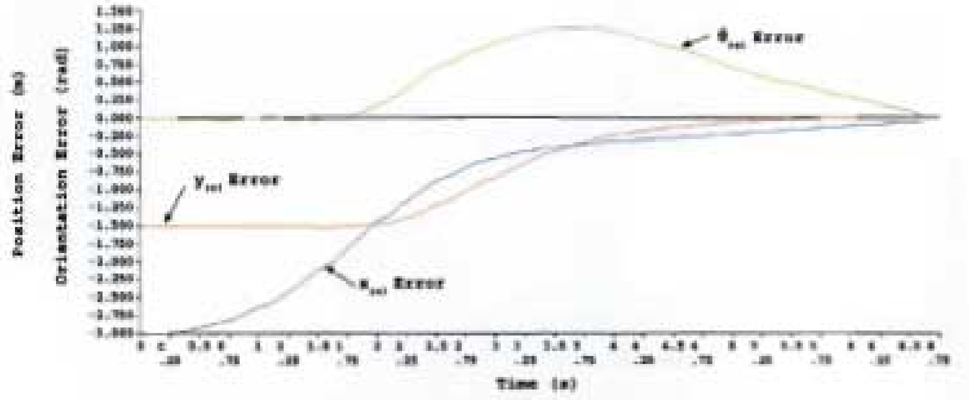


Figure 5.18: Optimal Convergence -

With these tuning parameter values, the steady state error is approximately 2% of the initial configuration error with an absolute value of $\rho = 7$cm, Figure 5.19.

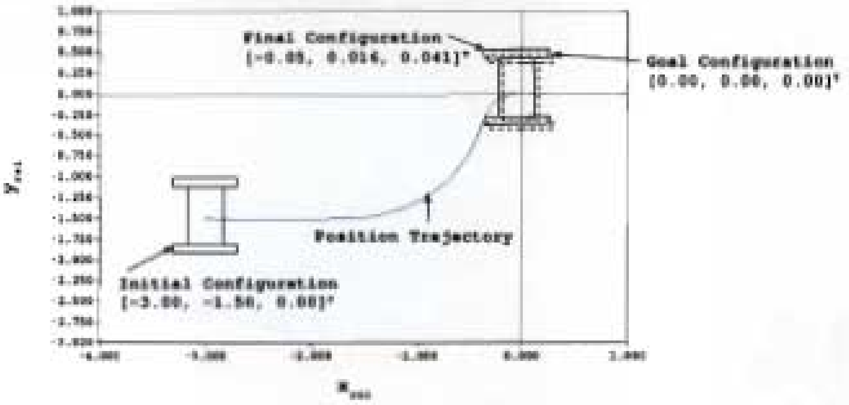


Figure 5.19: Time-Optimal Position Trajectory -

The translational velocity input command is seen to be maximised at 1.5 m/s at the initial configuration error thus contributing to the minimal convergence time, Figure 5.20.
The hardest task, in terms of posture stabilisation through full state feedback control, for a differential drive platform, is a parallel parking operation. This directly opposes the differential constraints imposed on the platforms generalised configuration velocity through the rolling without slipping condition. RollerMHP was given the task of parallel parking under posture stabilisation in order to rigorously test the capabilities of the polar coordinate transformation based controller. A response from an initial configuration of $[0.00, -3.00, 0.00]^T$ is shown in Figure 5.21. The position trajectory is shown in Figure 5.22.
The control inputs are within their limits as shown in Figure 5.23 RollerMHP can be seen performing a parallel parking operation in the video located on the CD accompanying this dissertation.

5.2.5 Summary

In this particular instance of physical platform implementation, facilitation of the motion requirements of the materials handling primitive of a FMRP requires posture stabilisation, through full state feedback control, of under-actuated mobility systems with nonholonomic

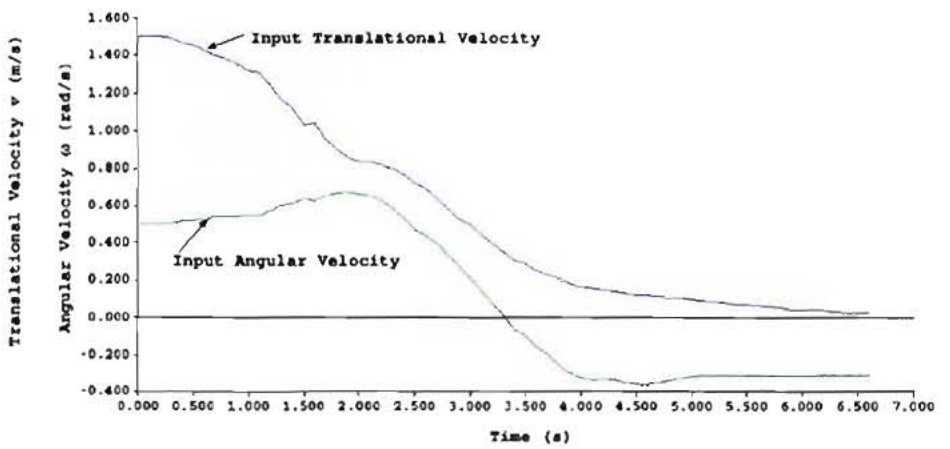


Figure 5.20: Time-Optimal Control Input Convergence -

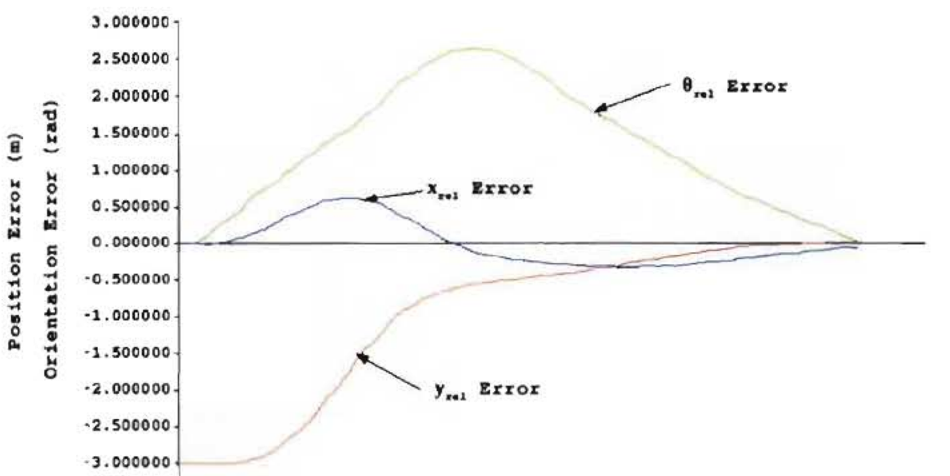


Figure 5.21: Configuration Error Convergence for Parallel Parking Operation -

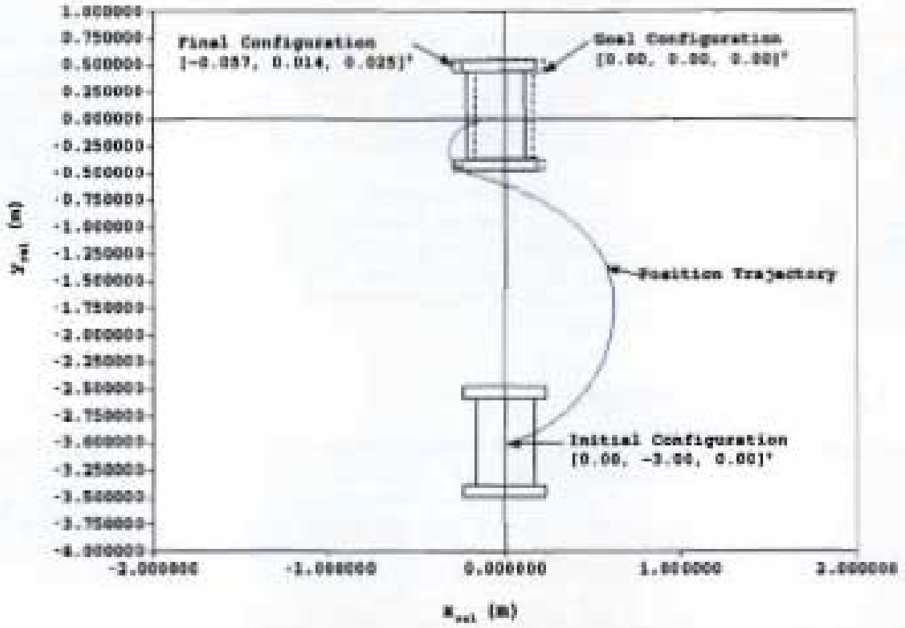


Figure 5.22: Position Trajectory for a Parallel Parking Operation -

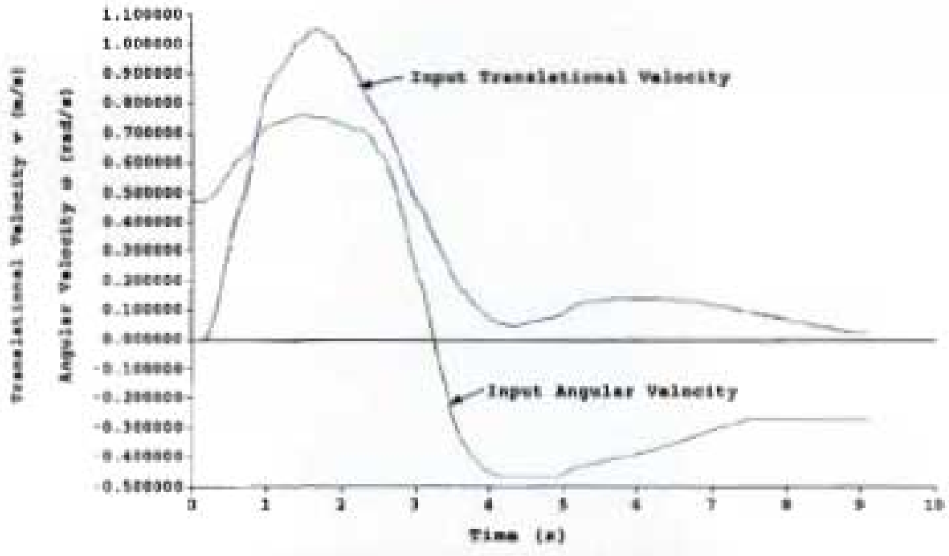


Figure 5.23: Control Input Convergence for Parallel Parking Operation -

kinematic constraints. These constraints limit the applicability of suitable controllers to non-linear, piecewise continuous and time varying control laws.

The response achieved with the polar coordinate transformation based controller , enables the occupancy region[1] to be estimated before hand. This is due to the "natural" response characteristics created through the implementation of the control law. Also, the properties of the resulting fully qualified vector field make optimisation of the controller, in terms of matching physical capabilities with the those produced by the tuned controller, a relatively simple task.

Due to the vector field properties of Brockett's system, i.e. the state locking characteristics of the nonholonomic integrator, posture stabilisation based on the application of control laws, designed to stabilise generic nonholonomic integrators, does not produce feasible state trajectories in terms of establishing deterministic occupancy regions. This is due to the limited applicability of diffeomorphisms required to transform the generalised configuration of differential drive platforms into that of Brockett's system.

5.3 Chapter Summary

The Task Execution Layer of the AMTS has been presented in terms of control and navigation requirements. The real-time aspects of material payload and transportation task execution is associated with this layer of the AMTS, and so methods of establishing "longest execution time" metrics should be of concern. Establishing these sort of performance metrics for the Global Navigation sub-block is a difficult, or perhaps even intractable task, due to the uncertainty in the placement of dynamic obstacles, such as other mobile materials handling and routing robot platforms[2]. The development of these metrics for the Materials Handling sub-block provides grounds for far more tractable efforts. The RoC in the definition of a FMRP is one such element that provides the grounds to develop "longest execution time" metrics. As seen in section 5.2.4, one can optimise certain classes of posture stabilisation controllers such that physical capabilities of materials handling and routing robot platforms can map to "longest execution time" quantities through considering the size of the RoC. This was achieved in section 5.2.4.3 and allows RollerMHP to have a, worst case, 10 second convergence time[3] independent of initial configurations on the boundary of a RoC.

[1]For this discussion, the occupancy region is considered as the total region of space required to perform posture stabilisation, in terms of the joint collective of positional trajectories and the physical volume occupied by the materials handling and routing robot platform

[2]Static obstacles are known with absolute certainty, i.e. the plant layout

[3]Analogous to settling time in control terms

Chapter 6

Summary and Future Research

''Problems worthy of attack prove their worth by fighting back'' - Paul Erdos

This chapter aims at summarising this dissertation in such a way as to bring into scope the crucial insights into the problems involved in facilitating the materials handling environments created through MCM production operations.

The research objectives of section 1.2 are once again addressed including the solution methods employed in achieving them.

6.1 Mass Customisation Manufacturing

In the context of facilitating modern niche markets and establishing first mover market share through responsive manufacturing techniques, the successful implementation of MCM production structures relies on a manufacturing firms ability to constructively and concurrently integrate all available manufacturing resources. This includes both passive and active manufacturing infrastructure.

The control theoretic model and description of the production structure involved in MCM, recall Figure 2.1 provides semantic homogeneity for engineers across multiple disciplines. The author considers this approach as a means of establishing Common Model Development (CMD) in the modern research community. The standardisation of concepts brought on by CMD can provide the necessary infrastructure to establish international research partnerships, thus aiding the development of third world nations through constructive and relevant research and development practices.

6.1.1 Effective Plant Layout Design

It has been established in chapter 2 that plant layout, as a passive measure of decreasing required materials handling, is just as important as the development of highly engineered autonomous materials handling and routing robots, in facilitating the materials handling requirements of customer-induced variations in production requirements.

The deterministic nature of customer-induced variations in production requirements plays a major role in establishing insight into the concurrent design and application of plant layout structures and active flexible materials handling infrastructure, such as the mobile robot platform developed during this research project. This aspect of production implementation once again highlights the need for concurrent engineering. Papers such as [49] describing methods of applying probabilistic models to develop metrics that describe customer preference in product choice is evidence that there exists a research community involved in the research required to achieve insight into the development of efficient MCM production operations.

6.1.2 MCM Product Design through DFMC

It is not uncommon for engineers to create and introduce unnecessary production problems into a manufacturing environment through aggressive design and development of products that are sensitive to process and product configuration variations. As far as the problem space associated with this research project is concerned, the design of products that minimise required materials handling is a particularly useful design goal. In this regard, DFMC plays one of the most crucial roles in establishing successful MCM production operations and implicitly determines the production dynamics and materials handling and routing environment associated with production rate output of a manufacturing system, recall the pre-filter analogy of Figure 2.1.

Production engineers should try to strike a balance between developing products that "behave well", in terms of producing bounded variations in production requirements, and products that have the capability of satisfying the psyche of demanding customers. This is by no means an easy task, although through the application of flexible materials handling and routing systems, the problems associated with customer-induced material routing variations could be to a lesser degree.

6.1.3 The FMRP Definition and Problem Space

As this research was concerned with materials handling and routing in MCM production environments, the problem space associated with performing a materials handling and routing task between distributed, unconnected, manufacturing infrastructure required quantification in order to establish a well defined problem space.

The FMRP task definition provided in section 2.3.2.3 takes into account the various forms of motion control required to provide robust material payload transfer to and from materials handling infrastructure via correct alignment, through posture stabilisation, of the mobile robot platform with the input/output port infrastructure of FMS processing cells.

A material transportation primitive of a FMRP task essentially decomposes into a global path planning and local navigation and real-time obstacle avoidance problem. There is much literature on this subject in the mobile robotics research community although, in the majority of the literature, application scope falls under unstructured environments unlike those utilising structured plant layouts for production implementation. There is a need to develop path planning algorithms for mobile materials handling and routing robot platforms that include optimisations based on knowledge of scheduling outputs in the production plant.

6.2 The AMTS for Generic Encapsulation

Quantifying the properties and requirements for developing solutions to the problem space spanned by the materials handling environment of MCM based customer-induced variations in production requirements, has allowed for the generic encapsulation of requisite functionality in an implementation architecture. This architecture, the AMTS, provides flexible materials handling and routing system implementations with the capability of interfacing with higher-level management frameworks at a level that can allow for the application of scheduling optimisations procedures.

By implementing an architecture rather than a specific materials handling system implementation, one can encapsulate the generic motion control and communication facilities required in order to perform high-level[1] material payload routing in the dynamic production environments of MCM production plant.

[1] High-level in the sense of scheduling management and autonomous operation

6.2.1 Device Collections versus Monolithic Robot Implementations

One of the main aspects of the AMTS is the near one to one mapping between the functionality specified by the sub-blocks of each respective layer and the associated functionality required for FMRP task execution. This axiomatic property allows for a shift in the way in which physical implementations of the ATMS are envisaged in a system scope. Rather than monolithic robot implementations with limited reconfigurability, the ATMS produces platforms which essentially represent the constructive integration of functionally correlated, yet operationally disjoint devices, each performing well defined operations. This can be seen in RollerMHP's physical implementation.

6.2.2 The Need for Software Scalability and Code Re-Use

In order to produce an extensible and scalable framework for developing control structures for mobile materials handling and routing robot platforms, methods acquired from Operating System theory can be put to good use. The functionality provided by the HAL software construct of Operating System implementations is highly applicable to modern robotic systems with heterogeneous hardware implementations.

The Player Robot Device Interface is the worlds leading implementation of this functionality and is one of the most successful Open Source software projects in the mobile robotics research community. The client-server operating model makes Player even more powerful by proving network scoped access to device instances, thus allowing for the distribution of device messages across IP networks. This also means that any software system which can communicate through TCP sockets is capable of manipulating robotic devices under Player's operating context, as long as each system adheres to XDR data marshaling and Player's messaging constructs.

RollerMHP's network scoped device access provides the capability of high information distribution and resource integration, for example, Player's server implementation can accept an arbitrarily large amount of client application subscriptions meaning that multiple clients can hold proxy's on the same devices. This has good application in large distributed manufacturing environments where the task allocation systems could implement dynamic client subscriptions in the underlying FMRP task allocation procedures as the mobile robot platforms navigate in and out of various processing regions of the plant. This would be required in high noise environments where there are already multiple active wireless networks, in order to maximise the communication signal strength providing FMRP task allocations.

6.2.3 The Motion Control Problem

Chapter 5 presented the control obstructions and solutions in implementing full state feedback control in order to perform posture stabilisation of differential drive platforms. Due to Brockett's Condition, linear control was not accessible and non-linear control laws or discontinuous control laws had to be implemented. This lessens stability criteria development to those established by Lyapunov which relies on the creation of auxiliary functions to determine the stability of the closed loop motion control system. As the AMTS places specification on the capabilities of the mobility generation devices rather than mechanical configuration, an effort to produce holonomic mobility devices would allow for the application of linear time-invariant controllers for posture stabilisation. However, the posture stabilisation performance provided by the polar coordinate transformation based posture stabiliser of section 5.2.4 shows that differential drive platforms are highly applicable in the materials handling environment of MCM production operations.

What holonomic mobility configurations provide in feedback stabilisability and controllability, an equally high gain in mechanical complexity accompanies the functionality. This brings up the question of platform maintenance and operational robustness. The simpler mechanical configuration would provide a more robust mobility generating device and so the author sees differential drives as a good choice of mobility generation for physical implementations facilitating the output specifications of the Mobility Hardware sub-block.

6.3 Research Project Summary

This research project has produced a materials handling and routing system implementation architecture with the axiomatic properties to shift the way in which materials handling and routing systems are envisaged in a system management and utilisation scope. By studying the problem space spanned by the systems implicitly linked to materials handling and routing requirements, such as plant layout and customer-induced variations in production requirements brought on by the operating structure of MCM production implementations, a well defined flexible materials handling task was quantified in a FMRP definition.

Based on the FMRP definition and requisite execution functionality encapsulation, the AMTS, a physical instance of a subset of the AMTS was realised to produce the RollerMHP mobile robot platform. By developing RollerMHP in alignment with the AMTS, a network scoped generic control structure was developed using the Player Robot Device Interface, a world leader in Open Source robotic software.

RollerMHP provided a sound test bed for establishing the applicability of differential drives in providing the necessary mobility in order to execute materials handling and routing operations in MCM production plants. The control obstructions established through nonholomic kinematics and differential constraints were overcome through the use of discontinuous control laws and nonlinear control laws, each showing asymptotic stability in the sense of Lyapunov.

Overall, the research project has provided great insight into the problem space spanned by the production dynamics associated with customer-induced variations in production requirements in Mass Customisation Manufacturing. Over the research project, the following papers were written and published in international conference proceedings.

- Bright. G and Walker. A.J. Standardised framework for flexible materials handling management, based on operating system primitives. *In proc. of the Australasian Conference on Robotics and Automation, ACRA*, 2007. Brisbane Australia

- Bright. G and Walker. A.J. Mobile mechatronic platform architecture for flexible materials handling. *In proc. of the Australasian Conference on Robotics and Automation, ACRA*, 2007. Brisbane Australia

- Bright. G and Walker. A.J. A mobile mechatronic platform architecture for the development of flexible materials handling systems. *in proc. of the 17th International Federation of Automatic Control, IFAC, World Congress* 2008. Seoul Korea

6.4 Future Research

Future research in the field of advanced manufacturing systems should involve the development of vertically integrated materials handling and routing systems. This vertical integration into higher-level manufacturing management frameworks would allow for higher-level manufacturing execution and control systems to perform real-time materials handling and routing scheduling optimisation procedures through structured access to "task sink interfaces" associated with the physical devices performing payload routing operations in a production plant. In this regard, the Task Allocation Layer of the AMTS architecture requires development. Furthermore, design of the systems used in integrating the sub-blocks of the Task Allocation Layer should develop on knowledge of the properties and characteristics of MCM production environments covered in chapter 2.

Middleware systems are required to provide the necessary syntactic and semantic translation between the software systems associated with higher-level manufacturing management frameworks and those associated with the encapsulation of the Task Allocation Layer of the AMTS. The request protocols of interest are associated with instilling mutually exclusive access specifications between the input/output port infrastructure of processing cells and materials handling robot platforms. The definition of a FMRP, recall section 2.3.2.3, incorporated the notion of a request access protocol for this reason.

Future research work in developing metrics to quantify the dynamic behaviour of production plant involved in MCM operations, would provide insight into the development of suitable production operations based on the concurrent concepts presented in chapter 2, recall Figure 2.1.

Appendix A

Embedded System Technology

A somewhat sparse description of the embedded control technology developed by Acroname Incorporated has been included here in order to provide insight into the technology used in implementing the embedded systems during this research project.

A.1 BrainStem® Technology

BrainStem® technology integrates hardware and software to produce embedded control modules that operate in a form analogous to the human nervous system. The technology was developed by Acroname Incorporated [37], and is used by research organisations, academic institutions, and in commercial Original Equipment Manufacturers (OEM) applications alike.

A.1.1 Hardware Module Implementation

BrainStem® technology has been encapsulated in a family of PIC18C252 based embedded control modules, Figure A.1.

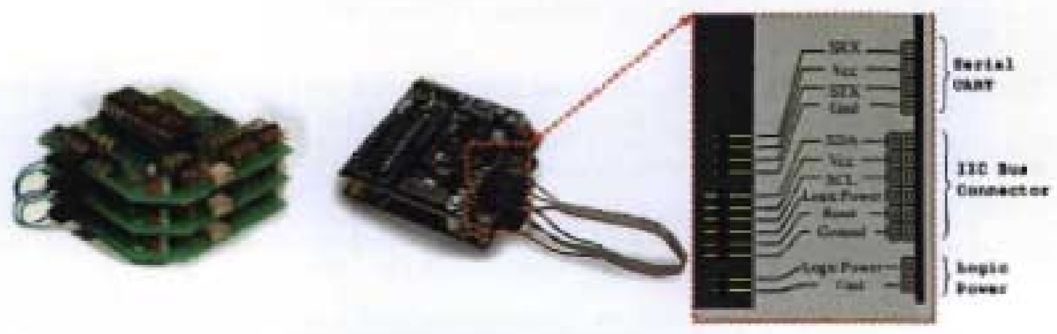


Figure A.1: BrainStem® Network and Form Factor - The IIC bus connector on each module has been designed to allow a set of modules to be stacked on top of each other, much like the PC 104 form factor

All control modules have a standardised serial UART, Inter-Integrated Circuit (IIC) Bus, and logic power interface which allows the technology to develop on networking concepts by relaying information across industry standard protocols such as IIC. This allows for scalability in hardware implementation through the utilisation of a common form factor.

In a BrainStem® network, one module is configured to act as a router and handles traffic between the host computing platform and all other modules on the network, utilising its built-in routing engine.

Each module has a different application scope ranging from general purpose use for interfacing with sensors, actuators, LCD screens, and other embedded hardware through to dedicated high resolution motion control. An introduction to the embedded control modules utilised during this research project is presented in sections A.1.1.1 and A.1.1.2.

A.1.1.1 BrainStem® Moto 1.0 Module

The BrainStem® Moto 1.0 module has been specifically designed for closed loop motion control applications and provides two high resolution motion control channels. Each motion control channel has the I/O capability of providing a Pulse Width Modulation (PWM) signal between 2.5kHz and 5MHz and a TTL[1] logic "direction" signal to drive a multitude of H-bridge DC motor drivers. Quadrature encoder inputs are also available on each motion control channel for velocity and position feedback in order to implement closed loop motion control of DC motor drive systems, Figure A.2.

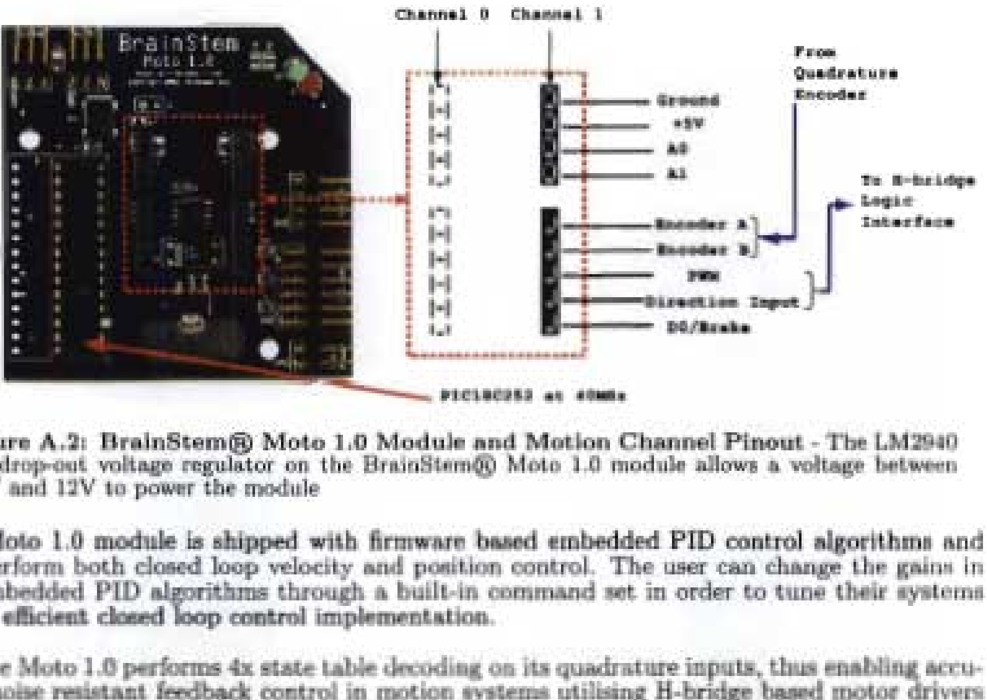


Figure A.2: BrainStem® Moto 1.0 Module and Motion Channel Pinout - The LM2940 low drop-out voltage regulator on the BrainStem® Moto 1.0 module allows a voltage between 4.5V and 12V to power the module

The Moto 1.0 module is shipped with firmware based embedded PID control algorithms and can perform both closed loop velocity and position control. The user can change the gains in the embedded PID algorithms through a built-in command set in order to tune their systems for an efficient closed loop control implementation.

The Moto 1.0 performs 4x state table decoding on its quadrature inputs, thus enabling accurate, noise resistant feedback control in motion systems utilising H-bridge based motor drivers as DC motor interface circuitry and quadrature encoders as feedback devices.

The Moto 1.0 keeps track of encoder counts for each motion control channel in a 32 bit accumulator. All feedback information into the embedded PID control loops from the quadrature encoder inputs is made available to the user through memory mapped I/O ports, allowing the user to aquire data to perform odometric calculations[2] by reading memory mapped I/O. Host computing platforms manipulate the Moto 1.0 module through a serial UART communication link. The access and operating modes for BrainStem® modules is discussed under section A.1.2.

A.1.1.2 BrainStem® GP 1.0 Module

The BrainStem® GP 1.0 module is a general purpose embedded controller and provides the same I/O facilities as most embedded micro-controller systems.

[1] Transistor-Transistor Logic
[2] Odometry is covered in section 3.2.2.3

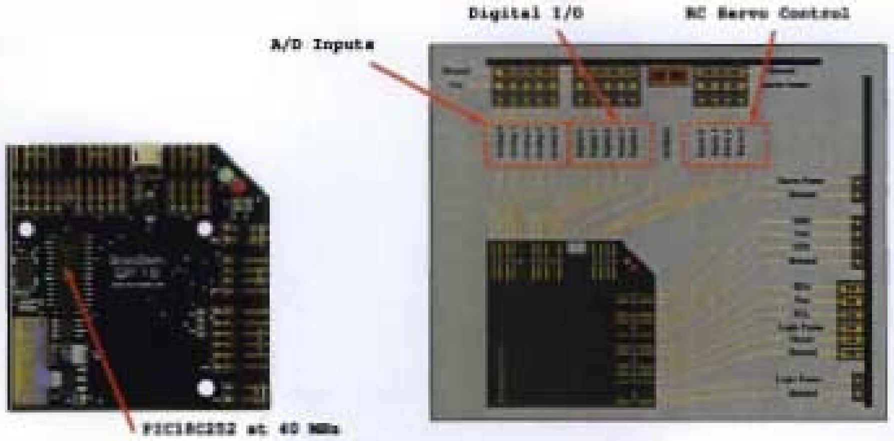


Figure A.3: BrainStem® GP 1.0 Module and I/O Pinout - The GP 1.0 module has recently been replaced by the GP 2.0 module, which has added functionality, [37]

Apart from providing the standard serial UART and IIC interface, the module provides A/D facilities, digital I/O and an array of PWM channels for driving standard RC servo's, Figure A.3.

BrainStem modules operate in the context of a system architecture that ranges from high-level software Application Programming Interfaces (API's) and support libraries, through to low-level device I/O for communicating with peripheral embedded hardware. An overview of the BrainStem® architecture follows in section A.1.2

A.1.2 System Architecture

The BrainStem® architecture encapsulates high-level software, industry standard communication and inter-connect standards, and hardware modules to provide a comprehensive environment for robotic system development, Figure A.4.

A.1.2.1 High-Level Application Programming Interface

The highest level of abstraction holds cross-platform software library support and Application Programming Interfaces (API's) for Java, C and C++ developers. Supported computing and Operating System (OS) platforms include Windows, WinCE, PalmOS, MacOS X, and Linux.

A.1.2.2 Embedded Run-Time Kernel and Programming Framework

A BrainStem® specific embedded programming language has been developed by Acroname called Tiny Embedded Application (TEA) that encapsulates a subset of the C programming language. Each BrainStem® module has been equipped with an embedded multi-tasking run-time kernel that allows several embedded TEA programs to execute concurrently. The embedded kernel executes op-codes stored in a peripheral EEPROM memory chip to provide concurrency for embedded application execution. Due to the EEPROM based implementation of the run-time kernel, BrainStem® modules are limited to a computational performance of around 9000 operations per second, which is considerably slow when considering that the PIC18C252 providing the CPU core and I/O peripherals runs at 40MHz. Flexibility in program execution is, however, greatly increased through the embedded run-time kernel. TEA source files are compiled by the "aSteep" compiler on the host machine, into executable objects. These executables can then be loaded onto a BrainStem® module through the host-to-module serial communication link in an In System Programming (ISP) fashion. A virtual machine abstraction has been developed and encapsulated in the "aTEAvm" software application that allows embedded

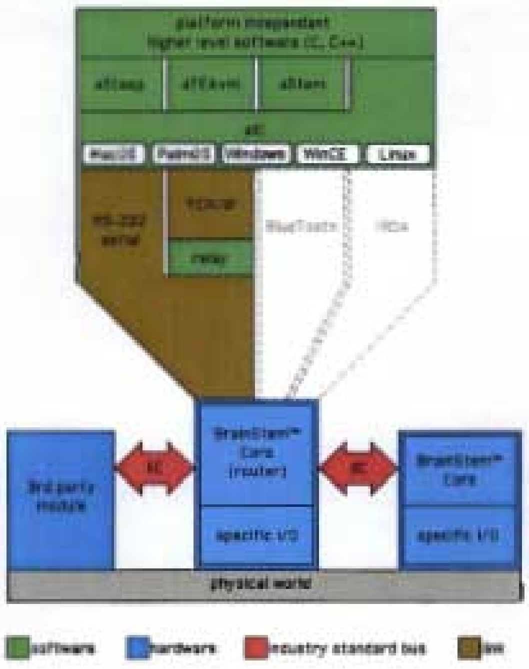


Figure A.4: The Architecture of the BrainStem® Technology - The "a" prefix on each software block is represents "Acroname" and is used throughout the software API's for maintain portability between computing platforms

TEA executables to be debugged on host computing machines, before being loaded onto the modules.

A.1.2.3 Packetised Communication Protocol

Host computing platforms communicate with BrainStem® modules through an industry standard serial UART, under a BrainStem® specific packet protocol. Data sent to BrainStem® modules from host platforms first passes through the "aStem" packet processing engine[1], which packages standard serial character streams into packet structures that are recognisable by the function table handling routines that form part of the embedded firmware on the BrainStem® modules. Data sent back from the BrainStem® modules is once again filtered through the "aStem" packet processing engine where the packets are dismantled back into standard streams of serial characters for host interpretation.

A.1.3 Operating Modes

BrainStem® modules can run in multiple operating modes. These modes are not mutually exclusive and it is possible for a module to be running in more than one operating mode at any particular instant.

A.1.3.1 Slave Mode

In this mode, a host computer manipulates or reads memory mapped I/O directly through the serial UART. BrainStem® modules act as translators between higher-level host platforms and a number of analog, digital, IIC and other devices.

A.1.3.2 TEA Mode

BrainStem® modules operate in this mode by executing, possibly multiple concurrent, embedded TEA programs through the facilities offerd by the embedded run-time kernel.

A.1.3.3 Reflex Mode

In this mode, one command or device I/O triggers another command or series of commands. Reflexes are at the lowest level of program execution and are generally used in the context of Inturrupt Service Routines (ISR) to free up the module from polling critical I/O transitions.

A.1.4 Summary

BrainStem® technology has a multi-layered operating architecture that can facilitate cross-platform development of embedded systems. This is benficial in modern times as embedded applications are realised across heterogeneous systems. BrainStem® technology has been implemented in a family of PIC18C252 based embedded control modules. Module application scope ranges from general purpose digital I/O to dedicated closed loop motion control.

[1]In the form of a shared library encapsulation of packet processing functions implemented in C

Appendix B

Embedded TEA Code Listing

Included here are source code listings for the embedded applications that were run on the BrainStem® modules.

B.1 TEA Source File for BrainStem® GP 1.0 Module

This code maintains sensor readings in the scratchpad, such that RollerMHP's onboard computer can read in arrays consisting of range data.

```
\\ The Devantech SRF08 and SRF02 ultrasonic sensors have the same
 IIC interface, infact the same embedded PIC,
and so use the same source code and header prototypes

#include <aSRF08.tea>
#include <aPad.tea>
#include <aCore.tea>
#include <aPrint.tea>

#define NUMOFSONARS 12

void main(void)
{
  int sonar_ret = 0;
  int debug;

  aCore_Sleep(50000);

  int i;
  int j;

  while(1)
    {
      char sonar_address = (char)0xE0;

      for(i = 0; i < NUMOFSONARS; i++)
{
  sonar_ret = aSRF08_RangeInt((char)sonar_address, aSRF08_CM);
  aPad_WriteInt(j, sonar_ret)
  j = j + 2;
  sonar_address = (char)(sonar_address + 2);
}
    }
}
```

B.2 BrainStem® Moto 1.0 TEA Source File

The code for controlling the left drive wheel has been listed here, the code for controlling the right drive wheel is the same, apart from channel selection.

```
// Embedded TEA code for the Brainstem Driving the Left Motor
```

```
#include <aMotion.tea>
#include <aPad.tea>

void set_channel_parameters(void)
{

aMotion_SetMode(0, aMOTION_MODE_ENCVEL, (1 << aMOTION_PWMFLAG_INVPID));
aMotion_SetMode(1, aMOTION_MODE_OFF, 0);

aMotion_SetParameter(0, aMOTION_PARAM_P, 320);

aMotion_SetParameter(0, aMOTION_PARAM_I, 0);

aMotion_SetParameter(0, aMOTION_PARAM_D, 1600);

aMotion_SetParameter(0, aMOTION_PARAM_COFFSET, 0);

aMotion_SetParameter(0, aMOTION_PARAM_PWMRAIL, 32767);

aMotion_SetParameter(0, aMOTION_PARAM_PERIOD, 200);

aMotion_SetParameter(0, aMOTION_PARAM_PWMFREQ, (int)(0b00000000|0b11111111));

}

void configure_motor_control(void)
{
// Set the ramp acceleration step
// time to 200ms for smooth
// velocity transitions
aMotion_SetRampAccStepTime(0, 200);

// Configure channels for velocity damping
aMotion_SetRampFlags(0, 0x0001);

aMotion_SetEnc32(0, 0, 0);

aMotion_RampEnable(0, 1);

return;
}

int main(void)
{

// local storage for the scratchpad "reference velocity"
int left_velocity;

set_channel_parameters();
configure_motor_control();

// Initialise the required scratchpad values
// to zero to prevent any nasty motor runaways
// at power on
aPad_WriteInt(0, (int)0);

// Start the infinite loop to continuously update
// velocity setpoints
while(1)
{
left_velocity = aPad_ReadInt((char)0);
aMotion_SetRampVel(0, left_velocity);
}
return 0;
}
```

Appendix C

C++ Player Driver Code

Below is a listing of the driver code developed to support RollerMHP under the position2d and sonar interface specifications of the Player Robot Device Interface.

C.1 position2d Driver Code

Included in the code listed here, are methods for creating file structures at run-time, for data logging purposes. Also included is the implementation of the polar coordinate transformation based posture stabilisation control law of section 5.2.4. This can be seen under the ModRollerMHP_Driver::PolarControlAlgorithm() method which runs in its own thread of execution after being invoked through the message processing methods associated with ModRollerMHP_Driver::ProcessMessage(), recall that a child implementation of a Driver object must sit in an infinite loop and process Messages which arrive on their associated MessageQueue object.

C.1.1 Header File modrollermhp.h

```
#ifndef _ModRollerMHP_Driver
#define _ModRollerMHP_Driver

#include <aIO.h>
#include <aStem.h>
#include <aPad.h>
#include <pthread.h>

#include <libplayercore/playercore.h>
#include <cstdio>


int max(double, double);

class ModRollerMHP_Driver : public Driver
{
  virtual void Main();

  player_devaddr_t position_addr;
  player_devaddr_t power_addr;

  int last_lticks;
  int last_rticks;
  bool odometryinitialised;
  double gamma_negative;
  double delta_negative;
  bool gammadeltainit;

  aStreamRef linkstream;
  aIOLib ioref;
  aStemLib stemref;

  pthread_mutex_t controller_exit_mutex;
  int GetOdometry(int* lt, int* rt, short* lv, short* rv);
  int UpdateOdometry(int lt, int rt);
  int GetLibraries();
  int OpenTerminal();
  int ClearAccumulator(const char channel);
 public:
  double posx;
  double posy;
  double posa;
  pthread_t polarcontrolthread;
  FILE* xerror;
  FILE* yerror;
  FILE* therror;
  FILE* xposition;
  FILE* yposition;
  FILE* velinput;
  FILE* omegainput;
  const char* serial_port;
  bool polar_controller_called;
  bool position_target_reached;
  player_position2d_cmd_pos_t localpositioncmd;
  ModRollerMHP_Driver(ConfigFile* cf, int section);
  // controller thread
  static void* polarentryfunc(void* arg);
  int ProcessMessage(MessageQueue* resp_queue,
     player_msghdr* hdr,
     void* data);
  int ProcessVelCommand(player_position2d_cmd_vel_t* cmd);
  int thread_controller();
  int PolarControlAlgorithm();

  virtual int Setup();
  virtual int Shutdown();
  int SetVelocity(int, int);
};
#endif // ModRollerMHP_Driver
```

C.1.2 Source File modrollermhp.cc

```
/*
 *  Player - One Hell of a Robot Server
 *  Copyright (C) 2000
 *     Brian Gerkey, Kasper Stoy, Richard Vaughan, & Andrew Howard
 *
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
*/
```

```
/* Source code for supporting wheeled mobile robots which utilise,
 * as there underlying drive control infrastructure, BrainStem Moto 1.0 modules
 *
 *
 * Author: Anthony John Walker, awalker@ukzn.ac.za
*/
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <aModuleUtil.h>
#include <aMotion.h>
#include <aModuleVal.h>
#include <aUtil.h>
#include <aModule.tea>
#include <aCmd.tea>
#include <aModuleVM.h>
#include <aErr.h>

#include "modrollermhp.h"
#include "modrollermhp_constants.h"

Driver* ModRollerMHP_Driver_Init(ConfigFile* cf, int section)
{
  return ((Driver*)(new ModRollerMHP_Driver(cf, section)));
}

void ModRollerMHP_Driver_Register(DriverTable* table)
{
  table->AddDriver("modrollermhp", ModRollerMHP_Driver_Init);
}

extern "C"
{
  int player_driver_init(DriverTable* table)
  {
    ModRollerMHP_Driver_Register(table);
    return(0);
  }
}

ModRollerMHP_Driver::ModRollerMHP_Driver(ConfigFile* cf, int section)
: Driver(cf, section)
{
  memset(&this->position_addr, 0, sizeof(player_devaddr_t));
  memset(&this->power_addr, 0, sizeof(player_devaddr_t));

  if(cf->ReadDeviceAddr(&this->position_addr, section, "provides",
   PLAYER_POSITION2D_CODE, -1, NULL) == 0)
    {
      if(AddInterface(this->position_addr) != 0)
{
  this->SetError(-1);
  return;
}
    }

  if(cf->ReadDeviceAddr(&this->power_addr, section, "provides",
PLAYER_POWER_CODE, -1, NULL) == 0)
    {
      if(AddInterface(this->power_addr) != 0)
{
  this->SetError(-1);
  return;
}
    }

  this->serial_port = cf->ReadString(section, "port", MODROLLERMHP_DEFAULTPORT);
```

```
}
int ModRollerMHP_Driver::Setup()
{
  puts("ModRollerMHP_Driver initialising...");
  this->posx = this->posy = this->posa = 0.0;
  this->odometryinitialised = false;
  this->polar_controller_called = false;
  this->gammadeltainit = false;
  pthread_mutex_init(&this->controller_exit_mutex, NULL);
  this->xerror =
    fopen("/home/awalker/development/motiondriver/logdata/x.txt", "w+");
  this->yerror =
    fopen("/home/awalker/development/motiondriver/logdata/y.txt", "w+");
  this->therror =
    fopen("/home/awalker/development/motiondriver/logdata/th.txt","w+");
  this->velinput =
    fopen("/home/awalker/development/motiondriver/logdata/velin.txt", "w+");
  this->omegainput =
    fopen("/home/awalker/development/motiondriver/logdata/omegain.txt", "w+");
  this->xposition =
    fopen("/home/awalker/development/motiondriver/logdata/xpos.txt", "w+");
  this->yposition =
    fopen("/home/awalker/development/motiondriver/logdata/ypos.txt", "w+");
  if(GetLibraries() != 0)
    {
      PLAYER_ERROR("Could not retrieve library entry addresses");
    }
  if(OpenTerminal() != 0)
    {
      PLAYER_ERROR1("failed to open terminal: %s\n", strerror(errno));
    }
  this->ClearAccumulator(0);
  this->ClearAccumulator(1);
  // Start the Main() thread... calls dummy static funcs etc.
  StartThread();
  puts("ModRollerMHP_Driver initialised");
  return(0);
}
int ModRollerMHP_Driver::Shutdown()
{
  puts("Releasing the ModRollerMHP_Driver...");
  // Set the heartbeat to link dependant
  aModuleVal_Set(this->stemref, 4, aMODULE_VAL_HBFLAG, 0);
  StopThread();
  if(this->ioref)
    {
      if(aIO_ReleaseLibRef(this->ioref, NULL) != 0)
{
  PLAYER_ERROR("Failed to release the IO library");
}
    }
  if(this->stemref)
```

```
    {
      if(aStem_ReleaseLibRef(this->stemref, NULL))
{
  PLAYER_ERROR("Failed to release the Stem library");
}
    }
  fclose(this->xerror);
  fclose(this->yerror);
  fclose(this->therror);
  fclose(this->velinput);
  fclose(this->omegainput);
  fclose(this->xposition);
  fclose(this->yposition);
  if(this->position_target_reached == true)
    {
      pthread_join(this->polarcontrolthread, NULL);
    }
  pthread_mutex_destroy(&this->controller_exit_mutex);
  puts("ModRollerMHP_Driver is shutdown.");
  return(0);
}
int ModRollerMHP_Driver::ProcessMessage(MessageQueue* resp_queue,
player_msghdr* hdr,
void* data)
{
  if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_CMD,
   PLAYER_POSITION2D_CMD_VEL,
   this->position_addr))
    {
      assert(hdr->size = sizeof(player_position2d_cmd_vel_t));
      this->ProcessVelCommand((player_position2d_cmd_vel_t*)data);
      return(0);
    }
  else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_CMD,
PLAYER_POSITION2D_CMD_POS,
this->position_addr))
    {
      assert(hdr->size = sizeof(player_position2d_cmd_pos_t));
      this->localpositioncmd = *(player_position2d_cmd_pos_t*)data;
      if(this->polar_controller_called == false)
{
  puts("Threading Controller...");
  this->thread_controller();
  this->polar_controller_called = true;
}
      return(0);
    }
  else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
   PLAYER_POSITION2D_REQ_GET_GEOM,
   this->position_addr))
    {
      assert(hdr->size = sizeof(player_position2d_geom_t));
      player_position2d_geom_t geom;
      geom.pose.px = 0.0;
      geom.pose.py = 0.0;
      geom.pose.pa = 0.0;
      geom.size.sl = 0.508;
      geom.size.sw = 0.600;
```

```
        this->Publish(this->position_addr, resp_queue, PLAYER_MSGTYPE_RESP_ACK,
      PLAYER_POSITION2D_REQ_GET_GEOM,
      (void*)&geom, sizeof(player_position2d_geom_t), NULL);
        return(0);
      }
  else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
PLAYER_POSITION2D_REQ_RESET_ODOM,
this->position_addr))
      {
        if(hdr->size != sizeof(player_position2d_reset_odom_config_t))
{
  PLAYER_WARN("Arguement to req reset odom is wrong size");
  return(-1);
}
        this->ClearAccumulator(0);
        this->ClearAccumulator(1);
        this->posx = 0;
        this->posy = 0;
        this->posa = 0;
        this->Publish(this->position_addr, resp_queue,
      PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_RESET_ODOM);
        return(0);
      }
  else
  return(-1);
}
int ModRollerMHP_Driver::SetVelocity(int lv, int rv)
{
  aErr err = aErrNone;
  err = aPad_WriteInt(this->stemref, 2, 0, lv);
  err = aPad_WriteInt(this->stemref, 4, 2, rv);
  if(err != aErrNone)
      {
        PLAYER_ERROR("SetVelocity Failed");
        return(-1);
      }
  return(0);
}
int ModRollerMHP_Driver::ProcessVelCommand(player_position2d_cmd_vel_t* cmd)
{
  double rotation = 0.0;
  double command_leftvel = 0.0;
  double command_rightvel = 0.0;
  int final_leftvel = 0;
  int final_rightvel = 0;
  double translation = cmd->vel.px;
  double rotspeed = cmd->vel.pa;
  double omega = rotspeed/MODROLLERMHP_MAXROTSPEED;
  double v = translation/MODROLLERMHP_MAXTRANS;
  int sigma = max(fabs(v), fabs(omega));
  switch(sigma)
      {
      case 1:
        {
if(translation > 0)
  translation = MODROLLERMHP_MAXTRANS;
else
  translation = -MODROLLERMHP_MAXTRANS;
rotspeed = rotspeed/fabs(v);
```

```
      gamma = atan2(yrel, xrel) - thetarelative + M_PI;
      delta = gamma + thetarelative;
      this->gamma_negative = gamma;
      this->delta_negative = delta;
    }
  double k1 = 0.5;
  double k2 = 0.4;
  double k3 = 0.6;
  player_position2d_cmd_vel_t velcmd;
  velcmd.vel.px = k1*rho*cos(gamma);
  double A = (sin(gamma)*cos(gamma)/gamma);
  double B = gamma + k3*delta;
  velcmd.vel.pa = k2*gamma + k1*A*B;
  fprintf(this->velinput, "%f\n", velcmd.vel.px);
  fprintf(this->omegainput, "%f\n", velcmd.vel.pa);
  this->ProcessVelCommand(&velcmd);
  if((rho + fabs(thetarelative)) < 0.1)
    {
      this->gammadeltainit = false;
      pthread_mutex_lock(&this->controller_exit_mutex);
      this->position_target_reached = true;
      this->polar_controller_called = false;
      pthread_mutex_unlock(&this->controller_exit_mutex);
    }
  return(0);
}
int ModRollerMHP_Driver::GetLibraries()
{
  aErr err = aErrNone;
  aIO_GetLibRef(&this->ioref, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("aIO_GetLibRef() failed");
    }
  aStem_GetLibRef(&this->stemref, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("aStem_GetLibRef() failed");
    }
  return((int)err);
}
int ModRollerMHP_Driver::OpenTerminal()
{
  aErr err = aErrNone;
  aStream_CreateSerial(this->ioref, this->serial_port, 9600,
       &this->linkstream, &err);
    if(err != aErrNone)
      {
PLAYER_ERROR1("aStream_CreateSerial() failed: %s\n", strerror(errno));
PLAYER_ERROR("Releasing the library references, please try again\n");
aStem_ReleaseLibRef(this->stemref, NULL);
aIO_ReleaseLibRef(this->ioref, NULL);
return(-1);
      }
    aStem_SetStream(this->stemref, this->linkstream, kStemModuleStream, &err);
    if(err != aErrNone)
```

```
break;
      }
    case 2:
      {
if(rotspeed > 0)
  rotspeed = MODROLLERMHP_MAXROTSPEED;
else
  rotspeed = -MODROLLERMHP_MAXROTSPEED;
translation = translation/fabs(omega);
break;
      }
    default:
      break;
    }
  rotation = rotspeed * MODROLLERMHP_AXLE_LENGTH / 2.00;
  command_rightvel = translation + rotation;
  command_leftvel = translation - rotation;
  final_leftvel = (int)rint(command_leftvel / MODROLLERMHP_MPS_PER_TICK);
  final_rightvel = (int)rint(command_rightvel / MODROLLERMHP_MPS_PER_TICK);
  if(SetVelocity(final_leftvel, final_rightvel) != 0)
    {
      PLAYER_ERROR("ProcessVelCommand() failed");
      pthread_exit(NULL);
    }
  return(0);
}
int ModRollerMHP_Driver::PolarControlAlgorithm()
{
  double deltax = this->posx - this->localpositioncmd.pos.px;
  double deltay = this->posy - this->localpositioncmd.pos.py;
  double thetarelative = this->posa - this->localpositioncmd.pos.pa;
  double rotate_into_goal = this->localpositioncmd.pos.pa;
  double xrel = deltax*cos(rotate_into_goal) + deltay*sin(rotate_into_goal);
  double yrel = -deltax*sin(rotate_into_goal) + deltay*cos(rotate_into_goal);
  fprintf(this->xerror, "%f\n", deltax);
  fprintf(this->yerror, "%f\n", deltay);
  fprintf(this->therror, "%f\n", thetarelative);
  double rho;
  double gamma;
  double delta;
  double euclid_xy = deltax*deltax + deltay*deltay;
  if(this->gammadeltainit == false)
    {
      rho = sqrt(euclid_xy);
      gamma = atan2(yrel, xrel) - thetarelative + M_PI;
      delta = gamma + thetarelative;
      this->gamma_negative = gamma;
      this->delta_negative = delta;
      this->gammadeltainit = true;
    }
  rho = sqrt(euclid_xy);
  if(rho < 0.2)
    {
     gamma = this->gamma_negative;
     delta = this->delta_negative;
    }
  else
    {
      {
PLAYER_ERROR("aStemSetStream failed to set the stem packet processor");
      }
    if(!aModuleUtil_EnsureModule(this->stemref, 4))
      {
PLAYER_ERROR("Active Module Not Found");
return(-1);
      }
    char hb;
    err = aModuleVal_Get(this->stemref, 4, aMODULE_VAL_HBFLAG, &hb);
    if((err == aErrNone) && (hb != 1))
      err = aModuleVal_Set(this->stemref, 4, aMODULE_VAL_HBFLAG, 1);

    return(0);
}

int ModRollerMHP_Driver::GetOdometry(int* lt, int* rt, short* lv, short* rv)
{
  char buff[4];
  aErr err = aErrNone;

  err = aMotion_GetEnc32(this->stemref, 2, 0, buff);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve left encoder data %d", err);
    }
  *lt = aUtil_RetrieveInt((const char*)buff);

  err = aMotion_GetEnc32(this->stemref, 4, 1, buff);

  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve right encoder data", err);
    }
  *rt = aUtil_RetrieveInt((const char*)buff);

  err = aMotion_GetPIDInput(this->stemref, 2, 0, lv);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve left encoder vel feedback", err);
    }

  err = aMotion_GetPIDInput(this->stemref, 4, 1, rv);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve right encoder vel feedback", err);
    }

  return(0);
}

int ModRollerMHP_Driver::UpdateOdometry(int lt, int rt)
{
  int ltdelta, rtdelta;
  double l_delta, r_delta, a_delta, d_delta;
  double theta_hat;

  if(!this->odometryinitialised)
    {
      this->last_lticks = lt;
      this->last_rticks = rt;
      this->odometryinitialised = true;
      return(0);
    }

  ltdelta = lt - this->last_lticks;
  rtdelta = rt - this->last_rticks;

  l_delta = ltdelta * MODROLLERMHP_M_PER_TICK;
  r_delta = rtdelta * MODROLLERMHP_M_PER_TICK;
  a_delta = (r_delta - l_delta) / MODROLLERMHP_AXLE_LENGTH;
  d_delta = (l_delta + r_delta) / 2.00;

  // Implement 2nd order Runge-Kutta numerical integration
  // for pose update
  theta_hat = this->posa + (a_delta/2);
  this->posx += (d_delta * cos(theta_hat));
  this->posy += (d_delta * sin(theta_hat));
  this->posa += a_delta;

  this->posa = NORMALIZE(this->posa);

  this->last_lticks = lt;
  this->last_rticks = rt;

  return(0);
}

int ModRollerMHP_Driver::ClearAccumulator(const char channel)
{
  aErr err = aErrNone;
  aPacketRef packet;
  char data[aSTEMMAXPACKETBYTES];

  data[0] = cmdMO_ENC32;
  data[1] = (char)channel;
  data[2] = 0;
  data[3] = 0;
  data[4] = 0;
  data[5] = 0;

  aPacket_Create(this->stemref, 2, 6, data, &packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to create the ClearAccumulator packet");
      return(-1);
    }
  aStem_SendPacket(this->stemref, packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to send ClearAccumulator packet to module 2");
      return(-1);
    }
  aPacket_Create(this->stemref, 4, 6, data, &packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to create the ClearAccumulator packet");
      return(-1);
    }
  aStem_SendPacket(this->stemref, packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to send ClearAccumulator packet to module 1");
      return(-1);
    }
  return(0);
}

void ModRollerMHP_Driver::Main()
{
  player_position2d_data_t position_data;
  //player_power_data_t power_data;

  double leftvel_mps = 0.0;
  double rightvel_mps = 0.0;
  int left_ticks = 0;
  int right_ticks = 0;
  short leftvel = 0;
  short rightvel = 0;
  bool first_entry;
  bool exited;
  for(;;)
    {
      pthread_testcancel();
      ProcessMessages();
      if(this->GetOdometry(&left_ticks, &right_ticks, &leftvel, &rightvel))
{
  PLAYER_ERROR("Failed to retrieve Odometry data");
}
      else
UpdateOdometry(left_ticks, right_ticks);
      fprintf(this->xposition, "%f\n", this->posx);
      fprintf(this->yposition, "%f\n", this->posy);
      position_data.pos.px = this->posx;
      position_data.pos.py = this->posy;
      position_data.pos.pa = this->posa;
      position_data.vel.py = 0.0;
      leftvel_mps = leftvel * MODROLLERMHP_MPS_PER_TICK;
      rightvel_mps = rightvel * MODROLLERMHP_MPS_PER_TICK;
      position_data.vel.px = (leftvel_mps + rightvel_mps) / 2.00;
      position_data.vel.pa = (rightvel_mps - leftvel_mps) /
MODROLLERMHP_AXLE_LENGTH;
      position_data.stall = 0;
      this->Publish(this->position_addr, NULL, PLAYER_MSGTYPE_DATA,
    PLAYER_POSITION2D_DATA_STATE,
    (void*)&position_data, sizeof(player_position2d_data_t),
    NULL);
      pthread_mutex_lock(&this->controller_exit_mutex);
      first_entry = this->polar_controller_called;
      exited = this->position_target_reached;
      pthread_mutex_unlock(&this->controller_exit_mutex);
      if(first_entry == true)
{
  if(exited = true)
    {
      pthread_join(this->polarcontrolthread, NULL);
    }
}
      usleep(10000);
    }
}
int ModRollerMHP_Driver::thread_controller()
{
  pthread_create(&polarcontrolthread, NULL, &polarentryfunc, this);
  return(0);
}

int max(double v, double w)
{
  if(v > w)
    {
      if(v > 1)
{
  return(1);
}
      else
return(0);
    }
  if(w > v)
    {
      if(w > 1)
{
  return(2);
}
      else
return(0);
    }
  return(0);
}
void* ModRollerMHP_Driver::polarentryfunc(void* arg)
{
  ModRollerMHP_Driver* modrop = (ModRollerMHP_Driver*)arg;
  sleep(3);
  while(modrop->position_target_reached == false)
    {
      modrop->PolarControlAlgorithm();
      usleep(150000);
    }
  modrop->SetVelocity(0,0);
  puts("Exiting control thread");
  pthread_exit(NULL);
}
```

C.2 sonar Driver Code

Listed her is the code which allows RollerMHP's array of 12 SRF02 ultrasonic sensors to appear as an array of generic sonar devices.

C.2.1 Source File sonaracc.cc

```
/*
 *  Player - One Hell of a Robot Server
 *  Copyright (C) 2000
 *     Brian Gerkey, Kasper Stoy, Richard Vaughan, & Andrew Howard
 *
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
*/
// Driver which implememts the device abstractions for the ultrasonic sensors
// and the conveyor system of the RollerMHP Materials Handling Platform.
// Interfaces supported : sonar
//   actarray
```

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <aIO.h>
#include <aStem.h>
#include <aModuleUtil.h>
#include <aModuleVal.h>
#include <aModuleVM.h>
#include <aUtil.h>
#include <aServo.h>
#include <aPad.h>
#include <stdlib.h>
#include <aModule.tea>
#include <aCmd.tea>
#include <math.h>

#include <libplayercore/playercore.h>

const int numberofsonars = 12;

class SonarAcc : public Driver
{
 private:
    virtual void Main();

    // Integrated sonar and conveyor system data storage
    int Sonardata[numberofsonars];
    // For future support of the actuator array interface
    player_actarray_data_t conveyordata;
    player_actarray_geom_t conveyorgeom;

    player_sonar_data_t sonardata;
    player_sonar_geom_t sonargeom;

    //Device addresses
    player_devaddr_t sonar_address;
    // For future support of the actuator array interface
    player_devaddr_t conveyor_address;

    //References and stream abstractions
    aIOLib ioref;
    aStreamRef Linkstream;

    // Internal Methods
    int GetSonarData(int* sonar);
    int GetSonarPose(player_sonar_geom_t* sonargeometry);

    int OpenSerialPort();
    int GetAllReferences();

    // int GetConveyorState(aStemLib stemref, player_actarray_actuatorgeom_t* ptr);
 public:
    aStemLib stemref;
    const char* serial_port;

    SonarAcc(ConfigFile* cf, int section);

    int ProcessMessage(MessageQueue* resp_queue, player_msghdr* hdr, void* data);
    //int ProcessConveyorPosition(player_actarray_position_cmd_t* cmd);
    //int ProcessConveyorHome(player_actarray_home_cmd_t* cmd);

   virtual int Setup();
   virtual int Shutdown();
};

//Constructor
SonarAcc::SonarAcc(ConfigFile* cf, int section) : Driver(cf, section)
{

    memset(&this->sonar_address, 0, sizeof(player_devaddr_t));
    memset(&this->conveyor_address, 0, sizeof(player_devaddr_t));
    if(cf->ReadDeviceAddr(&(this->sonar_address), section, "provides",
  PLAYER_SONAR_CODE, -1, NULL) == 0);
    if(this->AddInterface(this->sonar_address))
    {
this->SetError(-1);
return;
    }
    if(cf->ReadDeviceAddr(&(this->conveyor_address), section, "provides",
  PLAYER_ACTARRAY_CODE, -1, NULL) == 0);
    if(this->AddInterface(this->conveyor_address) != 0)
    {
this->SetError(-1);
return;
    }
    this->serial_port = cf->ReadString(section, "port", "ttyUSB0");

static player_sonar_geom_t sonargeometry = {12, {{0.0, -0.236, (3*(M_PI/2))},
{0.0, 0.236, (M_PI/2)},
{0.236, 0.0, 0.0},
        {-0.236, 0.0, M_PI},
{0.204, 0.118, (M_PI/6)},
{-0.204, -0.118, (7*(M_PI/6))},
{0.204, -0.118, (11*(M_PI/6))},
{-0.204, 0.118, (5*(M_PI/6))},
{0.118, -0.204, (5*(M_PI/3))},
{-0.118, 0.204, (2*(M_PI/3))},
{0.118, 0.204, (M_PI/3)},
{-0.118, -0.204, (4*(M_PI/3))},}
};
this->sonargeom = sonargeometry;
}
//Driver Initialisation method
Driver*
SonarAcc_Init(ConfigFile* cf, int section)
{
    return ((Driver*)(new SonarAcc(cf, section)));
}
// Driver Registration Function
void
SonarAcc_Register(DriverTable* table)
{
    table->AddDriver("sonaracc", SonarAcc_Init);
}
// To avoid C++ name mangling
extern "C"
{
    int player_driver_init(DriverTable* table)
    {
SonarAcc_Register(table);
return(0);
    }
}
int
SonarAcc::GetSonarData(int* sonar)
{
    aErr err = aErrNone;
    int j = 0;
    for(int i = 0; i < 12; i++)
    {
      err = aPad_ReadInt(this->stemref, 2, j, &sonar[i]);
        this->sonardata.ranges[i] = (sonar[i]/100.0);
j= j +2;
//usleep(1000);
    }
    this->sonardata.ranges_count = 12;
    return(0);
}
int
SonarAcc::GetAllReferences()
{
    int err = aErrNone;
    if(aIO_GetLibRef(&this->ioref, NULL) != 0)
fprintf(stderr, "Failed to get the IO Library Reference\n");
    if(aStem_GetLibRef(&this->stemref, NULL) != 0)
fprintf(stderr, "Failed to get the Stem Library Reference\n");
    return(err);
}
int
SonarAcc::OpenSerialPort()
{
    if(aStream_CreateSerial(this->ioref, this->serial_port, 9600,
    &this->Linkstream, NULL))
fprintf(stderr, "Failed to Open serial port %s\n",
this->serial_port);
    if(aStem_SetStream(this->stemref, this->Linkstream, kStemModuleStream,
       NULL))
fprintf(stderr, "Failed to set the packet stream\n");
 // Set autoheartbeat stuff to ensure a healthy stream and watchdog safety
    if(!aModuleUtil_EnsureModule(this->stemref, 2))
    {
fprintf(stderr, "Active Module Not Found");
return(-1);
    }
    char heartbeat;
    int ret = aModuleVal_Get(this->stemref, 2, aMODULE_VAL_HBFLAG, &heartbeat);
    if((ret == 0) && (heartbeat != 1))
ret = aModuleVal_Set(this->stemref, 2, aMODULE_VAL_HBFLAG, 1);
    return(0);
}
int
SonarAcc::Setup()
{
    puts("SonarAcc Driver Initialising");
    if(GetAllReferences() !=0)
    {
fprintf(stderr, "Setup GetAllReferences failed");
return(-1);
    }
    if(OpenSerialPort() != 0)
    {
fprintf(stderr, "Setup OpenSerialPort Failed");
return(-1);
    }
    aModuleVal_Set(this->stemref, 2, aMODULE_VAL_IICBAUD, 1);
    StartThread();
    puts("SonarAcc Driver Initialised and awaiting commands");
    return(0);
}
int
SonarAcc::Shutdown()
{
puts("Shutting Down The sonaracc Driver on RollerMHP");
    StopThread();
    aModuleVal_Set(this->stemref, 2, aMODULE_VAL_IICBAUD, 2);
    aModuleVal_Set(this->stemref, 2, aMODULE_VAL_HBFLAG, 0);
    aStream_Destroy(this->ioref, this->Linkstream, NULL);
    if(this->ioref)
{
if(aIO_ReleaseLibRef(this->ioref, NULL) != 0)
 {
PLAYER_ERROR("Failed to release the IO library");
 }
}
if(this->stemref)
{
if(aStem_ReleaseLibRef(this->stemref, NULL) != 0)
    {
PLAYER_ERROR("Failed to release the Stem library");
  }
}
return(0);
}
int
SonarAcc::GetSonarPose(player_sonar_geom_t* sonargeometry)
{
  // Functionality handled internally by the ProcessMessage() method
    return(0);
}
int
SonarAcc::ProcessMessage(MessageQueue* resp_queue, player_msghdr* hdr, void*
 data)
{
    if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
     PLAYER_SONAR_REQ_GET_GEOM ,this->sonar_address))
    {
    this->Publish(this->sonar_address, resp_queue,
  PLAYER_MSGTYPE_RESP_ACK, PLAYER_SONAR_REQ_GET_GEOM,
  (void*)&this->sonargeom, sizeof(player_sonar_geom_t), NULL);
    return(0);
    }
    else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
  PLAYER_ACTARRAY_GET_GEOM_REQ,
  this->conveyor_address))
```

```
    {
// do future conveyor handler here
return(0);
    }
    else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_CMD,
  PLAYER_ACTARRAY_POS_CMD,
  this->conveyor_address))
    {
assert(hdr->size == sizeof(player_actarray_position_cmd_t));
this->SetConveyorPos(this->stemref,
  (player_actarray_position_cmd_t*)data);
return(0);
    }
    else
return(-1);
}

void
SonarAcc::Main()
{
    for(;;)
    {
pthread_testcancel();
ProcessMessages();
if(this->GetSonarData(this->Sonardata) != 0)
{
    fprintf(stderr, "GetSonarData Failed");
}

this->Publish(this->sonar_address, NULL, PLAYER_MSGTYPE_DATA,
      PLAYER_SONAR_DATA_RANGES, (void*)&this->sonardata,
      sizeof(player_sonar_data_t), NULL);
usleep(10000);
    }
}
```

Appendix D

Logic Based Switching Controller Implementation

Listed below is the code implementation of the logic based switching controller coverd in section 5.2.3. The controller was integrated into the RollerMHP_Driver driver during testing purposes. Currently, the code is being ported to an abstract driver.

D.1 Modified ModRollerMHP_Driver Code

Listed here is the driver code which implements the logic based switching controller by creating a Switcher object on the heap, which then implements the algorithms considered in section 5.2.3 to provide RollerMHP with posture stabilisation.

D.1.1 Header File modrollermhp.h

```
#ifndef _ModRollerMHP_Driver
#define _ModRollerMHP_Driver

#include <aIO.h>
#include <aStem.h>
#include <aPad.h>
#include <pthread.h>

#include <libplayercore/playercore.h>
#include "switcher.h"

// function used in maintaining curvature during saturation of control inputs
int max(double, double);

// forward declaration of Switcher Class
class Switcher;

class ModRollerMHP_Driver : public Driver
{
  // the Main thread
  virtual void Main();
  // address structures for the drivers interface specifications
  player_devaddr_t position_addr;
  player_devaddr_t power_addr;
  // internal odometry book keeping data
  int last_lticks;
  int last_rticks;
  bool odometryinitialised;
  // shared library references, BrainStem Libraries
  aStreamRef serialstream;

  aIOLib ioref;

  // the serial file stream abstraction
  aStemLib stemref;
  // Internal Methods
  int GetOdometry(int* lt, int* rt, short* lv, short* rv);
  int UpdateOdometry(int lt, int rt);
  int GetLibraries();
  int OpenTerminal();
  int ClearAccumulator(const char channel);
 public:
  // Internal odometric pose, public so that it is accessible from the Switcher
  // class
  double posx;
  double posy;
  double posa;
  // Pointer to the serial device file
  const char* serial_port;
  // public bool for keeping track of whether the controller has been
  // called and initialised, i.e. temporally active
  bool controller_called;
  // pointer to the underlying low-level switching based logic controller
  Switcher* controllerp;

  ModRollerMHP_Driver(ConfigFile* cf, int section);

  int ProcessMessage(MessageQueue* resp_queue,
     player_msghdr* hdr,
     void* data);
  int ProcessVelCommand(player_position2d_cmd_vel_t* cmd);
  // StartController() calls ExecuteControl() through the controllerp pointer
  int StartController();

  virtual int Setup();
  virtual int Shutdown();
  int SetVelocity(int, int);
};
#endif // ModRollerMHP_Driver
```

D.1.2 Source File modrollermhp.cc

```
/*
 *  Player - One Hell of a Robot Server
 *  Copyright (C) 2000
 *     Brian Gerkey, Kasper Stoy, Richard Vaughan, & Andrew Howard
 *
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <aModuleUtil.h>
```

```
#include <aMotion.h>
#include <aModuleVal.h>
#include <aUtil.h>
#include <aModule.tea>
#include <aCmd.tea>
#include <aModuleVM.h>
#include <aErr.h>

#include "modrollermhp.h"
#include "modrollermhp_constants.h"
#include "switcher.h"

// Forward declaration of Switcher class

class Switcher;

Driver* ModRollerMHP_Driver_Init(ConfigFile* cf, int section)
{
  return ((Driver*)(new ModRollerMHP_Driver(cf, section)));
}

void ModRollerMHP_Driver_Register(DriverTable* table)
{
  table->AddDriver("modrollermhp", ModRollerMHP_Driver_Init);
}

extern "C"
{
  int player_driver_init(DriverTable* table)
  {
    ModRollerMHP_Driver_Register(table);
    return(0);
  }
}

ModRollerMHP_Driver::ModRollerMHP_Driver(ConfigFile* cf, int section)
: Driver(cf, section)
{
  memset(&this->position_addr, 0, sizeof(player_devaddr_t));
  memset(&this->power_addr, 0, sizeof(player_devaddr_t));

  if(cf->ReadDeviceAddr(&this->position_addr, section, "provides",
   PLAYER_POSITION2D_CODE, -1, NULL) == 0)
     {
       if(AddInterface(this->position_addr) != 0)
{
  this->SetError(-1);
  return;
}
     }

  if(cf->ReadDeviceAddr(&this->power_addr, section, "provides",
PLAYER_POWER_CODE, -1, NULL) == 0)
     {
       if(AddInterface(this->power_addr) != 0)
{
  this->SetError(-1);
  return;
}
     }

  this->serial_port = cf->ReadString(section, "port",
     MODROLLERMHP_DEFAULT_SERIALPORT);
}

int ModRollerMHP_Driver::Setup()
{

  puts("ModRollerMHP_Driver Initialising...");

  this->posx = 0.0;
  this->posy = 0.0;
  this->posa = 0.0;
  this->odometryinitialised = false;
  this->controller_called = false;
  if(GetLibraries() != 0)
    {
      PLAYER_ERROR("Failed To Gain Shared Library References\n");
    }
  if(OpenTerminal() != 0)
    {
      PLAYER_ERROR("Failed To Create Serial Stream\n");
    }
  this->ClearAccumulator(0);
  this->ClearAccumulator(1);
  // Start the Main() thread... calls dummy static funcs etc.
  StartThread();
  // Create a logic based switching controller on the heap
  this->controllerp = new Switcher(this);
  puts("ModRollerMHP_Driver initialised");
  return(0);
}
int ModRollerMHP_Driver::Shutdown()
{
  puts("Releasing the ModRollerMHP_Driver...");
  // Set the heartbeat to link dependant
  aModuleVal_Set(this->stemref, 4, aMODULE_VAL_HBFLAG, 0);
  // Free up heap memory used by the logic based switching controller
  delete this->controllerp;
  StopThread();
  if(this->ioref)
    {
      if(aIO_ReleaseLibRef(this->ioref, NULL) != 0)
{
  PLAYER_ERROR("Failed To Release The IO Library");
}
    }
  if(this->stemref)
    {
      if(aStem_ReleaseLibRef(this->stemref, NULL))
{
  PLAYER_ERROR("Failed To Release The Stem Library Reference");
}
    }
  puts("ModRollerMHP_Driver is shutdown.");
  return(0);
}
int ModRollerMHP_Driver::ProcessMessage(MessageQueue* resp_queue,
player_msghdr* hdr,
void* data)
{
  if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_CMD,
   PLAYER_POSITION2D_CMD_VEL,
   this->position_addr))
    {
      assert(hdr->size = sizeof(player_position2d_cmd_vel_t));
      this->ProcessVelCommand((player_position2d_cmd_vel_t*)data);
      return(0);
    }
  else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_CMD,
PLAYER_POSITION2D_CMD_POS,
this->position_addr))
    {
      /*assert(hdr->size = sizeof(player_position2d_cmd_pos_t));
      this->controllerp->throughputcmd = *(player_position2d_cmd_pos_t*)data;
      if(this->controller_called == false)
{
  puts("Starting/Restarting the Switching Controller");
  this->StartController();
  this->controller_called = true;
  return(0);
  }*/
      return(0);
      }
  else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
   PLAYER_POSITION2D_REQ_GET_GEOM,
   this->position_addr))
    {
      assert(hdr->size = sizeof(player_position2d_geom_t));
      player_position2d_geom_t geom;
      geom.pose.px = 0.0;
      geom.pose.py = 0.0;
      geom.pose.pa = 0.0;
      geom.size.sl = 0.508;
      geom.size.sw = 0.610;
      this->Publish(this->position_addr, resp_queue, PLAYER_MSGTYPE_RESP_ACK,
    PLAYER_POSITION2D_REQ_GET_GEOM,
    (void*)&geom, sizeof(player_position2d_geom_t), NULL);
      return(0);
    }
  else if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ,
PLAYER_POSITION2D_REQ_RESET_ODOM,
this->position_addr))
    {
      if(hdr->size != sizeof(player_position2d_reset_odom_config_t))
{
  PLAYER_WARN("Arguement to req reset odom is wrong size");
  return(-1);
}
      this->ClearAccumulator(0);
      this->ClearAccumulator(1);
      this->posx = 0;
      this->posy = 0;
      this->posa = 0;
      this->Publish(this->position_addr, resp_queue,
    PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_RESET_ODOM);
      return(0);
    }
  else
  return(-1);
}
int ModRollerMHP_Driver::SetVelocity(int lv, int rv)
{
  aErr err = aErrNone;
  err = aPad_WriteInt(this->stemref, 4, 0, lv);
  err = aPad_WriteInt(this->stemref, 2, 2, rv);
  if(err != aErrNone)
    {
      PLAYER_ERROR("SetVelocity() Failed");
      return(-1);
    }
  return(0);
}
int ModRollerMHP_Driver::ProcessVelCommand(player_position2d_cmd_vel_t* cmd)
{
  double rotation = 0.0;
  double command_leftvel = 0.0;
  double command_rightvel = 0.0;
  int final_leftvel = 0;
  int final_rightvel = 0;
  double translation = cmd->vel.px;
  double rotspeed = cmd->vel.pa;
  double omega = rotspeed/MODROLLERMHP_MAXROTSPEED;
  double v = translation/MODROLLERMHP_MAXTRANS;
  int sigma = max(fabs(v), fabs(omega));
  switch(sigma)
    {
    case 1:
      {
if(translation > 0)
  translation = MODROLLERMHP_MAXTRANS;
else
  translation = -MODROLLERMHP_MAXTRANS;
rotspeed = rotspeed/fabs(v);
break;
      }
    case 2:
      {
if(rotspeed > 0)
  rotspeed = MODROLLERMHP_MAXROTSPEED;
else
  rotspeed = -MODROLLERMHP_MAXROTSPEED;
translation = translation/fabs(omega);
break;
      }
    default:
      break;
    }
  rotation = rotspeed * MODROLLERMHP_AXLE_LENGTH / 2.00;
  command_rightvel = translation + rotation;
  command_leftvel = translation - rotation;
  final_leftvel = (int)rint(command_leftvel / MODROLLERMHP_MPS_PER_TICK);
  final_rightvel = (int)rint(command_rightvel / MODROLLERMHP_MPS_PER_TICK);
  if(SetVelocity(final_leftvel, final_rightvel) != 0)
    {
      PLAYER_ERROR("ProcessVelCommand() failed");
      pthread_exit(NULL);
    }
  return(0);
}
int ModRollerMHP_Driver::StartController()
```

```
{
  puts("StartController()");
  this->controllerp->ExecuteControl();
  return(0);
}
int ModRollerMHP_Driver::GetLibraries()
{
  aErr err = aErrNone;
  aIO_GetLibRef(&this->ioref, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("aIO_GetLibRef() failed ");
    }
  aStem_GetLibRef(&this->stemref, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("aStem_GetLibRef() failed ");
    }
  return((int)err);
}
int ModRollerMHP_Driver::OpenTerminal()
{
  aErr err = aErrNone;
  aStream_CreateSerial(this->ioref, this->serial_port, 9600,
       &this->serialstream, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("aStream_CreateSerial() failed: %s\n", strerror(errno));
      PLAYER_ERROR("BrainStem's Serial Connection Failed\n");
      aStem_ReleaseLibRef(this->stemref, &err);
      return(-1);
    }
  aStem_SetStream(this->stemref, this->serialstream,
  kStemModuleStream, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("aStem_SetStream(): BrainStem Packet Processor Failed");
    }
  if(!aModuleUtil_EnsureModule(this->stemref, 4))
    {
      PLAYER_ERROR("BrainStem Module Not Active\n");
    }
  char heartbeat;
  err = aModuleVal_Get(this->stemref, 4, aMODULE_VAL_HBFLAG, &heartbeat);
  if((err == aErrNone) && (heartbeat != 1))
    {
      aModuleVal_Set(this->stemref, 4, aMODULE_VAL_HBFLAG, 1);
    }
  return((int)err);
}
int ModRollerMHP_Driver::GetOdometry(int* lt, int* rt, short* lv, short* rv)
{
  char buff[4];
  aErr err = aErrNone;
  err = aMotion_GetEnc32(this->stemref, 4, 0, buff);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve left encoder data %d", err);
    }
  *lt = aUtil_RetrieveInt((const char*)buff);
  err = aMotion_GetEnc32(this->stemref, 2, 1, buff);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve right encoder data", err);
    }
  *rt = aUtil_RetrieveInt((const char*)buff);
  err = aMotion_GetPIDInput(this->stemref, 4, 0, lv);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve left encoder vel feedback", err);
    }
  err = aMotion_GetPIDInput(this->stemref, 2, 1, rv);
  if(err != aErrNone)
    {
      PLAYER_ERROR1("Failed to retrieve right encoder vel feedback", err);
    }
  return(0);
}
int ModRollerMHP_Driver::UpdateOdometry(int lt, int rt)
{
  int ltdelta, rtdelta;
  double l_delta, r_delta, a_delta, d_delta;
  double theta_hat;
  if(!this->odometryinitialised)
    {
      this->last_lticks = lt;
      this->last_rticks = rt;
      this->odometryinitialised = true;
      return(0);
    }
  ltdelta = lt - this->last_lticks;
  rtdelta = rt - this->last_rticks;
  l_delta = ltdelta * MODROLLERMHP_M_PER_TICK;
  r_delta = rtdelta * MODROLLERMHP_M_PER_TICK;
  a_delta = (r_delta - l_delta) / MODROLLERMHP_AXLE_LENGTH;
  d_delta = (l_delta + r_delta) / 2.00;
  // Implement 2nd order Runge-Kutta numerical integration
  // for pose update
  theta_hat = this->posa + (a_delta/2);
  this->posx += (d_delta * cos(theta_hat));
  this->posy += (d_delta * sin(theta_hat));
  this->posa += a_delta;
  this->posa = NORMALIZE(this->posa);
  this->last_lticks = lt;
  this->last_rticks = rt;
  return(0);
}
int ModRollerMHP_Driver::ClearAccumulator(const char channel)
{
  aErr err = aErrNone;
  aPacketRef packet;
  char data[aSTEMMAXPACKETBYTES];

  data[0] = cmdMO_ENC32;
  data[1] = (char)channel;
  data[2] = 0;
  data[3] = 0;
  data[4] = 0;
  data[5] = 0;

  aPacket_Create(this->stemref, 2, 6, data, &packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to create the ClearAccumulator packet");
      return(-1);
    }
  aStem_SendPacket(this->stemref, packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to send ClearAccumulator packet to the module");
      return(-1);
    }

  aPacket_Create(this->stemref, 4, 6, data, &packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to create the ClearAccumulator packet");
      return(-1);
    }
  aStem_SendPacket(this->stemref, packet, &err);
  if(err != aErrNone)
    {
      PLAYER_ERROR("Failed to send ClearAccumulator packet to the module");
      return(-1);
    }
  return(0);
}
void ModRollerMHP_Driver::Main()
{
  player_position2d_data_t position_data;

  // power interface is disabled for now
  // player_power_data_t power_data;

  double leftvel_mps = 0.0;
  double rightvel_mps = 0.0;
  int left_ticks = 0;
  int right_ticks = 0;

  short leftvel = 0;
  short rightvel = 0;

  for(;;)
    {
      pthread_testcancel();
      ProcessMessages();

      if(this->GetOdometry(&left_ticks, &right_ticks, &leftvel, &rightvel))
{
  PLAYER_ERROR("Failed to retrieve Odometry data");
}
      else
UpdateOdometry(left_ticks, right_ticks);

      position_data.pos.px = this->posx;
      position_data.pos.py = this->posy;
      position_data.pos.pa = this->posa;
      position_data.vel.py = 0.0;
      leftvel_mps = leftvel * MODROLLERMHP_MPS_PER_TICK;
      rightvel_mps = rightvel * MODROLLERMHP_MPS_PER_TICK;
      position_data.vel.px = (leftvel_mps + rightvel_mps) / 2.00;
      position_data.vel.pa = (rightvel_mps - leftvel_mps) /
MODROLLERMHP_AXLE_LENGTH;
      position_data.stall = 0;
      this->Publish(this->position_addr, NULL, PLAYER_MSGTYPE_DATA,
    PLAYER_POSITION2D_DATA_STATE,
    (void*)&position_data, sizeof(player_position2d_data_t),
    NULL);
      usleep(15000);
    }
}

int max(double v, double w)
{
  if(v > w)
    {
      if(v > 1)
{
  return(1);
}
      else
return(0);
    }
  else if(w > v)
    {
      if(w > 1)
{
  return(2);
}
      else
return(0);
    }
  else
    return(0);
}
```

D.1.3 Switcher Header File switcher.h

```
#ifndef _Switcher
#define _Switcher
#include "modrollermhp.h"
#include <libplayercore/playercore.h>
#include <pthread.h>
#include <cstdio>
class ModRollerMHP_Driver;
class Switcher {
  // storage for x1, x2 and x3. The diffeomorphic state variables
  double nonholo_int_state[3];
  double relative_configuration[3];
  // switch variable for triggering an exit status
  int target_reached;
  // the discrete image and left limit of the switching signal
  int sigma;
  int sigma_negative;
  bool sigma_initialised;
  // Logfile Handles
  FILE* region_data_w1;
  FILE* region_data_w2;
  FILE* sigma_image_history;
  FILE* x_error;
  FILE* y_error;
  FILE* th_error;

  // a variable for holding control reqion status
  int control_region;

  // mutex for controlled access to controller internals
  pthread_mutex_t access_mutex;
  pthread_mutex_t sigma_and_sigma_neg;

 public:

  // pointer to the respective position2d interface
  ModRollerMHP_Driver* modrollerp;
  // thread id for the state tracking thread
  pthread_t sigma_transition;
  // thread id for the control implementation thread
  pthread_t control_application;

  // local storage of the goal point to converge onto
  player_position2d_cmd_pos_t throughputcmd;

  // And here begins the declaration of methods used to implement the algorithm
  // update the internal diffeomorphic state
  int UpdateStateInternal();
  // determines the current state reqion
  int Update_Control_Region(double, double, double);
  // the switching functions image manipulator f: R^3 x S -> S
  int Phi();
  // function to handle the setting of control inputs u = g_(x) : R^3 x S -> R^2
  int Control();
  // thread entry function for diffeomorphic state tracking
  static void* phi_R_sigma_neg(void*);
  // thread entry for control signal implementation
  static void* U_g_sigma_x(void*);
  // function for getting the controller up and running
  int ExecuteControl();

  // Constructs onto the underlying 'ModrollerMHP_Driver' driver
  Switcher(ModRollerMHP_Driver*);
  ~Switcher();
};

#endif // Switcher
```

D.1.4 Switcher Source File switcher.cc

```
/*
 *  Player - One Hell of a Robot Server
 *  Copyright (C) 2000
 *     Brian Gerkey, Kasper Stoy, Richard Vaughan, & Andrew Howard
 *
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
*/
/* Multi threaded C++ implementation of the logic based switching controller
 *
 *
 * Author: Anthony John Walker, awalker@ukzn.ac.za
*/
#include "switcher.h"
#include <math.h>
#include <unistd.h>
#include <string.h>
#include <cstdio>

// Forward declaration of ModRollerMHP_Driver class for reference semantics
class ModRollerMHP_Driver;

Switcher::Switcher(ModRollerMHP_Driver* modro)
{
  puts("Commisioning low-level Controller...");

  this->sigma_initialised = false;

  // Gain reference to the associated ModRollerMHP_Driver
  this->modrollerp = modro;

  pthread_mutex_init(&this->access_mutex, NULL);
  pthread_mutex_init(&this->sigma_and_sigma_neg, NULL);

  this->target_reached = 0;

  memset(&this->throughputcmd, 0, sizeof(player_position2d_cmd_pos_t));

  region_data_w1 =
    fopen("/home/awalker/development/ModRollerMHP/logdata/w1.txt", "w+");

  region_data_w2 =
    fopen("/home/awalker/development/ModRollerMHP/logdata/w2.txt", "w+");

  sigma_image_history =
    fopen("/home/awalker/development/ModRollerMHP/logdata/sigma.txt", "w+");

  x_error =
    fopen("/home/awalker/development/ModRollerMHP/logdata/error_x.txt", "w+");

  y_error =
    fopen("/home/awalker/development/ModRollerMHP/logdata/error_y.txt", "w+");

  th_error =
    fopen("/home/awalker/development/ModRollerMHP/logdata/error_th.txt", "w+");

  puts("Low-Level Controller Has Been Commissioned");
}

Switcher::~Switcher()
{
  puts("Uncommisioning Low-Level Controller");

  fclose(this->region_data_w1);
  fclose(this->region_data_w2);
  fclose(this->sigma_image_history);
  fclose(this->x_error);
  fclose(this->y_error);
  fclose(this->th_error);

  if(this->target_reached == 1)
    {
      puts("joining any 'zombie' threads");
      pthread_join(this->sigma_transition, NULL);
      pthread_join(this->control_application, NULL);
    }

  pthread_mutex_destroy(&this->access_mutex);
  pthread_mutex_destroy(&this->sigma_and_sigma_neg);

  puts("Done");
```

```
}
int Switcher::UpdateStateInternal()
{
  // Calculate the relative disturbance input, i.e the error relative to goal
  // pose
  double deltax = this->modrollerp->posx - this->throughputcmd.pos.px;
  double deltay = this->modrollerp->posy - this->throughputcmd.pos.py;
  double thetarel = this->modrollerp->posa - this->throughputcmd.pos.pa;
  fprintf(this->x_error, "%f\n", deltax);
  fprintf(this->y_error, "%f\n", deltay);
  fprintf(this->th_error, "%f\n", thetarel);
  double rotate_theta_into_goal = this->throughputcmd.pos.pa;

  double xrel = deltax*cos(rotate_theta_into_goal) +
    deltay*sin(rotate_theta_into_goal);
  double yrel = -deltax*sin(rotate_theta_into_goal) +
    deltay*cos(rotate_theta_into_goal);

  // Apply a diffeomorphism to the configuration state
  // in order to produce the generic "nonholonomic integrator"
  // described by Rodger Brockett
  double x1 = xrel*cos(thetarel) + yrel*sin(thetarel);
  double x2 = thetarel;
  double x3 = 2*(xrel*sin(thetarel) - yrel*cos(thetarel)) - thetarel*
    (xrel*cos(thetarel) + yrel*sin(thetarel));

  // Lock access to the controller internal when updating the state
  pthread_mutex_lock(&this->access_mutex);

  this->nonholo_int_state[0] = x1;
  this->nonholo_int_state[1] = x2;
  this->nonholo_int_state[2] = x3;

  this->relative_configuration[0] = xrel;
  this->relative_configuration[1] = yrel;
  this->relative_configuration[2] = thetarel;

  // Unlock the mutex as we are finished with the important data
  pthread_mutex_unlock(&this->access_mutex);

  // Expose the current control region
  this->Update_Control_Region(x1, x2, x3);

  return(0);
}
// This function propagates the sigma image and is defined as a mapping
// from R^3 x S -> S. sigma = phi(x,sigma_left_lim) : R^3 x S -> S
int Switcher::Phi()
{

  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  int sigma_left_lim = this->sigma_negative;
  int region = this->control_region;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);

  if(sigma_left_lim != region)
    {
      puts("Sigma Image Switch...");
      pthread_mutex_lock(&this->sigma_and_sigma_neg);
      this->sigma = this->control_region;
      this->sigma_negative = this->sigma;
      pthread_mutex_unlock(&this->sigma_and_sigma_neg);
      fprintf(this->sigma_image_history, "%d\n", this->sigma);
      return(0);
    }
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  this->sigma = this->sigma_negative;
  this->sigma_negative = this->sigma;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  fprintf(this->sigma_image_history, "%d\n", this->sigma);
  return(0);
}
int Switcher::Control()
{
  double ControlInputs[2];
  pthread_mutex_lock(&this->access_mutex);
  double x1 = this->nonholo_int_state[0];
  double x2 = this->nonholo_int_state[1];
  double x3 = this->nonholo_int_state[2];
  double X = this->relative_configuration[0];
  double Y = this->relative_configuration[1];
  double Th = this->relative_configuration[2];
  pthread_mutex_unlock(&this->access_mutex);
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  int sigma =  this->sigma;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  switch(sigma) {
  case 1:
      ControlInputs[0] = 1;
      ControlInputs[1] = 1;
      break;
  case 2:
      ControlInputs[0] = x1 + (x2*x3)/(x1*x1 + x2*x2);
      ControlInputs[1] = x2 - (x1*x3)/(x1*x1 + x2*x2);
      break;
  case 3:
      ControlInputs[0] = -x1 + (x2*x3)/(x1*x1 + x2*x2);
      ControlInputs[1] = -x2 - (x1*x3)/(x1*x1 + x2*x2);
      break;
  case 4:
      ControlInputs[0] = 0;
      ControlInputs[1] = 0;
      break;
  default:
    break;
  }
  player_position2d_cmd_vel_t cmd;
  cmd.vel.px = ControlInputs[0] + ControlInputs[1]*(X*sin(Th) - Y*cos(Th));
  cmd.vel.pa = ControlInputs[1];
  cmd.vel.px *= 0.4;
  cmd.vel.pa *= 0.4;
  modrollerp->ProcessVelCommand(&cmd);
  return(0);
}
int Switcher::Update_Control_Region(double x1, double x2, double x3)
{
  // Create and store the region variables
  double w1 = x3*x3;
  double w2 = x1*x1 + x2*x2;
  // For data logging and debugging purposes
  fprintf(this->region_data_w1, "%f\n", w1);
  fprintf(this->region_data_w2, "%f\n", w2);
  double regiongain = 0.5;
  // Develop the current image values of the control regions
  double pi_1 = regiongain*1*(1 - exp(-sqrt(w1)));
  double pi_2 = regiongain*1.5*pi_1;
  double pi_3 = regiongain*2*pi_1;
  double pi_4 = regiongain*4*pi_1;
  double epsilon = sqrt(w1 + w2);
  // Initialise the sigma image on first call to controller
  if(this->sigma_initialised = false)
    {
      if(0 <= w2 < pi_1)
{
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  this->sigma_negative = 1;
  this->control_region = 1;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  epsilon = 1;
  this->sigma_initialised = true;
}
      else if(pi_1 <= w2 < pi_3)
{
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  this->sigma_negative = 2;
  this->control_region = 2;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  epsilon = 1;
  this->sigma_initialised = true;
}
      else if(w2 >= pi_3)
{
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  this->sigma_negative = 3;
  this->control_region = 3;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  epsilon = 1;
  this->sigma_initialised = true;
}
      else
{
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  this->sigma_negative = 4;
  this->control_region = 4;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  epsilon = 1;
  this->sigma_initialised = true;
}
    }
  pthread_mutex_lock(&this->sigma_and_sigma_neg);
  int local_sigma_neg = this->sigma_negative;
  pthread_mutex_unlock(&this->sigma_and_sigma_neg);
  // Set current control region
  if(0 <= w2 < pi_2)
    {
      if((w2 > pi_1) && (local_sigma_neg == 2))
{
  this->control_region = 2;
}
      else
{
  this->control_region = 1;
}
    }
  else if(pi_2 <= w2 < pi_4)
    {
      if((w2 > pi_3) && (local_sigma_neg == 3))
{
  this->control_region = 3;
}
      else
{
  this->control_region = 2;
}
    }
  else if(w2 >= pi_4)
    {
      this->control_region = 3;
    }
  else
    {
      this->control_region = 4;
    }
  // Update sigma image
  this->Phi();
  if(epsilon < 0.1)
    {
      this->target_reached = 1;
      this->sigma_initialised = false;
    }
  return(0);
}
int Switcher::ExecuteControl()
{
  // Thread the handler functions to start the ball rolling
  this->sigma_initialised = false;
  pthread_create(&sigma_transition, NULL, &phi_R_sigma_neg, this);
  pthread_create(&control_application, NULL, &U_g_sigma_x, this);
  return(0);
}
void* Switcher::phi_R_sigma_neg(void* arg)
{
  Switcher* control = (Switcher*)arg;
  sleep(2);
  while(control->target_reached == 0)
  {
      control->UpdateStateInternal();
      usleep(150000);
  }
  pthread_exit(NULL);
}
void* Switcher::U_g_sigma_x(void* arg)
{
  Switcher* control = (Switcher*)arg;
  sleep(2);
  while(control->target_reached == 0)
  {
      control->Control();
      usleep(150000);
  }
  pthread_exit(NULL);
}
```

D.2 Configuration File for RollerMHP

```
driver
(
name "modrollermhp"
  plugin "/home/awalker/development/ModRollerMHP/libmodrollermhp.so"
  provides ["position2d:0" "power:0"]
port "ttyS1"
alwayson 0
)
driver
(
name "sonaracc"
plugin "/home/awalker/development/MHPsonaractuator/libsonaracc.so"
provides ["sonar:0" "actarray:0"]
port "ttyS0"
alwayson 0
)
driver
(
name "vfh"
provides ["position2d:1"]
requires ["position2d:0" "sonar:0"]
distance_epsilon 0.3
angle_epsilon 5
max_speed 0.15
max_acceleration 0.1
safety_dist_0ms 0.4
free_space_cutoff_0ms 500000.0
)
```

D. LOGIC BASED SWITCHING CONTROLLER IMPLEMENTATION

References

[1] Mobile Robots Inc. [Accessed 5 September 2007]. http://www.mobilerobots.com. 47

[2] RFC1832 XDR: External Data Representation Standard. [Accessed 27 November 2008]. http://www.faqs.org/rfcs/rcf1832.html. 48

[3] J. Balakrishnan. The dynamics of plant layout. *Management Science*, 39(5):654–655, 1993. 9

[4] Irani. S.A Benjafaar. S, Heragu. S.S. Next generation factory layouts: Research challenges and recent progress. *Interfaces*, 32(6):58–76, 2002. 9

[5] MacDonald. B. Biggs. G. Generic interfaces for robotic limbs. *In proc. of the Australasian Conference on Robotics and Automation*, 2006. 52

[6] McClamroch. N.H Bloch. A.M, Reyhanoglu. M. Control and stabilization of nonholonomic dynamic systems. *IEEE Transactions on Automaic Control*, 37:1746–1757, 1992. 62

[7] Koren. Y. Borenstein. J. Real-time map-building for fast mobile robot obstacle avoidance. *SPIE Symposium on Advances in Intelligent Systems, Mobile Robots, V*, 1990. 59

[8] Koren. Y. Borenstein. J. Histogramic in-motion mapping for mobile robot obstacle avoidance. *IEEE journal of Robotics and Automation*, 7(4):535–539, 1991. 59

[9] Koren. Y. Borenstein. J. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE journal of Robotics and Automation*, 7(3):278–288, 1991. 59

[10] Ulrich. I. Borenstein. J. Vfh+: Reliable obstacle avoidance for fast mobile robots. *In proceedings of the 1998 IEEE International Conference on Robotics and Automation*, pages 1572–1577, 1998. 59

[11] Zavanella. L. Braglia. M, Zanoni. S. Layout design in dymanic environments: Strategies and quantitative indices. *Internat. J. Production Res. Forthcomming*, 2002. 10

[12] Balestrino. A. Casalino. A.M, Bicchi. A. Closed loop steering of unicycle-like vehicles via lyapunov techniques. *IEEE Robotics & Automation Magazine*, 2:27–35, 1995. 76

[13] Gerkey. B. Collet. T, MacDonald. B. Player 2.0: Toward a practical robot programming framework. *In proceedings of the Australasian Conference on Robotics and Automation, ACRA*, 2005. 48

[14] S. Davis. *Future Perfect*. Basic Books, 1997. 6

[15] Vendittelli. M. De Luca. A, Oriolo. G. Control of wheeled mobile robots: An experimental overview. *in RAMSETE: Articulated and Mobile Robots for SErvices and TEchnology*, 270, 2000. 29, 63, 65, 76, 77

[16] Thrun et al. Monte-carlo localization for mobile robots. *In proc. of the International Conference on Robotics and Automation*, pages 1322–1328, 1999. 59

[17] Thrun et al. Simultaneous localisation and mapping with sparse extended information filters. *International Journal of Robotics Research*, 23(7):693–716, 2004. 59

[18] Tseng et al. Generic bill-of-materials-and-operations for high variety production management. *Concurrent Engineering: Research and Applications*, 8(4):297–322, 2000. 10

[19] M.P Groover. *Automation, Production Systems and Computer Integrated Manufacturing.* Prentice Hall, 2nd edition, 2000. 9, 10

[20] Wortmann. J.C. Hegge. H.M.H. Generic bill-of-material: A new product model. *International Journal of Production Economics*, 23:117–128, 1991. 10

[21] Stephen Morse. A. Hespanha. J.P. Stabilization of nonholonomic integrators via logic based switching. *Automatica*, 35:385–393, 1999. 65, 67, 68, 69

[22] Stephen Morse. A. Hespanha. J.P, Liberzon. D. Logic based switching control of a nonholonomic system with parametric modeling uncertainty. *Syst. & Contr. Lettr, Special Issue of Hybrid Systems*, 38:167–177, 1999. 67

[23] Huang. H. Irani. S.A. Custom design of facility layouts for multi-product facilities using layout modules. *IEEE Trans. on Robotics and Automation*, 16:259–267, 2000. 10

[24] Balakrishnan. J. and C.H. Cheng. Dynamic plant layout algorithms: A state of the art survey. *Omega*, 26(4):507–521, 1998. 9

[25] Tseng. M.M. Jiao. J. Design for mass customization. *Annuals of the CIRP*, 45(1):153–156, 1996. 6

[26] Tseng. M.M. Jiao. J. An information modeling framework for product families to support mass customization manufacturing. *Annuals of the CIRP*, 48(1):93–98, 1999. 10

[27] Tseng. M.M Jiao. J. Customizability analysis in design for mass customization. *Annuals of the CIRP*, 36(8):745–757, 2004. 6

[28] Pars. L.A. *A Treatise on Analytical Dynamics.* Heinemann, 1965. 63

[29] Newman. P. Leong Ho. K. Detecting loop closure with scene sequences. *International Journal of Computer Vision*, 74(3):261–286, 2007. 59

[30] [Accessed 21 August 2007] Logic Supply. http://www.logicsupply.com. 42

[31] Yannier S. Sabanovic A. Onat A. Bastan M. Sliding mode based behaviour control. *Proceedings of the World Academy of Science, Engineering and Technology*, 3:118–121, 2005. 66

[32] J.H. Mikkola. Capturing the degree of modularity embedded in product architectures. *Journal of Product Innovation Management*, 23:128–146, 2006. 7

[33] J.H. Mikkola. Measuring the degree of mass customization: A product architecture modularization perspective. *POMS* 18^th *Annual Conference*, 2007. 7

[34] Sastry. S. S. Murray. R.M. Nonholonomic motion planning: Steering using sinusoids. *IEEE Transactions on Automaic Control*, 38:700–716, 1993. 66

[35] Nise. N. *Control Systems Engineering.* John Wiley & Sons, 4th edition, 2004. 61

[36] Listed on Frank Pillar's Web Site. http://www.mass-customization.de/glossary.htm#mc. 6

[37] Acroname Incorporated online website [Accessed 10 July 2008]. http://www.acroname.com. 36, 91, 93

[38] Image reference from website [Accessed 5 February 2008]. http://www.flickr.com/photos/72038961N00/383482529/. 20

[39] Brockett. R.W. Asymptotic stability and feedback stabilization. *In R.Brockett, R. Millman, H. Sussmann, "Differerntial geometic control theory", Birkhauser*, pages 181–191, 1983. 65, 66

[40] Lavalle. S. *Planning Algorithms*. Cambridge University Press, 2006. 60, 61, 63, 64, 66

[41] J. Shoval. S. Borenstein. Using coded signals to benefit from ultrasonic sensor crosstalk in mobile robot obstacle avoidance. *IEEE International Conference on Robotics and Automation, Seoul Korea*, pages 2879–2884, 2001. 35

[42] Jarvis. R. Spero. D. Towards exteroceptive based localisation. *Proceedings of the 2004 IEEE Conference on Robotics, Automation and Mechatronics*, 2004. 59

[43] [Accessed 14 April 2007] Technical listing on Devantech website. http://www.robot-electronics.co.uk/htm/md03tech.htm. 25, 35

[44] [Accessed 20 November 2008] The official website of the Player/Stage/Gazebo project. http://playerstage.sourceforge.net. 44, 45, 52

[45] A. Toffler. *Future Shock*. Bantam Books, ISBN: 0553277375, 1971. 6

[46] Bolton. W. *Mechatronics*. Pearson, Prentice Hall, 2003. 20

[47] Bright. G Walker. A.J. Standardised framweork for flexible materials handling management based on operating system primitives. *In Proc. of the Australasian Conference on Robotics and Automation, ACRA*, 2007. 13

[48] G. Wang, D. Xu. Full-state tracking and internal dynamics of nonholonomic wheeled mobile robots. *IEEE/ASME Transactions on Mechatronics*, 8(2):203–214, 2003. 66

[49] Tseng. M.M. Wang. Y. Incorporating probabilistic model of customers' preferences in concurrent engineering. *in Annuals of the CIRP - Manufacturing Technology*, 57:137–140, 2008. 85