The Implementation of a Hierarchical Hybrid Navigation System for a Mobile Robotic Vehicle

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Engineering in
Electronic and Computer Systems Engineering
at
Victoria University of Wellington

By
Buddika Kasun Talwatta

2012

Abstract

One of the challenges of robotics is to develop a robot control system capable of obtaining intelligent, suitable responses to dynamic environments. The basic requirements for accomplishing this is a robot control architecture and a hardware platform that can adapt the software and hardware to the current state of the environment. This has led researchers to design control architectures composed of distributed, independent and asynchronous behaviours.

In line with this research, this thesis details the development of a control system which adopts a hierarchical hybrid navigation architecture designed at Victoria University of Wellington. The implementation of the control system is aimed towards one of Victoria University of Wellington's fleet of mobile robotic platforms called MARVIN. MARVIN is a differential drive robot and the sensory equipment on the device includes infrared sensors and odometry.

The control system has been implemented in C# .NET programming language adopting a Service-Oriented Architecture. This software framework provides several services along with a graphical user interface to configure the control system.

Several experiments have been carried out to test the control system and the results indicate that the features of the navigation architecture have been accomplished.

Acknowledgements

I would like to gratefully acknowledge the following people for the help and time they have given me over the course of this project.

Thanks to Professor Dale Carnegie, my project supervisor, for his invaluable advice, patience, expertise and direction in guiding me through the process of completing this project and during the write-up of this thesis. His assistance, especially during the review process of thesis chapters, has been immensely helpful.

A big thank you to Johnny Robert Keogh McClymont, a master's student, who introduced me to MARVIN during the initiation of the project and also for passing on his wisdom and guidance.

Thank you to Praneel Chand, a PhD student whose work has been the foundation of this project. He designed and simulated the hybrid navigation system. I greatly appreciate all the time spent explaining his research that has been invaluable in designing the control system of this project.

My gratitude also goes to the technical assistance provided by Tim Exley and Jason Edwards.

Lastly, I would like to thank my wife, Malsha Kularatna, my Mum and Dad, and my in-laws for being there when I needed the support and the encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

A mobile robot is a type of robot that has the capability of moving in its environment, so it is not confined to one specific location. An autonomous mobile robot is a system that can sense its environment, and achieve goals by acting on it. The main characteristic that defines an autonomous robot is the ability to act on the basis of its own decisions and not through the control of a human (Maja J. Mataric, 2007).

A common necessity for autonomous mobile robots is navigation and it refers to the way a robot finds its way in the environment (Maja J. Mataric, 2007). An intelligent and effective navigation system is responsible for allowing a robot to operate autonomously in dynamic and unpredictable environments; taking decisions based on on-the-spot sensed local data and/or previously acquired global knowledge of the environment. It primarily guides the robot to a desired destination or along a pre-specified path in which the environment consists of landmarks and obstacles. Robot navigation is a vast field and can be defined as the combination of three fundamental competences for better understanding, described briefly by the three following questions (Leonard & Durrant-Whyte, 1991):

- ***"Where am I?"***

This question constitutes the localization problem which can be stated as: A robot is at an unknown position in an environment for which it has a map. It "looks" about its position, and based on these observations it must infer the place (or set of places) in the map where it could be located (Guibas, Motwani, & Raghavan, 1997).

- ***"Where am I going?"***

This question is principally concerned with a robot being in an unknown environment which it does not have a map for, and by navigating through it enables the creation of a map representation. The robot must identify special features in environment landmarks in order to acknowledge where obstacles are located.

- ***"How do I get there?"***

This states a general concern with finding paths connecting different locations in an environment according to specific objectives and constraints.

To provide the robot with these capabilities, a robot control architecture is necessary. A robot control architecture is defined as a mapping of sensory information into actions in the real world, in

order to accomplish a certain task. There are three paradigms of robot control architectures, namely: deliberative, reactive and hybrid (combination of deliberative and reactive). These control architectures are analysed further in chapter 2.

1.1 Objectives

The broad object of this project focuses on implementing an intelligent control system for a mobile robot providing autonomous navigation capabilities around a given indoor environment. The system is implemented using off-the-shelf robotic control software platform and evaluated in the real world environments. At Victoria University, a hierarchical hybrid navigation system has been developed combining a deliberative motion plan and reactive obstacle behaviour (Praneel Chand & Carnegie, 2011). The hybrid navigation system has been simulated on multiple heterogeneous robots in known and unknown environments however, only limited real world experiments have been conducted (Praneel Chand & Carnegie, 2011). The control system will adapt this hybrid navigation system. While simplicity and robustness is a high priority, it is also highly important for this system to be flexible, avoiding binary decisions and hard limits in favour of parameters that can be adjusted and tuned to suit a wide variety of mobile robot platforms and environments.

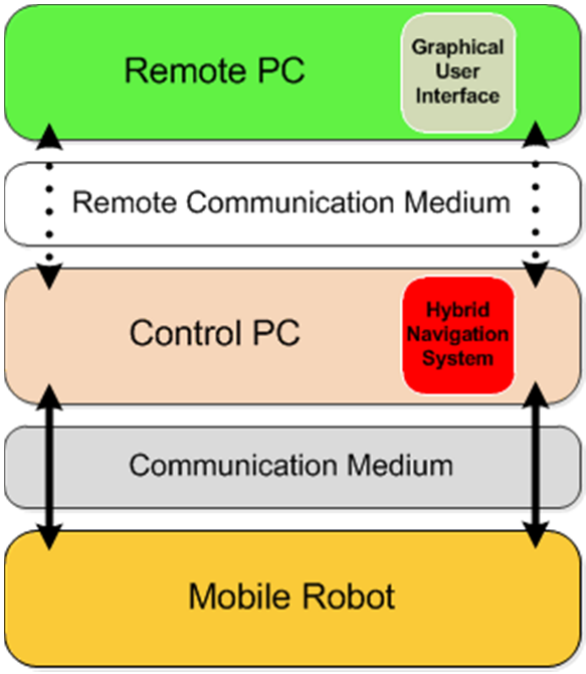

Figure 1.1 Overview of the system

A design philosophy had to be adopted prior to the development stage of the system to ensure a flexible, extendible and configurable modular software framework is produced, thus to minimize the complex adaptation across multiple mobile robot platforms.

Figure 1.1 shows the structure of the final system. The mobile robot is controlled by a control PC, where navigation algorithms are managed, generating controls that are transmitted back to the mobile robot utilizing a communication medium (such as USB, serial, Bluetooth) satisfied by both control PC and the mobile robot. This system is controlled further by the addition of a remote PC which runs diagnostic and performance measurement tools such as a graphical user interface (GUI) over a wireless communication medium (such as wireless LAN).

1.2 Mobile robot platform

Despite the design of the navigation system being targeted towards heterogeneous robot platforms, this thesis project will be implemented on a single robotic platform. Victoria University of

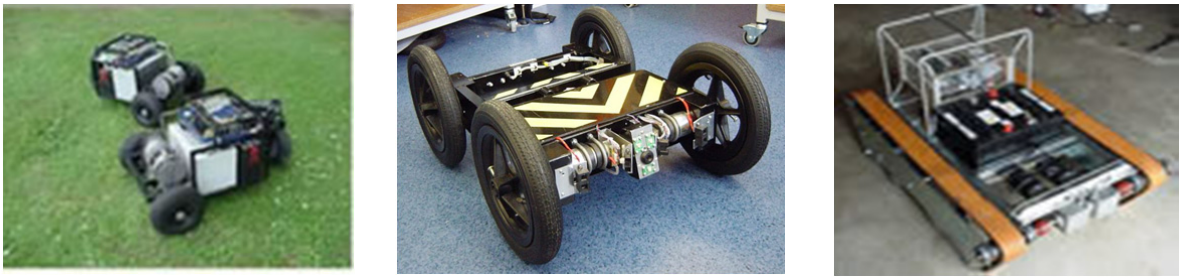

Figure 1.2 (Left) Itchy and Scratchy, (Centre) Rubblebot, (Right) Tank ("MARVIN," 2011)

Wellington's Mechatronic Group has several mobile robots which were developed from the accumulated work of post graduate projects. As depicted in Figure 1.2 these robots have various drive configurations and sensors for navigation tasks. These robots are constructed to traverse a variety of terrains including smooth indoor surfaces and rocky environments. However, at the onset of this project, none of these were in a working condition to test the navigation system and they required numerous hardware upgrades for autonomous navigation.

The Mechatronic Group also has a number of other robots including a differential drive robot called MARVIN (shown in Figure 1.3). MARVIN (backronymed to Mobile Autonomous Robotic Vehicle for Indoor Navigation) is the flagship of Victoria University's large-scale autonomous guided vehicles fleet. The original long-term objective of the MARVIN project was to develop an autonomous mobile security device that would patrol the corridors of the university however, in

recent years the project's focus has shifted towards human-machine interactions and public relations applications (Carnegie, Prakash, Chitty, & Guy, 2004).

MARVIN can alter his shape depending upon his "mood", becoming more aggressive and assertive, as allowing it to respond to humans in a more natural way ("MARVIN," 2011).

Figure 1.3 MARVIN with its upper torso attached

Over the past years, MARVIN has undergone major hardware revisions, the latest being the work of a master's student Johnny McClymont. Johnny has redesigned MARVIN's hardware adopting a design philosophy to minimise the need for future hardware modifications when the control computer is upgraded (McClymont & Carnegie, 2008).

1.3 Robot Software System

It is extremely difficult to develop larger programs like navigation systems due to the sheer size of the required code, as it must comprise a deep stack starting from low-level motion control and continuing up through perception, cognition, and beyond. Additionally, different types of robots can have considerably varying hardware, making code reuse unmanageable.

To meet these challenges, many robotics researchers have created a wide variety of frameworks, resulting in many robotic software systems to manage complexity and enable rapid prototyping of AI software (James Kramer & Matthias Scheutz, 2007). Some of the off-the-shelf robotic software

systems available, listed in the Table 1.1, were considered in the development of the hybrid navigation system.

Table 1.1 Summary of Robotic Software Systems

	Microsoft Robotics Developer Studio	**ROS**	**OROCOS**	**Player/Stage**
Open source	No	Yes	Yes	Yes
Free of charge	Yes	Yes	Yes	Yes
Windows	Yes	Yes (Partially)	No	Yes
Linux	No	Yes	Yes	Yes
Distributed environment	Yes	Yes	No	Yes (Limited)
Behavior coordination	Yes	Yes	No	No
Simulation environment	Yes	Yes	No	Yes
Range of supported hardware	Large	Large	Medium	Medium
Reusable service building blocks	Yes	Yes	Yes	No
Real-time	No	No	Yes	No

Each of these listed systems has its strengths and weaknesses but most importantly these systems were commercially available for free.

Microsoft Robotics Developer Studio (MRDS) was preferred due to the two following reasons:

- One of the requirements of the control system, in addition to receiving instructions from the navigation system, was to provide the ability to control the mobile platform remotely via a user interface. This interface would also expose localisation feedback. MRDS attempts to address this by providing a distributed system so that applications can be designed to interact over a network.

- The majority of the robotic platforms in the Mechatronic Group have a control computer running variants of the Microsoft Windows operating system (OS). It has been anticipated that future control computers will continue to run the Windows operating system and as

such the development of the hybrid navigation system has been targeted to run on Windows OS.

MRDS is implemented using .NET framework; therefore it is necessary to develop applications on MRDS using .NET languages. Implementations would be written in the primary programming language C#, though other languages such as C++, Visual Basic offer alternatives. Chapter 5 provides a more detailed discussion of MRDS and its components.

1.4 Operating Environment

MARVIN is intended to operate primarily in the corridors of the third floor of the Laby building at the Victoria University of Wellington. This restricted environment is to be used initially for debugging and testing the hybrid navigation system. Overhead view of the floor map is given in Figure 1.5 while the perspective of the robot is illustrated in Figure 1.4.

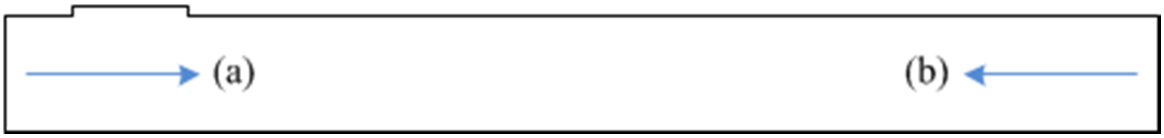

Figure 1.5 Overhead view of a corridor of the third floor of Laby building

(a) **(b)**

Figure 1.4 Perspective of the robot as indicated on the floor map

Once the tuning of the navigation system is completed, the system can be expanded to incorporate other indoor environments. The control system of MARVIN will be less reliant on location, and it should perform at full capacity inside any indoor environment with walls that are within the range of the infrared sensors. In an environment where the walls are spaced out and not detectable by the

sensors, the control system will still operate, however it must rely heavily on odometry for localisation, and therefore will become more susceptible to cumulative error.

1.5 Project Management

The project objectives require the following steps to be completed.

- Research the hierarchical hybrid navigation system.
- Develop firmware for the differential drive which includes a PID control system to maintain wheel velocities and obtain velocity profiles.
- Obtain obstacle distances, position and orientation information from rangefinder data.
- Obtain velocity, position and orientation information from odometer data.
- Implement the hybrid navigation system as part of a software architecture that is easily configurable and maintainable.
- For the hybrid navigation system, combine the localisation information from odometers and rangefinders to produce a single representation of the position and orientation of MARVIN.
- Test, debug and evaluate the developed navigation system.

1.6 Thesis Structure

The thesis is organized into eight chapters, the first being the current introductory chapter. The rest of the thesis is presented as follows:

Chapter 2 This chapter presents the different kinds of robot architectures proposed in the literature and also reviews several robotic development environments considered during the research process.

Chapter 3 This chapter covers the hardware aspects of the robot providing a brief overview of the current status of the robot. It also details the modifications made as a result of the requirements needed to achieve the objectives of this thesis.

Chapter 4 This chapter presents a background into the proposed navigation architecture. The process of obtaining localisation information from sensor data and combining data from multiple sensors in a useful manner is also discussed.

Chapter 5 This chapter details the development environment utilized to develop the software framework. It also provides an overview of the programming languages that are utilized to implement the architecture.

Chapter 6 This chapter covers the software implementation of the various features of the navigation architecture. It outlines the modular software framework developed to maintain flexibility and reusability.

Chapter 7 This chapter presents the results obtained during hardware testing and the results of the robot's motion under various conditions.

Chapter 8 This chapter concludes the thesis by summarizing the work and presenting conclusions as well as recommendations for future work.

Chapter 2 Background

2.1 Introduction

A definition of an intelligent mobile robot was considered to be was provided in chapter 1. Three paradigms are used to accomplish the task of designing an intelligent mobile robot, namely: deliberative, reactive and hybrid. This chapter begins by discussing the topics related to these paradigms followed by reviewing the robotic development environments (RDE) which aid designers to develop the control architectures. The overview of the architectures provides the background necessary to understand the choices and decisions made when selecting the architecture developed at Victoria University as given in chapter 4.

The deliberative architecture is the oldest and is based on traditional artificial intelligence (AI) (Nakhaeinia, Tang, Mohd Noor, & Motlagh, 2011). This kind of architecture is made up of multiple layers and each layer provides sub-goals to the layer below. The deliberative techniques use a global world model which is provided by user input or updated from the sensory information (through perception). Based on this world model, planning and reasoning are carried out that generate appropriate actions for the robot to reach the target. This means that the robot first senses its surroundings, then it plans an optimal path to follow and finally moves. Top-down approach in planning module is an important characteristic of this architecture where high level constraints are broken into low level commands (Nakhaeinia et al., 2011). However, in order to permit classic planners to work with, it requires an accurate model of the environment. To obtain a completely known map can be challenging and since the world model is application dependent it also becomes very difficult to build and maintain. In addition, reasoning becomes a bottleneck in the architecture as it takes a significant amount of time. Due to the complexity of the system, it requires enormous processing power and memory capabilities to perform the necessary calculations. Another drawback of this kind of architecture is the difficulty of functioning properly in the presence of uncertainty (due to actuator errors and sensor noise) in dynamic or real world.

In contrast to deliberative architectures, reactive and behaviour-based architectures (originally proposed by R. Brooks in 1986) are characterized by a close coupling between perception and action (Nakhaeinia et al., 2011). Unlike the deliberative approaches, behavioural architectures have a horizontal decomposition, in which behaviours work in parallel, concurrently and asynchronously. Behaviours provide a direct coupling between sensory inputs and the robot's actions. They are also inherently modular and support incremental layer expansion to build a navigation system. In reactive architectures, the control commands are generated based on the current sensory

information. The robot uses the local model of environment without planning process to take actions. Therefore, it is not necessary to build a complete model of the environment (Nakhaeinia et al., 2011). One disadvantage of reactive architectures is that the interaction between the environment and the system can be difficult and less predictable. Due to the lack of a planning module, when carrying out complex tasks, they do not perform well. Also, as behaviours are low level they do not reflect high level tasks.

Although reactive navigation architectures established a successful framework for mobile robot navigation, there are still some problems in regards with the complex unknown environment. Neither the purely deliberative architectures (sense-plan-act scheme) nor the purely reactive scheme (sense-act scheme) perform well when performing complex tasks, because of difficulties in modelling the world and relying too much on inadequate sensors (Orebäck & Christensen, 2003).

Thus, to perform an autonomously navigation in a real world environment some features of reactive architecture combines with the deliberative architecture resulting in hybrid architectures. These architectures can respond in real-time to changes in dynamic environments as well as plan actions ahead in time.

Hybrid architectures execute reactive behaviours independently of deliberative functions by utilizing asynchronous processing techniques such as multi-tasking and threads. For example, while a robot is reactively navigating towards its current goal, a planner running as an independent process can slowly compute the next goal. Additionally, good software modularity allows subsystems or objects in hybrid architectures to be mixed and matched for specific applications (Murphy, 2000). For controlling a single robot there are many hybrid architectures. As mentioned in chapter 1, a hybrid architecture developed at Victoria University is chosen for this study.

Figure 2.1 summarizes the three paradigms used to develop robot control architecture. As can be

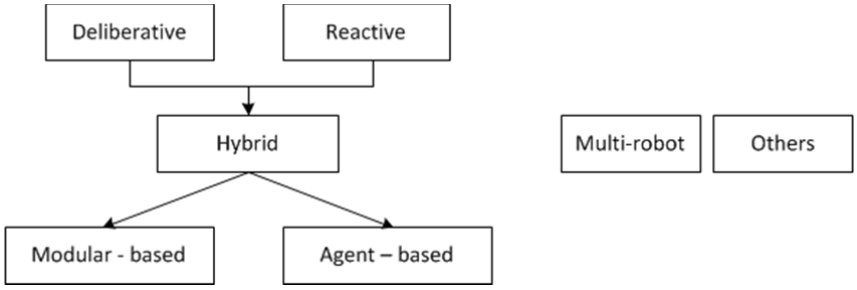

Figure 2.1 Robot control architecture

seen, the hybrid paradigm has the advantages of both deliberative and reactive schemas. It can be

achieved using an agent-based or a modular-based approach. Although not discussed in the context of this thesis, multi-robot and other systems are also included in the figure to complete the overview of the various schemas.

2.2 Deliberative Control Architectures

These types of architectures are realized in a top-down fashion and are structured in layers. Deliberative control uses a simple sequence of three steps as shown in Figure 2.2. First the robot senses the environment, constructs a global world map and gathers the necessary data for calculating possible action outcomes. Then, it plans by searching throughout possible outcomes and choosing the directive which best achieves a goal. Finally, it acts to carry out the first directive (Murphy, 2000). A top-down approach is taken to perform each step at the corresponding layer.

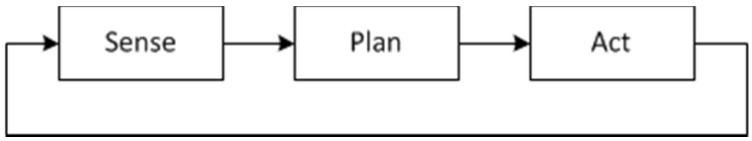

Figure 2.2 Hierarchical architecture sequence

Deliberative reasoning systems have the following characteristics:

- Structure of the system is hierarchical with functionality divided into clearly identifiable subsections.
- Top levels in the hierarchical system produce sub-goals for lower levels.
- With little if any lateral movement, communication and control flow top-down the hierarchy in a predictable and predetermined manner.
- Deliberative planning is integrated at all levels of the entire hierarchical architecture, with different spatial and temporal scales at each level. At the lower levels, time requirements are shorter and spatial considerations are more local.
- Dependency on explicit representation of the world models.
- Communication order of the system is structured.

A relatively accurate model of the environment is required to predict the outcome of the robot's actions. This enables the robot to optimize its performance relative to the model. Deliberative reasoning often requires strong assumptions about the world model. Accordingly, the knowledge upon which this reasoning is based upon has to be consistent and reliable. The outcome of the reasoning may error, if the information model is inaccurate or has changed since it was obtained.

Thus, hierarchical control is not suitable for dynamic environments which require timely responses however, is seemingly well appropriate for highly predictable and structured environments. Due to the slow response to changes in the real world dynamic environments this project does not rely solely on deliberative architecture.

One of the most representative architecture in this paradigm is RAP: Reactive Action Packages (Firby, 1990). As shown in Figure 2.3, this three-layered architecture is formed by planning, execution and control systems (Firby, 1990).

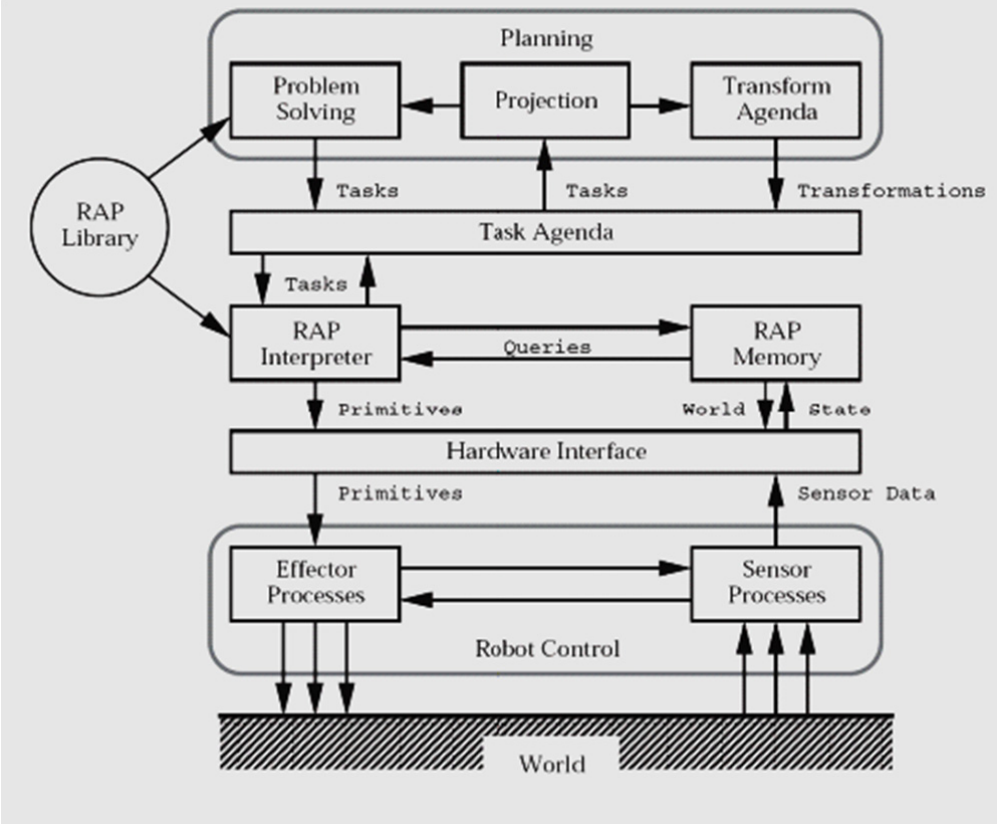

Figure 2.3 Layout of RAP architecture (extracted from (Firby, 1990)).

In order to achieve the navigation goal, the planner splits the mission into multiple high-level sequences of tasks for execution. The execution layer expands each of these tasks into more detail. And finally, the control system manipulates the robot's sensors and actuators by translating the tasks into low level commands.

Each task consists of a list of methods and a set of conditions that must be satisfied to apply the methods. A method is installed in the task queue as a primitive action or a list of sub-tasks. The system consecutively expands tasks in the queue until they either successfully complete or fail. An alternate method is tried when a failure occurs. Consequently, a success check is maintained and

failing to achieve the desired state, the RAP system repeats the process with another method. Eventually, the RAP system will give up when all possible methods have been tried several times. Another feature of the RAP system is that it can deal with opportunities (situations in the world that cannot be predicted by the planning system) and emergencies (new facts that appear and that the system must respond to immediately, dropping the current task) (Innocenti, 2008). This is achieved by allowing reaction tasks to exist on the execution agenda concurrently with tasks making up a plan, and assigning them priorities that allow the system to shift from one goal to another when appropriate (Innocenti, 2008). In this way some receptiveness to changing environment conditions is achieved.

2.3 Reactive Control Architectures

Reactive control architecture is an alternative to the deliberative scheme. Figure 2.4 illustrates the sequence used to control robots with this architecture.

Reactive systems often have several common characteristics:

- The basic building blocks for robotic actions are behaviours. In reactive systems, a behaviour typically consists of a simple sensor-motor (sense-act) relationship. Each behaviour can receive sensory information for a given task resulting in a particular low-level motor response (stimulus-response).
- As the world is sensed, purely reactive systems react directly to it, avoiding the use of explicit abstract representational knowledge in the generation of a response. In other words the planning process or internal state information is not required.
- From a software design perspective, these systems are inherently modular and concurrent.
- There is less computation and shorter delays between perception and action hence reactive systems are very fast.
- Lack of a planning module means purely reactive architectures are unable to learn.

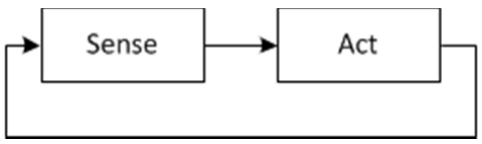

Figure 2.4 Reactive architecture sequence

Several researchers (M.J. Mataric, 2009; Pirjanian, 1997) agree in considering a behaviour-based paradigm as different from a reactive paradigm, and some would also add free market and immunity based architectures to the previous list (Singh & Thayer, 2001).

Using reactive systems is most beneficial when the real world cannot be accurately modelled or defined. Often it can be very difficult to filter noise, uncertainty and unpredictability from these models. Therefore, relying heavily on sensing, reactive architectures are developed without constructing global world models prone to potential errors. This project does not adopt a reactive architecture due to its reduced predictive capabilities and its requirement on explicit world representations.

Subsumption architecture is one of the most representative architectures in this paradigm (Brooks, 1986) (as given in Figure 2.5). This architecture is a composition of several levels of competence that are informal descriptions of how the robot should behave for any environment it will encounter (Brooks, 1986). The levels are arranged by their class of behaviours, with enhancing each level of competence supplied by the level before it to provide a higher competence.

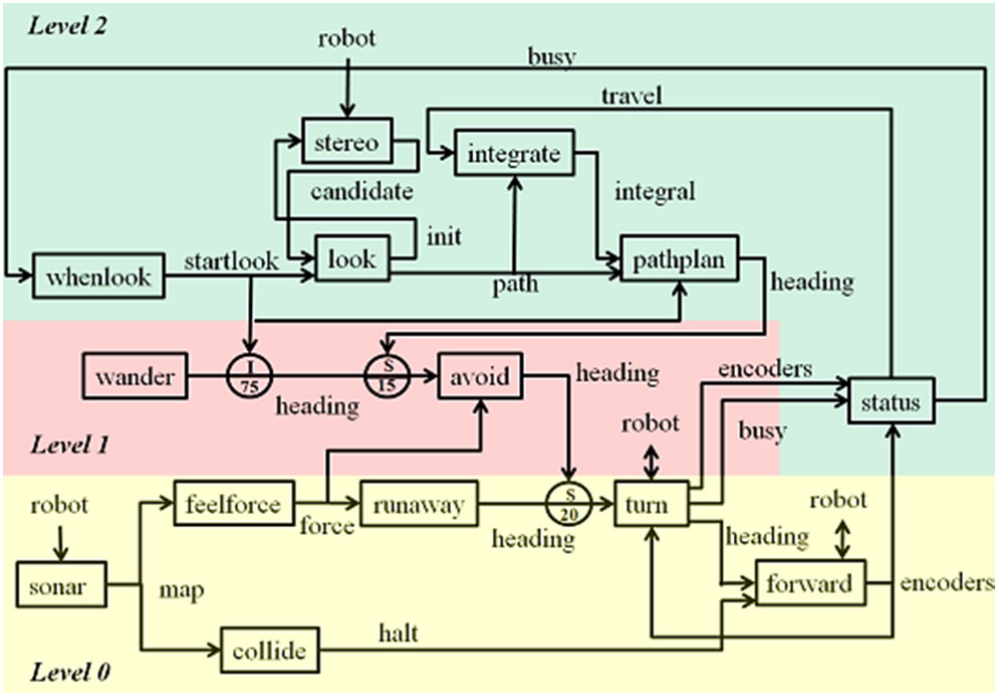

Figure 2.5 Layout of Subsumption architecture (extracted from (Brooks, 1986))

Each level of competence is implemented incrementally by adding a corresponding layer of control to the existing set of levels of competence so that the next highest level of overall competence can be achieved. In a successful implementation of layers of control systems, a given lower layer

remains unaware of its higher level, except for the occasional intervention by the higher level to make the refinements to its behaviour necessary for a higher level of competence.

Layers are implemented from fixed topology networks of augmented finite state machines (small asynchronous processors) that transmit messages to each other. The communication among these finite state machines is established via connecting "wires" (as inputs and outputs of the processors). These processors (squares) along with the inputs and outputs are shown in Figure 2.5. The wires, since they carry dataflow, are the elements that higher layers interact with to enhance lower layers (Fitzpatrick, 1997). To examine data flow in a lower layer, a higher layer has the ability to attach an extra wire to whatever it wishes to monitor. There are a number of schemes for overriding normal data flow. An extra wire can terminate at the target of another wire (denoted by circles with a capital letter I in Figure 2.5). If any message flows in the extra wire, it inhibits any message or signal along the other wire. An extra wire can also terminate at the source of another wire (denoted by circles with a capital letter S in Figure 2.5). Any message flowing in the extra wire will not only inhibit any message or signal along the other wire, but will be inserted on to it, suppressing and replacing the normal dataflow. This interrupts the dataflow along the augmented finite state machine and thus do not reach the actuators. Suppression and inhibition and are the two mechanisms used to coordinate behaviours.

2.4 Hybrid Control Architectures

Although reactive navigation architectures established a successful framework for mobile robot navigation, there are still some problems in regards to the complex unknown environment (Nakhaeinia et al., 2011). Hybrid control architectures combine the benefits of both worlds: the use of high-level planning and abstract representational knowledge of deliberate control and the robustness, flexibility and responsiveness of purely reactive systems. A hierarchical hybrid navigation system can achieve the benefits of both deliberative and reactive control (P. Chand, 2011).

Hybrid architectures permit reconfiguration of reactive control systems based on available world knowledge through their ability to reason about underlying behavioural components (Innocenti, 2008). Figure 2.6 depicts the hybrid architecture sequence used to control robots.

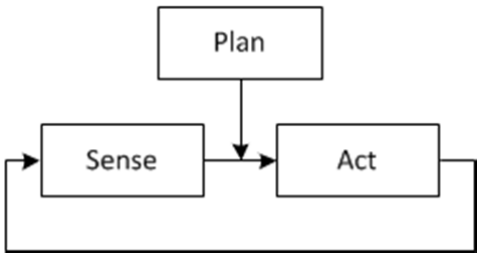

Figure 2.6 Hybrid architecture sequence

Hybrid approach implies that, an autonomous robot first plans how to employ a global world model to accomplish a mission or a task, followed by activating a set of low level behaviours to satisfy the plan that is executed until it is completed. Then the planner continues to repeat the process after selecting another set of behaviours.

The common components of hybrid architectures are (Murphy, 2000):

- Sequencer - this generates the set of behaviours to use in order to accomplish a subtask, and determines any sequences and activation conditions.
- Resource Manager - this allocates resources to behaviours.
- Cartographer - this is responsible for creating, storing and maintaining a map or special information, plus methods for accessing the data. It often contains a global world model and knowledge representation, even if it is not a map
- Mission Planner - this interacts with humans, operationalizes the commander in robot terms, and constructs a mission plan.
- Performance Monitor and Problem Solving - this allows the robot to be aware of whether or not it is making progress.

Hybrid architectures can be loosely divided into three categories (Murphy, 2000):

- Managerial styles: these focus on decomposing the deliberative portion into layers based on responsibilities, or scope of control of each deliberative function. Mission planning module, which is more abstract than the path planning module, is responsible for performing high level planning; directing subordinate deliberative modules such as navigation, which refine it and gather resources; and then passing down to the modules in the lowest levels, which behave reactively. Some examples of architectures belonging to this style include AuRA (Arkin, 1987), Socio-Intentional (Kolp, Giorgini, & Mylopoulos, 2006), DAMN (Rosenblatt, 1997).

- State hierarchies: these use the knowledge of the robot's state to distinguish between reactive and deliberative activities. Reactive behaviours are viewed as having no state, no self-awareness, and function only in the present. Deliberative functions can be divided into those that require knowledge about the robot's past state (where it is in a sequence of commands) and about the future (mission and path planning). Some examples of architectures based on this style are 3-Tiered (3T) (Bonasso et al., 1997), ATLANTIS (Gat, 1991).
- Model-oriented styles: unlike the managerial or state-hierarchies, these architectures have a more top-down, symbolic flavour. One hallmark of these architectures is that they concentrate symbolic manipulation around a global world model. However, unlike most other hybrid architectures, which create a global world model in parallel with behaviour specific sensing, this global world model also serves to supply perception to the behaviours. In this case, the global world model serves as a virtual sensor. Architectures developed based on this style are Saphira (Konolige, Myers, Ruspini, & Saffiotti, 1997) and Task Control Architecture (TCA) (Simmons, 1994).

In the following sections, several early adaptations of the hybrid architectures are described.

2.4.1 Autonomous Robot Architecture (AuRA)

AuRA (Arkin, 1987) is one of the first implemented hybrid architectures and it is based on the schema theory. It consists of the following five major subsystems (as shown in Figure 2.7) (Murphy, 2000):

- A planning subsystem – responsible for mission and task planning. It is subdivided into three components: the mission planner, the navigator and the pilot. The mission planner serves as the human interface; the navigator computes a path to achieve the goal and breaks it into sub-tasks. Finally, the pilot takes the subtasks and gets relevant information to generate behaviours.
- A cartographic subsystem – encapsulates all the map making and reading functions needed for navigation. The three components of the planner would interact with the cartographer through methods to obtain a path to follow, broken down into sub-segments.
- A homeostatic control subsystem that modifies the relationship between behaviours by changing their gains as a function of the "health" of the robot or other constraints

- A perceptual (sensor) subsystem that gets sensory information through perceptual schemas. This subsystem extracts information from the environment, structures it in a coherent and consistent manner and delivers it to the cartographer and motor schema manager.
- A motor subsystem that contains the motor schemas that form the behaviour schemas. This subsystem allows the robot to interact with the environment in response to sensory stimuli and plans. It contains the motor schema manager that is responsible for controlling and monitoring the behavioural process at run-time.

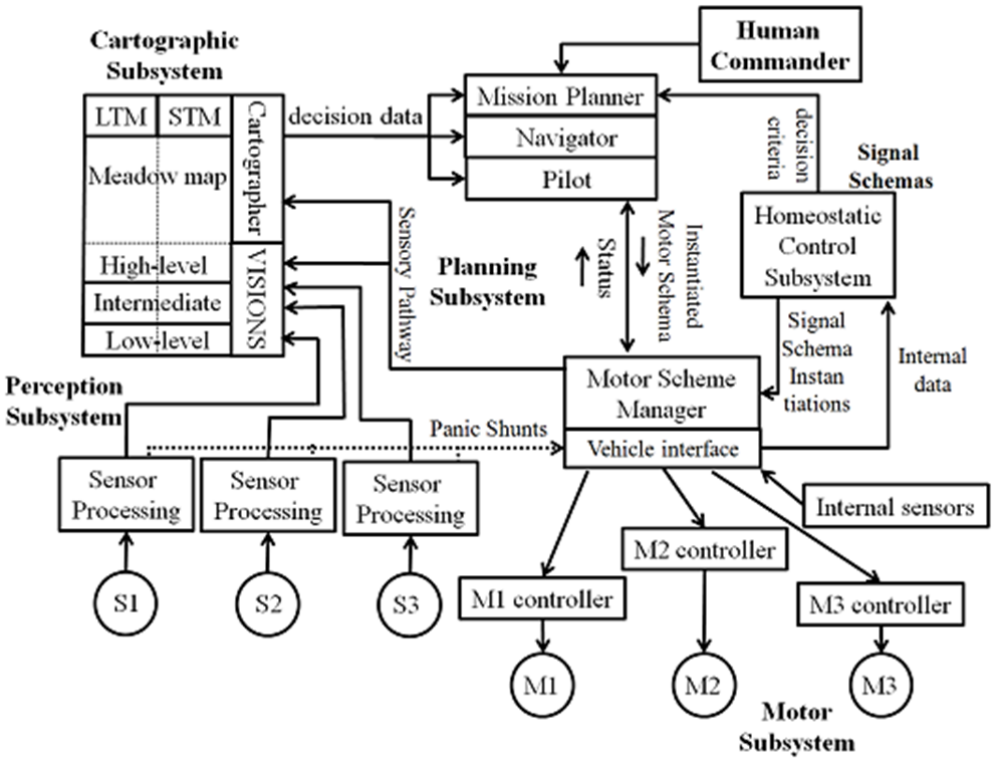

Figure 2.7 Overview of the AuRA architecture (extracted from (Arkin, 1987))

In AuRA, planner and cartographer subsystems form the deliberative portion, while the sensor and motor subsystems make up the reactive portion. Homeostatic control subsystem falls into a grey area between reactive and deliberation.

2.4.2 A three-layer Architecture for Navigating Through Intricate Situations (ATLANTIS)

ATLANTIS (Gat, 1991) is a combination of a classical planning system with a reactive control substrate. It can undertake multiple tasks in real time in partially unpredictable and noisy environments. Planning is seen as an attempt to transform one world state to another using "operators", which map on to associated physical actions when executed (Fitzpatrick, 1997). These

operators are executed in an atomic fashion, so there is a strict one-to-one correspondence between operators and actions (Fitzpatrick, 1997). Each of the behaviours in the Subsumption controller is associated with some "goal" and can report on its progress toward achievement of this objective. The problem with this type of system is that detecting the signalling as a result of goal completion at such low level can be difficult (Connell, 1992). The advantage of this architecture is that the low-level reactive layer and higher-level layers are asynchronous. This allows the controller to deal with random events in the environment while planning takes place in the deliberative layer (Jones, 2008).

ATLANTIS consists of three main components (as shown in Figure 2.8) (Fitzpatrick, 1997):

- The controller – Manages collections of primitive activities that are mostly reactive.
- The sequencer – Concerned with controlling sequences of both physical activities and deliberation. It manages the activation of modules in the controller, monitors them, and provides them with suitable parameters for their operation.
- The deliberator – In charge of performing computationally expensive, long term tasks such as planning and world modelling.

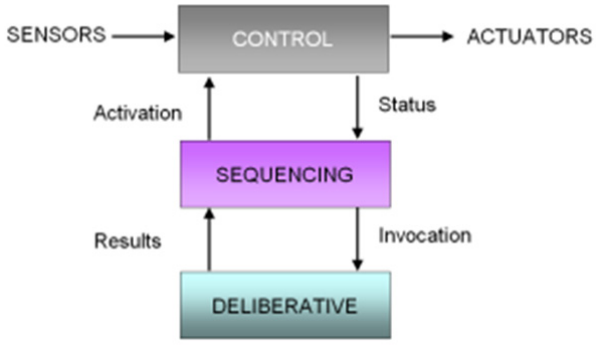

Figure 2.8 Overview of ATLANTIS architecture (extracted from (Arkin, 1998))

2.4.3 Saphira

The Saphira (Konolige et al., 1997) architecture is made up of two layers; deliberative and reactive (see Figure 2.9). The deliberative layer consists of the planning agent, some deliberation activities divided into several different software agents, the topological planner agent, the local perceptual space agent, the navigation tasks agent and the localization and map maintenance agent (Innocenti, 2008). The reactive layer comprises of the reactive behaviours and the fuzzy module that fuses the competing demands from the behaviours (i.e. coordinates the behaviours).

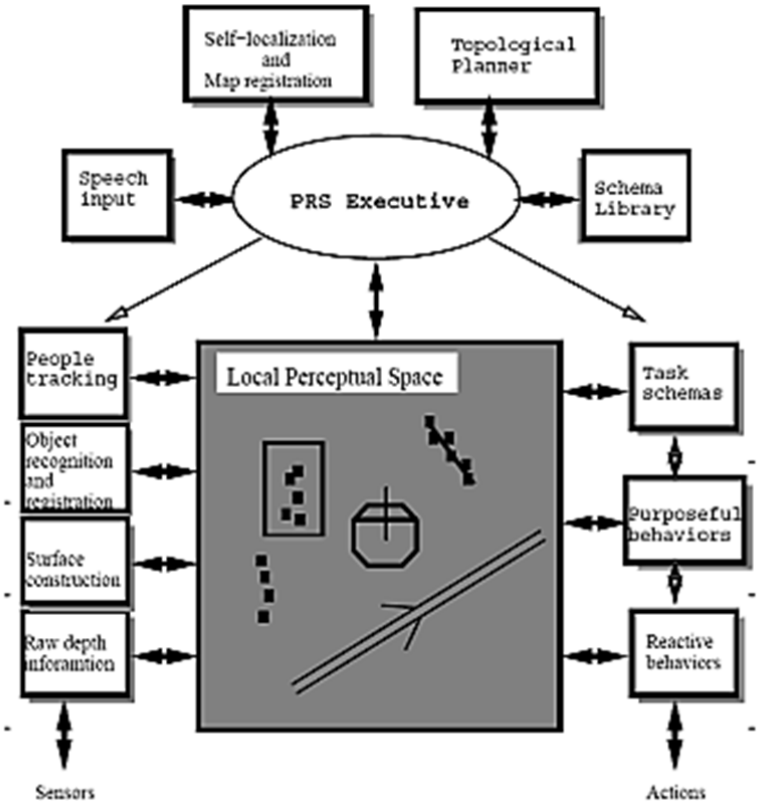

Figure 2.9 Layout of the Saphira architecture (extracted from (Konolige, Myers, Ruspini, & Saffiotti, 1997))

2.4.4 Dynamic window approach

The dynamic window approach (Fox, Burgard, & Thrun, 1997) is a reactive avoidance technique to avoid collisions with obstacles. It is a velocity-based local planner that calculates the optimal collision-free velocity for a robot required to reach its goal. Hence the dynamic window approach is derived directly from the dynamics of the robot. It translates a Cartesian goal into translational and rotational velocities. Two main components of the system are: generating a valid space to search, and choosing an optimal solution in the search space which can be reached within a short time interval.

This architecture has been adapted, combined with deliberative techniques and merged with other reactive techniques to produce a range of navigation architectures.

2.4.5 Integration of A* path planning algorithm and dynamic window obstacle avoidance approach

In this approach, the previously discussed dynamic window architecture is combined with the A* path planner (Macek, PetroviC, & Ivanjko, 2003). The velocity space based search advantages of the dynamic window are integrated with the local minimum free search characteristics of the A* algorithm (Macek et al., 2003). At every control cycle, the incremental build-up of the occupancy map and A* search algorithm are performed. In unknown environments, this gradual build-up provides the planner more detailed information of the environment. In the literature, this method is verified in a simulated environment using a circular differential drive robot.

A similar architecture has been developed by Lee-Johnson that employs the dynamic window method and an A* path planner (Lee-Johnson & Carnegie, 2006). To accomplish path-planning, it utilizes a pre-generated map with fixed binary occupancy grid data. Hence, this system does not have map updating capabilities. A major draw-back of this rudimentary system is the lack of support for heterogeneous robots as this is only designed for differential drive robots with circular shapes.

2.5 Robotic Development Environments

Robotic development environments (RDE) have been developed to facilitate research in autonomous robotics and various aspects of the agent development process, ranging from the design of an agent architecture, to its implementation on robot hardware, to executing it on the robot (J. Kramer & M. Scheutz, 2007). These environments support the development of reusable, portable, flexible, modular and extendable software application infrastructure that can support heterogeneous robotic platforms.

This section analyses a list of RDEs and several mobile robot architectures that can be considered as RDEs. These architectures have been used either to implement a real robot control, as in (Botti, Carrascosa, Julián, & Soler, 1999) or to model some existing architecture as in (Muscettola, Dorais, Fry, Levinson, & Plaunt, 2002), in a modular/multi-agent way. Moreover, there are several wide-ranging projects, OROCOS (Bruyninckx, 2001) and ROS (Quigley et al., 2009), whose aim is to become general-purpose robot control software packages.

2.5.1 Open Control Robot Software (OROCOS)

OROCOS (Bruyninckx, 2001) architecture is meant to develop a general-purpose, free software, modular framework for robot and machine control. Presently, OROCOS project supports four C++ libraries: Orocos Component Library, the Bayesian Filtering Library, the Real-Time Toolkit, and the Kinematics and Dynamics Library.

The Orocos Component Library provides off-the-shelf control components.

The Bayesian Filtering Library provides a framework for recursive information processing algorithm and estimation algorithm based on Bayes' rule (Innocenti, 2008).

The Real-Time Toolkit provides an environment for application designers to build highly configurable and interactive component-based real-time control applications (Innocenti, 2008).

The Kinematics and Dynamics Library provides a framework for modelling and the computation of kinematic chains (Innocenti, 2008).

A weakness in the OROCOS architecture is the lack of support for common hardware and the level of complexity in setting up the development environment.

2.5.2 Robot Operating System (ROS)

ROS (Quigley et al., 2009) is a software framework for robot software development, providing a structured communications layer above the host operating systems of a heterogeneous compute cluster. The primary goal of the ROS project is to support code reuse in robotics research and development. It provides standard operating system services, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management (Willow Garage, 2012). It also provides tools and libraries for obtaining, building, writing and running code across multiple computer environments (Willow Garage, 2012). It currently supports three different languages: C++, Python, and Lisp.

An application developed using ROS consists of a peer-to-peer network of processes, potentially on a number of different hosts, loosely coupled using the ROS communication infrastructure. The fundamental concepts of the ROS implementation are nodes, messages, topics and services.

Node – A node is a process that performs computation. At a fine-grained scale, a system designed with ROS typically comprises multiple nodes. Nodes enable software developers to modularize ROS applications (Willow Garage, 2012).

Messages – The nodes use messages, which are strictly typed data structures, to communicate with each other.

Topic – A node sends a message by publishing it to a given topic which is simply a string such as "odometry" or "map". A node that is interested in a certain kind of data will subscribe to the appropriate topic (Quigley et al., 2009).

Service – The publish-subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for synchronous interactions, which are often required in a distributed system (Quigley et al., 2009). Therefore, services are utilized to simplify this broadcasting routing scheme.

Despite of ROS being a feature rich software stack, it is still not a mature product and therefore is not supported by all the operating systems.

2.5.3 Player/Stage

The Player/Stage (B. Gerkey, Vaughan, & Howard, 2003, 2005; B. P. Gerkey et al., 2001) project is designed to be a programming interface, specifically avoiding being a development environment (J. Kramer & M. Scheutz, 2007). Unlike other IDEs, it focuses on the devices as the primary units of agency, i.e., the sensors and actuators that can be in a robot. Collection of these devices does not necessarily have to be located in the same robot. It includes features such as obstacle avoidance, vector field histogram goal-seeking, a wave front propagation path planner, and adaptive Monte-Carlo localization.

Player refers specifically to the device and server interface while Stage is the graphical 2-dimensional device simulator. Devices register with a Player server to become accessible to clients (J. Kramer & M. Scheutz, 2007). Clients that use these devices communicate with the server using socket connections allowing clients to be programmed using any language with socket support.

Unlike ROS framework, Player/Stage is not regularly maintained and hence does not support most of the robot hardware.

2.5.4 Miro - Middleware for Robots

Miro (University of Ulm Robotics Group, 2005; Utz, Sablatnog, Enderle, & Kraetzschmar, 2002) is a distributed object oriented framework for mobile robot control. The Miro core components have been developed in C++ under the aid of ACE (Adaptive Communications Environment), a

communication framework based on CORBA (Common Object Request Broker Architecture) technology. Currently, Miro can only be compiled on a Linux platform.

Miro abstracts the hardware devices of robots as active services exporting the sensors and actuators functionality via the communication framework that can be accessed transparently from other programs probably running on totally different machines ("Miro Manual," 2009). These abstraction interfaces include range sensor (infrared, sonar, laser, and bumper), motion, odometry, video, stall, GPS, pan-tilt, speech, and GUI buttons ("Miro Manual," 2009).

2.6 Summary

This chapter has reviewed the literature relating to robot architectures and robot development environments. The representative examples of work from both of these areas have been presented however are too large for exhaustive coverage.

The architectures have been classified according to the paradigm they represent: deliberative, reactive -based or hybrid. The review of the robot architectures will serve as background to chapter 4 in which an existing hybrid architecture is chosen to be implemented and tested on a real robot. The hybrid architecture this thesis focuses on has been designed to work on memory constrained heterogeneous robots (Praneel Chand & Carnegie, 2011). This enables the architecture to be adapted for any type of robot with varying size, shape, drive type and sensor quantity for future research.

In addition, a review of the robotic development environments is also included since some of them are required to develop or implement the robot control architecture. The review of the development environments provides context for chapter 5, in which a suitable development environment is discussed further. The solutions reviewed in this chapter were not suitable as the weaknesses outweighed the benefits of using them. Hence, chapter 5 gives an overview of another software platform that would enable the development of robust and highly distributed applications.

Chapter 3 Hardware

3.1 Overview

As mentioned in earlier in the introduction, MARVIN is selected as the mechatron to target the development and testing of the navigation system. This chapter describes the physical characteristics and the hardware components of MARVIN.

MARVIN's hardware can be divided into two main components: an upper body and a differential drive base. The primary function of the upper body is to improve the robot's aesthetic appeal and facilitate human-machine interaction. The base contains sensors, actuators and electronics required for navigation and high-level computation. Since this thesis focuses on the application of navigation to mobile robot control rather than human-machine interaction, the upper body is of less relevance. Figure 3.1 displays the base and the upper torso of MARVIN.

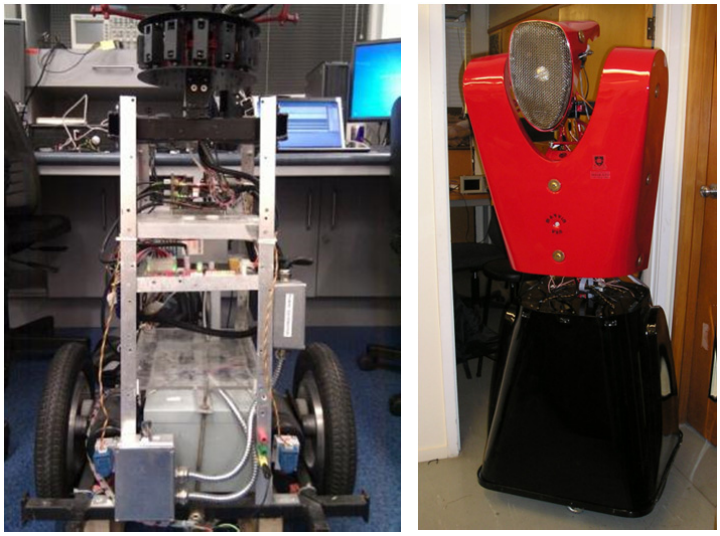

Figure 3.1 (Left) Differential drive of MARVIN, (Right) MARVIN with the upper torso

Before the navigation algorithm could be fully tested on MARVIN, certain hardware issues had to be addressed. In its initial state the drive base was non-operational and required a mini overhaul to get it up to a standard where testing in the initial environment could occur. At the onset of the project the mechatron was not equipped with a control PC therefore a suitable PC had to be chosen to perform the required control, mapping and planning. The details of these changes will be further discussed in this chapter.

3.2 Mechanical Details

MARVIN's frame is 0.777 m high, 0.583 m wide and 0.515 m long. The base of the frame is constructed from 25 mm × 25 mm steel tubing, welded together for strength. The upper section consists of aluminium struts that are riveted or screwed together for easy modifiability.

In order to lower the centre of gravity and improve stability, the heaviest components, the motors and batteries, are mounted at MARVIN's base. Three Perspex platforms (as shown in Figure 3.2) are attached above the drive base, providing non-conductive surfaces on which to mount the various electronics. For accessibility reasons, the platform above the drive base accommodates the PC. The ATX power supply, supplying power to the on-board controllers and the sensors, is housed on the second platform above the PC. The controllers are mounted on the upper platform.

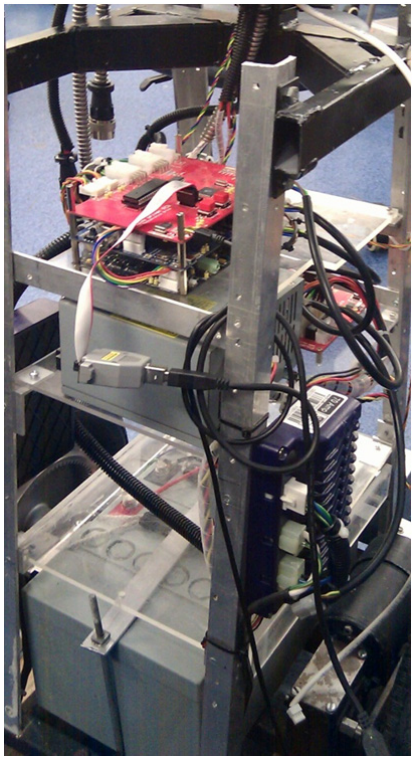

Figure 3.2 Hardware mounted on the three Perspex platforms

Locomotion is provided by two wheels in the standard wheelchair configuration located on the centre axis allowing them to rotate at different speeds. The wheels have a circumference of 1.037 m, and the distance between the centres of each wheel is 0.508 m. To provide stability, castor wheels are utilized at the front and the rear. Power to the wheels is provided by two independent motors. MARVIN's configuration allows it to turn through 360° while staying in the same position by driving the wheels in opposite directions. Turning can also be implemented by driving one wheel

faster than the other causing the robot to turn in an arc. This configuration also enables the wheels to support the majority of the device's weight.

MARVIN's linear velocity and heading are controlled by varying the angular velocity of each wheel. This yields tighter turning circles and reduced wheel slippage in comparison to other possible arrangements such as tricycle or quad wheel configurations (Loughnane, 2001).

The wheels are driven independently by two 24 V DC permanent magnet brush scooter motors, with a worm gearbox mounted to the face of the motor. The gear reduction ratio from the motor to the wheel is 51.56:1. This means that for every revolution of the driving wheel, the motor must turn 51.56 revolutions (Loughnane, 2001).

Flooded Lead Acid (FLA) batteries are used to power the motors and other electronics. Two of these batteries have been connected in series to provide 24 V. Each battery has a Cold Cranking Amperage (CCA) of 310 A and a Reserve Capacity (CA) of 55 minutes (Loughnane, 2001). The maximum current that can be drawn from this arrangement is 10 A with the batteries being able to be run at full power for 110 minutes before needing recharging (Loughnane, 2001).

No mechanical alternations were made to MARVIN.

3.3 Hardware

MARVIN is controlled by a distributed network of control modules. Each of these modules controls a specific function of MARVIN and interfaces with a control computer to receive and process commands via a virtual COM port over USB technology.

The control modules by MARVIN were designed and implemented by master's student Johnny McClymont to simplify future upgrades with the added benefit of removing the dependence on proprietary hardware (McClymont, 2011).

All control operations are implemented on a dedicated microcontroller on each module thus reducing the burden on the control computer's resources as it no longer needs to constantly control and monitor the peripheral hardware. Primarily the Atmega128 microcontroller from the Atmel 8-bit AVR family of microcontrollers is employed and a set of unique commands is implemented for each module. Figure 3.3 illustrates the main control modules and how each of these modules interacts with other aspects of MARVIN. The figure shows that the control computer communicates with each of the hardware modules over USB communication. Each of the rangefinder sensors communicates with the sensor network interface via a RS-485 link. Power to the microcontroller boards is provided by the ACE-828 power supply unit supplying 5 V.

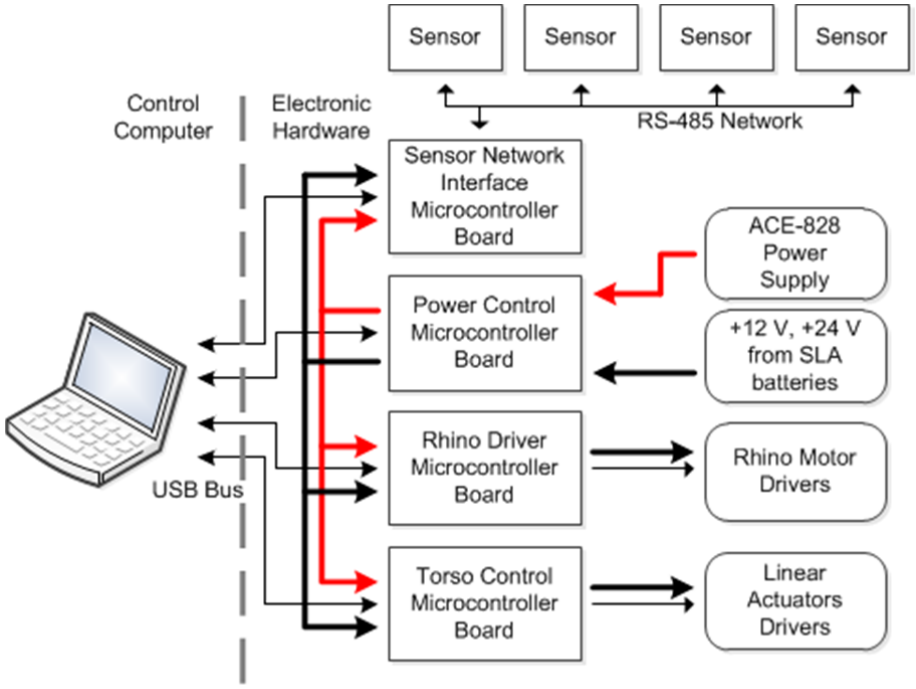

Figure 3.3 Hardware control module layout of MARVIN (McClymont & Carnegie, 2008)

3.4 Rhino Driver Module

Figure 3.4 shows the Rhino driver module which controls MARVIN's differential drive.

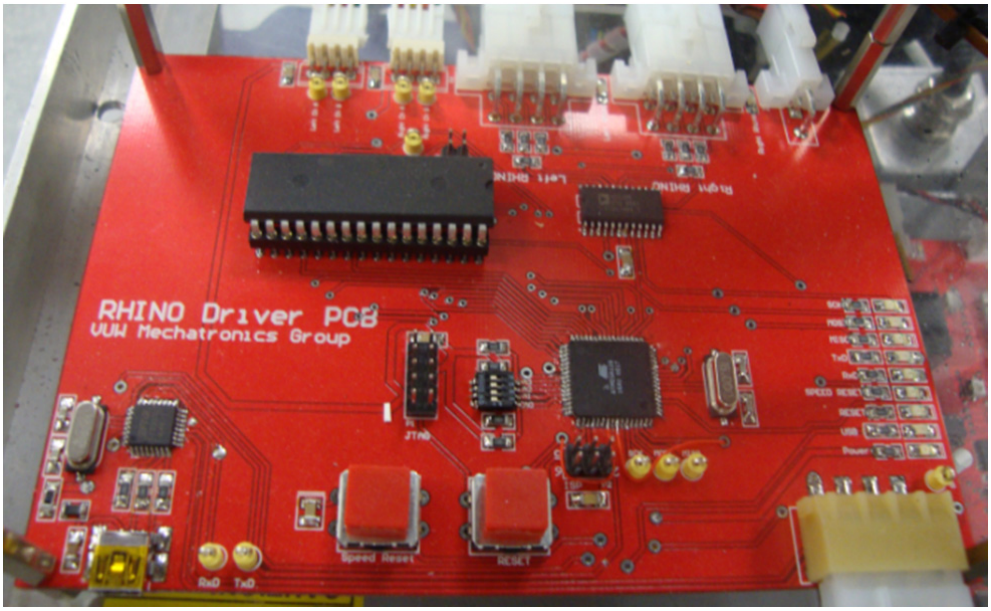

Figure 3.4 Rhino driver module

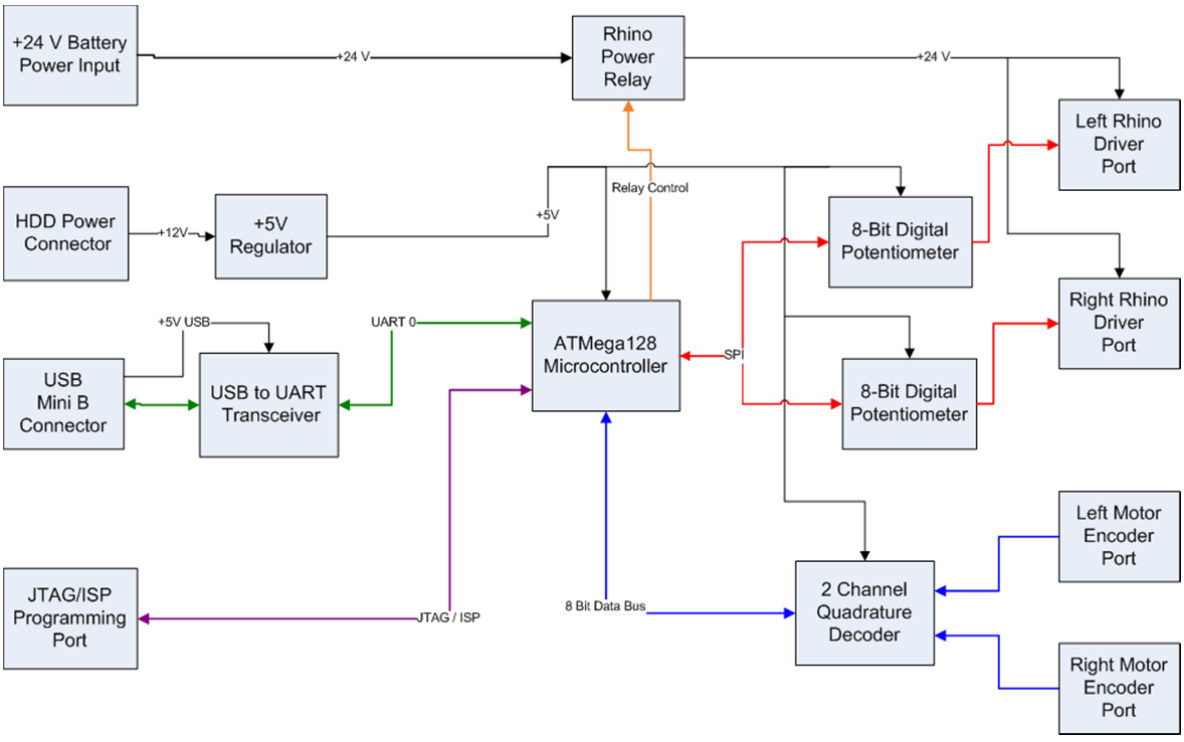

Figure 3.5 Block diagram of Rhino driver module

Each of MARVIN's motors is controlled by a Rhino DS72K H-bridge motor driver from Dynamic Controls Ltd and is mounted with a HEDS-5701 quadrature optical encoder to provide feedback to control the speed and distance precisely. Rhino driver module consists of an 8 bit digital potentiometer per H-bridge driver. The microcontroller of the module sends speed data to the H-bridge drivers serially via the digital potentiometers to set the speed of the motors.

A block diagram of the Rhino driver control module is depicted in Figure 3.5. The microcontroller can be seen at the centre controlling the digital potentiometer and receiving encoder feedback from a quadrature decoder. This decoder decodes both left and right encoders. The power regulation and the USB interface can be found to the left of the microcontroller.

3.5 Sensors

3.5.1 Sensor Network

The sensor network is implemented to provide a 360° field of view utilising a mix of 14 sensor nodes; four Sharp short range GP2Y3A002K0F and ten medium range GP2Y3A003K0F IR sensors. The sensors are mounted at the top of MARVIN's base section (as illustrated in Figure 3.6), approximately 1 m above the ground (McClymont, 2011).

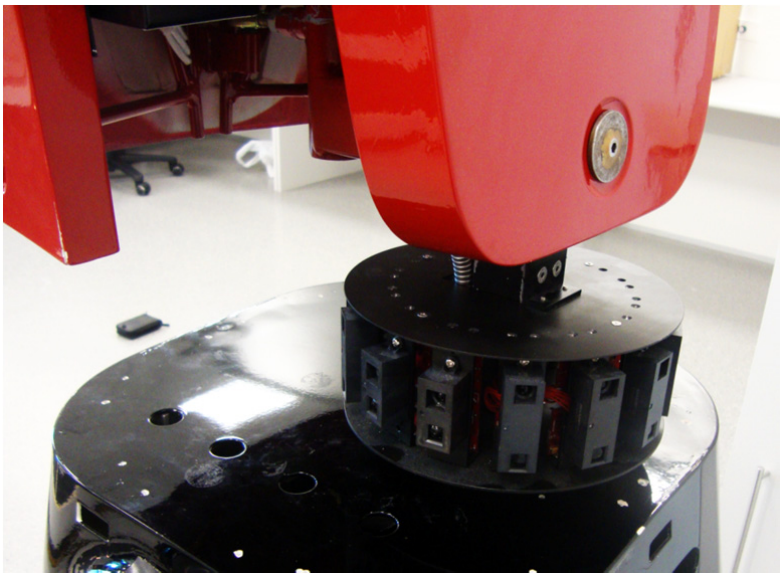

Figure 3.6 Sensor network mounted on MARVIN

The sensor network topology uses a RS-485 bus to interface to all sensors on the network (McClymont, 2011). Each sharp sensor can measure distances over a 25° angle. A dedicated

microcontroller board on each sensor node (shown in Figure 3.7) continually polls its sensor by performing the following steps (McClymont, 2011).

- Completes analog to digital conversion.
- Controls the sampling rate
- Calculates the distances from the sensor readings.
- And finally reporting the measurements back on the network.

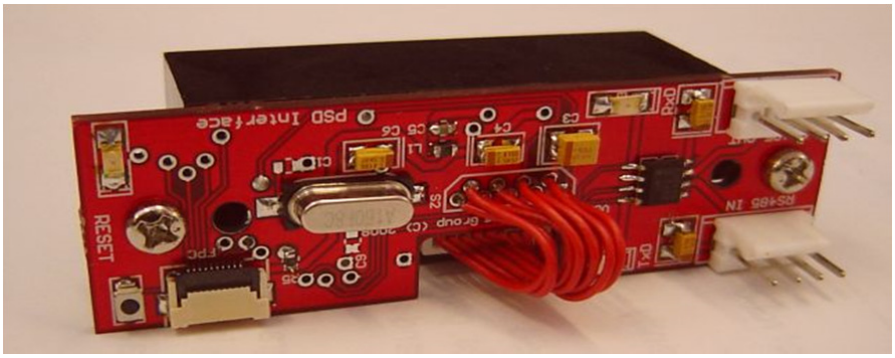

Figure 3.7 Sensor node and the dedicated microcontroller board

The sensor network has a detection field of 6.2 m in diameter and a dead zone of 1 m in diameter in

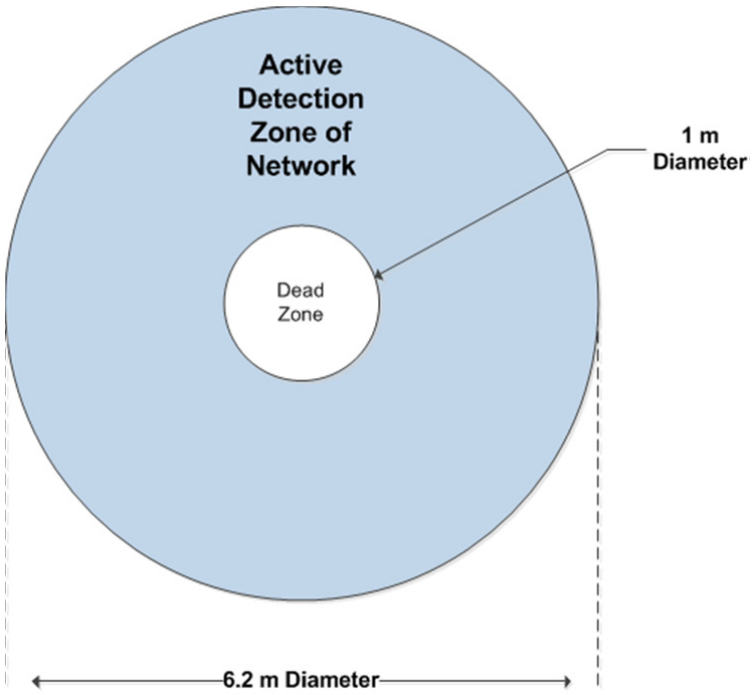

Figure 3.8 Active detection zone of sensor network (McClymont, 2011)

the centre. As MARVIN is primarily operating in an indoor environment, the four short range IR modules (GP2Y3A002K0F) have been mounted in lateral locations of the network chassis reducing the dead zone from 1 m to 0.6 m. This provides ranging capability allowing MARVIN to pass through office doorways that have a typical clearance of 0.8 m. Figure 3.8 illustrates the active detection zone of the sensor network including the 1 m diameter dead zone (McClymont, 2011).

Infrared sensors are preferred as they have faster responsive time and are more accurate due to their narrow field of view while being economical compared to other inexpensive options such as sonar sensors (McClymont, 2011). The disadvantages of using infrared sensors include interference from other infrared sources such as incandescent light bulbs and returning erroneous values for certain close-range distances.

3.5.2 Odometers

MARVIN utilises the HEDS-5701 optical encoder module, illustrated in Figure 3.9 for wheel position and velocity measurements.

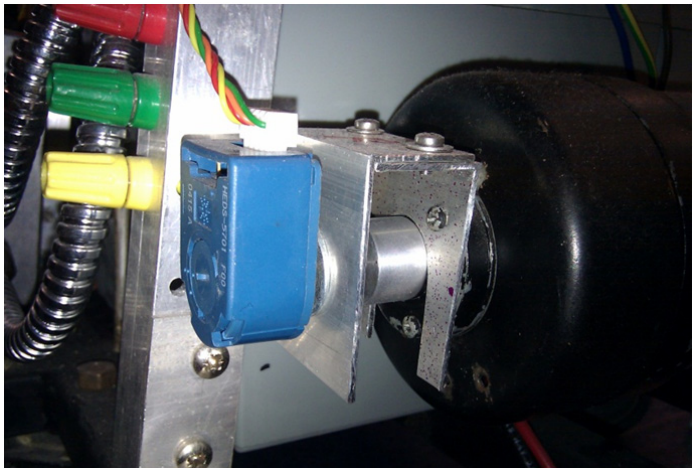

Figure 3.9 Encoder attached to the motor on MARVIN

Optical encoders are devices that convert a mechanical position into representative electrical pulses by means of passing light from a light source through holes on the perimeter of a circular disk (the code wheel) onto an optical receiver. The change in wheel position is measured by counting these generated pulses.

As the code wheel rotates, signals produced from the encoder's two output channels are in "quadrature", where one channel is shifted by 90 electrical degrees from the other. These are commonly called the quadrature "A" and "B" channels. The clockwise direction is defined as the "A" channel going positive (rising edge of the signal) before the "B" channel. This represents the

forward motion of the code wheel. When the relationship between the two channels is reversed the code wheel is spinning in the opposite direction. This is illustrated further by the signal output in Figure 3.10.

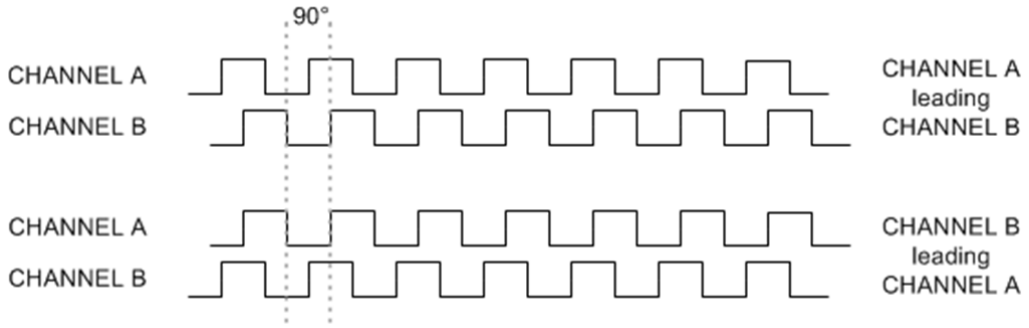

Figure 3.10 Quadrature Channel A and Channel B

The HEDS-5701 encoder module produces 256 pulses per revolution. Since the odometers are mounted on the motors, the gearing ratio of 51.56 results in a resolution of 13199 pulses per wheel revolution or 1 pulse per 0.079 mm of travel. Therefore, 1 metre of travel would result in 12731 pulses.

3.5.3 Sensor Network Interface Module

A sensor network interface module, operating as a master sensor node, was developed to allow the sensor network to operate independently of a control computer. This module establishes communication with a control computer to report back sensor data.

At the centre of the module is an Atmel ATMega128 micro-controller (MCU) that defines the application programming interface (API) to control the sensor network. The underlying hardware of the interface module also provides a communication translator between the RS-485 communication technology of the network and either an Ethernet or USB communication interface of a control computer (McClymont, 2011). As the number of nodes on a multipoint RS-485 network is limited to 32 the master microcontroller controls two RS-485 buses allowing up to 64 sensors in the network. The master microcontroller will employ one bus to access the sensor network mounted on MARVIN's base unit, and the other bus for additional sensors used when needed. Figure 3.11 shows the sensor network interface module.

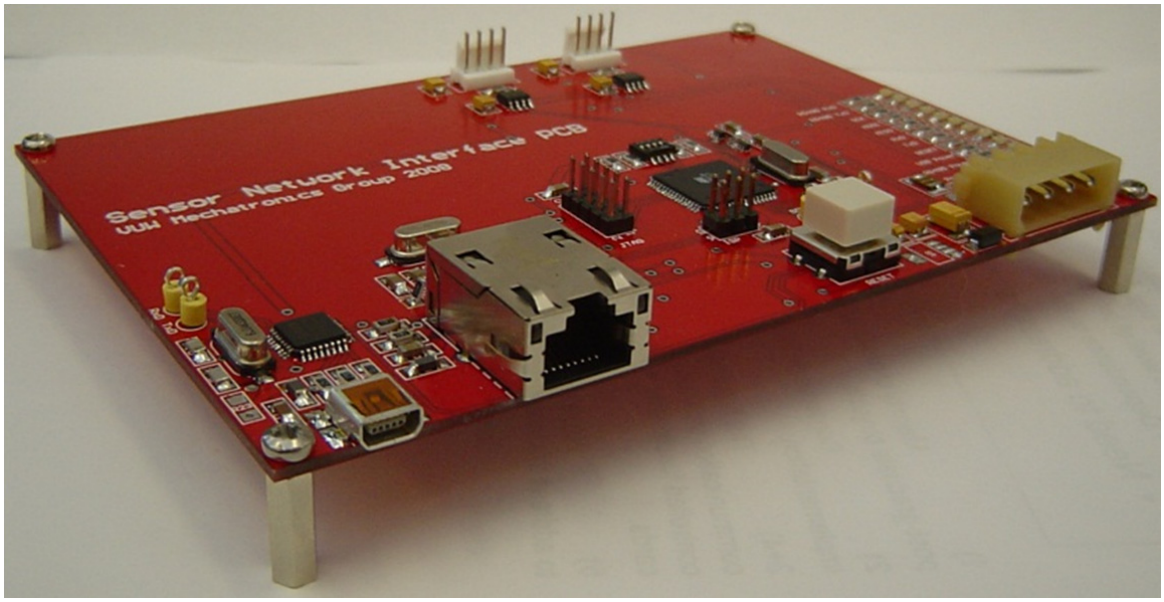

Figure 3.11 Sensor network interface module

The interface module is responsible for defining the sensor network at initialisation allowing a control computer to query the size and operational capabilities of the network (McClymont, 2011). The control computer can send commands to configure the sensor network and receive information about the sensor network and sensor measurements. Furthermore, the interface module responds with a set of error codes to these commands to indicate if the commands were successfully processed. Table 3.1 and Table 3.2 show the commands and the error codes respectively.

Table 3.1 Sensor network interface types (McClymont, 2011)

Packet ID	Packet Type	Parameters	Description
0x01	Dummy	-	This packet has no data payload and is primarily used to check correct communication on the command interface.
0x02	Reset	-	This packet resets the Network Master and Sensor Network.
0x20	Get Address	-	This packet requests the Network Master's hardware address. It is used to indicate to the control computer that the attached hardware is a sensor network.
0x21	Get Network Definition	-	This packet requests the network definition from the Network Master. The network definition is comprised of the number of detected nodes and, for each node, the node address and its calibration ID.
0x22	Get Node Data	Node	This packet requests the sensor data for a

			specified sensor. The data returned is the latest data the Network Master has read from the network.
0x26	Get Node Reading	Node	This packet requests that the Network Master instruct a specified sensor to take a reading and return the data back to the master and the control computer.
0x28	Get Auto Update	-	This packet requests the state of the Network Master's auto update flag. The auto update flag indicates if the master is automatically polling the sensor network, every 50 ms, for new sensor data.
0x43	Set Node Measurement	Node, Mode, Start Time, Stop Time, Channel	This packet instructs the Network Master to issue new measure task data to a specified node on the network.
0x44	Set Node Mode	Node, Mode	This packet instructs the Network Master to change a specified sensor's measurement mode to the specified mode.
0x45	Set Global Mode	Mode	This packet instructs the Network Master to globally change the measurement mode of all sensors on the network to the specified mode.
0x46	Set Auto Update	Auto Update Flag	This packet instructs the Network Master to change its auto update flag to the specified value.
0x47	Set Network Time	-	This packet instructs the Network Master to reset and sync the network time.

Table 3.2 Sensor network interface error codes (McClymont, 2011)

Error Code	Error Type	Description
0x01	No Error	This bit is set if no error has occurred.
0x02	Packet ID Invalid	This bit indicates that the received packet ID does not match a packet ID defined on the master or control computer.
0x04	Checksum Error	This bit indicates that the checksum byte did not match the checksum value of the packet. Data corruption must have occurred during transmission.
0x08	Parameter Invalid	This bit indicates that a parameter sent in the data payload is invalid, for example, if the control computer requests sensor data from a specified sensor that cannot be found on the network.
0x10	Frame Error	This bit indicates that a communication error occurred on the sensor network. This code is set when the control computer instructs the network master to send data to a sensor node but the master could not achieve the data transfer.
0x20	reserved	Reserved for future use.
0x40	reserved	Reserved for future use.
0x80	reserved	Reserved for future use.

3.5.4 PC Hardware

At the commencement of this project MARVIN was not equipped with a control computer to host high-level software. The hardware architecture currently installed on MARVIN encouraged a shift away from a custom made microcontroller board or an embedded controller to a standard PC platform. PC platforms have numerous advantages over embedded controllers, including:

- Multiple communication interfaces such as wireless, Ethernet and USB
- Provides support for human interface devices (HID) such as mouse, keyboard and monitor
- Support for multiple operating systems
- Increased magnitude of processing speed and memory
- Improved code debugging
- Less hardware design necessary

But standard PCs do have a few flaws that limit their usefulness in a mobile robot:

- Requirement on AC-power supply
- Space occupied

A standard PC case is too large to fit inside MARVIN's chassis, so it was decided to employ a laptop PC. Laptop PCs are available in different form factors and usually theirs sizes are proportional to their prices.

The minimum requirements for Microsoft Robotics Developer Studio runtime environment is 1 GHz processing power with 512 MB memory. To satisfy this requirement a netbook, a small and a lightweight category of laptops, was selected rather than a full-sized laptop. This has the added advantages of low weight, low cost and can easily accommodate inside MARVIN's chassis. Unlike other small-sized computers such as PDAs, netbooks remain competitive with laptops in terms of price and performance.

The only significant drawback is that unlike standard PCs it is much more expensive to upgrade, limiting the potential for future expansion. However, for the immediate future the selected notebook should have sufficient processing power for complex calculations.

The specifications for the netbook (shown in Figure 3.12) are as follows:

CPU:	Intel Atom N270 (1.6 GHz)
RAM:	1 GB

Hard Disk:	120 GB, 5400 rpm
OS:	Windows XP Home Edition SP3
General purpose I/O Ports:	3 × USB 2.0
	1 × VGA video-out
Connectivity:	Realtek 10/100 Mbit/s Ethernet
	Atheros 802.11b/g WLAN
DC power rating:	30 W (19 V, 1.58 A)

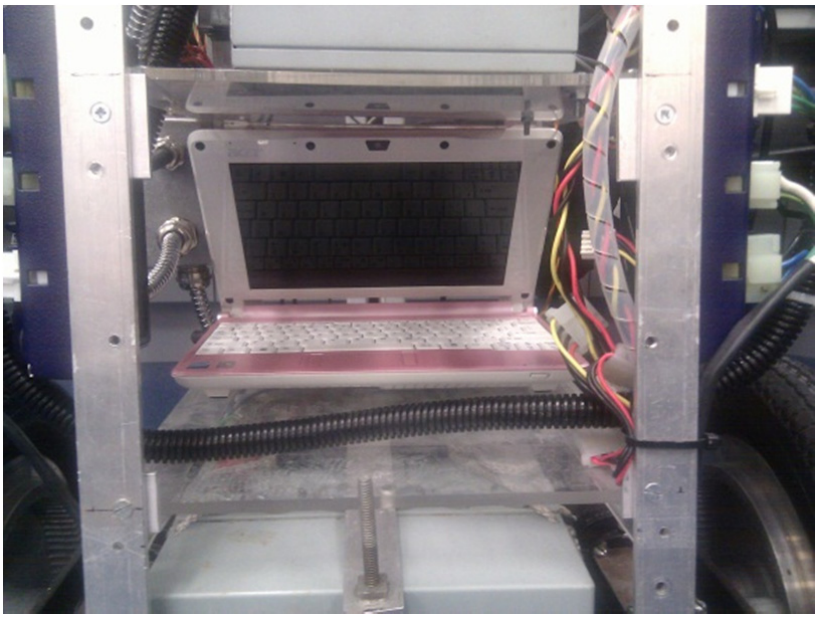

Figure 3.12 Netbook installed in MARVIN's chassis

A cheap universal car laptop adapter (rated maximum at 20 V, 40 W output) was used to convert voltage from one of MARVIN's 12 V batteries to the netbook's rated voltage. A typical small car battery such as the battery installed on MARVIN is rated at around 45 amp-hours (Ah) i.e. it supplies 1 A for 45 hours. Unfortunately, the operational time of the laptop using these batteries can only be determined experimentally due to the age of the batteries and the non-linear relationship between the load and the discharge time. When the experimentations mentioned in chapter 7 were carried out, the laptop was found to be operational for about 30 minutes before a noticeable drop was seen in the voltages of the batteries.

3.5.5 Upgrades

The original Rhino driver module as shown in Figure 3.5 had the encoder channels A and B connected through a quadrature decoder to the microcontroller. However, at the onset of this

project, master's student Johnny McClymont changed the module to bypass the decoder and use the microcontroller directly to decode and dissect the encoder signals. This was done due to erroneous encoder values resulting from the quadrature decoder. It was much cheaper and less time consuming to alter the existing PCB rather than designing new hardware. The encoder interface is now directly controlled by the microcontroller interrupts. This is illustrated below in Figure 3.13. Figure 3.13 shows the removal of the quadrature decoder which was connecting the encoders to the microcontroller (shown in Figure 3.5).

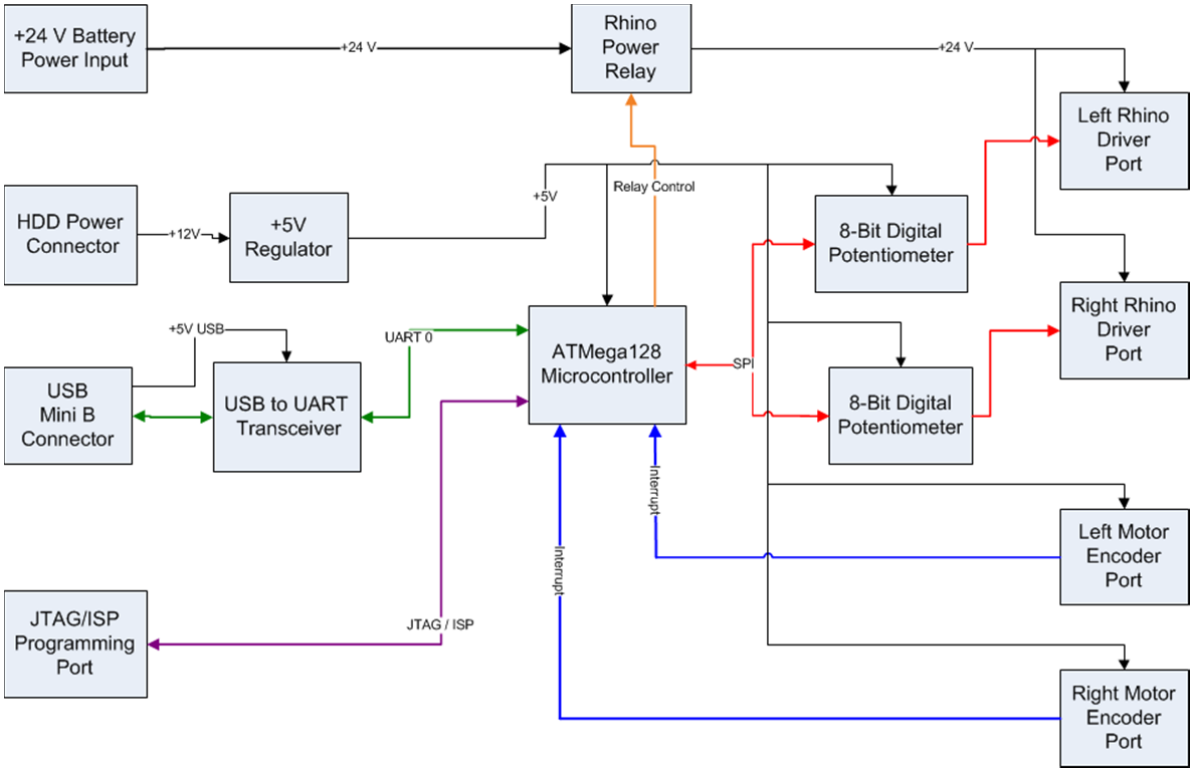

Figure 3.13 New Rhino driver module

Furthermore, the existing velocity controller implemented in the firmware was not functional. Hence, a decision was made to implement a new velocity controller that enabled the manipulation of MARVIN's wheels independently. To accommodate the new velocity controller the following new instructions were also introduced.

- *Stop* – Emergency stopping the motors. This causes the motors to brake immediately.
- *DriveUpdates* - This triggers the Rhino driver hardware module to keep track of the total number of encoder ticks updated on the left and right wheels. These updates will be sent automatically via serial communication to the control computer at a frequency of 1 kHz.

The control computer can set the velocity of the right and the left wheels in terms of odometer pulse counts. The microcontroller converts this velocity command to the digital potentiometer values to drive the right and left Rhino controllers appropriately. The encoders attached on the motors, monitor the velocities ensuring the motors are driven at the specified speeds. To achieve this feedback, a PID (Proportional, Integral and Derivative) controller is implemented on the microcontroller. The functional diagram of the PID controller is shown below.

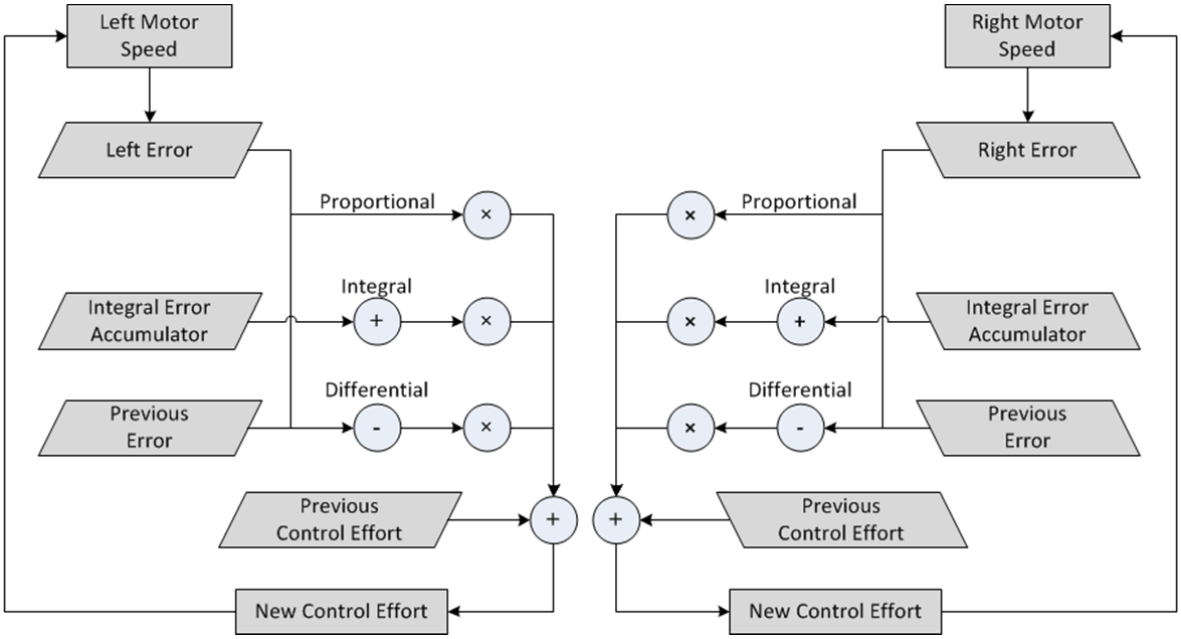

Figure 3.14 Functional overview of the PID controller

As illustrated in Figure 3.14, the microcontroller utilises two independent feedback loops to control MARVIN's right and left motors. Both the loops are running at 1 kHz or 1000 microcontroller clock cycles per second. Each control loop is used to set and control the velocity of its respective wheel, based on the required odometer pulse count. The error value of each loop is the result of the difference between the required velocity set by the control computer and the current velocity measured by the encoder. This error value is then scaled by three PID correcting terms, Proportional (K_p), Integral (K_i) and Derivative (K_d) as given by Equation 3.1. The sum of the terms constitutes the input to each of the Rhino controllers which results in the motors turning.

$$u(n) = u(n-1) + K_p e(n) + K_i \sum_{k=0}^{n} e(k) + K_d(y(n) - y(n-1)) \qquad \textbf{Equation 3.1}$$

The main function of both the control loops is to ensure that the wheel velocity set points are reached as quickly and accurately as possible. This is vital to ensure both wheels have closely matched velocities as MARVIN utilises a differential drive configuration.

MARVIN will primarily operate with dynamically changing velocities and as such each of the control loops is tuned manually by changing the PID correcting terms. A detailed description of the results from the tuning process is given in chapter 7.

3.6 Summary

At the start of this project, MARVIN was not equipped with a PC to host the navigation system thereby denying the control of the actuators, sensors and the hardware modules. After careful consideration, a decision was made to employ a laptop PC with a form factor that was small enough to fit inside MARVIN's chassis. Enhancements were also made to the Rhino driver module which is responsible for manipulating the motors and obtaining encoder feedback. The module was changed to bypass the quadrature decoder and use the microcontroller directly to decode and dissect the encoder signals. Two tuned PID controllers were also introduced to set and maintain the desired velocities for the left and right motors respectively.

Chapter 4 Navigation Architecture

4.1 Navigation Overview

The main goal of this project is for a mobile robot to be able to navigate autonomously from a known start point, generating safe paths through an environment, while achieving goals. This means that the mobile robot will utilize sensor data to evaluate terrain and locomotion to determine an appropriate path, being aware of its position at all times.

Autonomous navigation of mobile robots can be achieved by different architectures. Navigation architecture defines how mobile robots should integrate the ability to sense the surroundings, perceive the working environment, plan a trajectory and using the information execute a proper reaction. Three main types of navigation architectures which differ in speed, modularity and representation include reactive, deliberative and hybrid systems (Maja J. Mataric, 2007). The hybrid system involves the combination of the reactive and deliberative control architectures.

Deliberative architectures, work in a more predictable way (compared to reactive), have a high dependency on a precise and complete world knowledge, and can generate optimized performance (relative to the world representations) and trajectories for the robot (Praneel Chand & Carnegie, 2011). Compared to deliberative architectures, reactive architectures respond much faster to dynamic changes in the environment, employ minimal world representations, and require less computational power. Hybrid architectures combine the benefits of both approaches: the intelligence of deliberative control with reactive behaviours (Praneel Chand & Carnegie, 2011).

The hierarchical hybrid navigation system developed at Victoria University forms an integral part of this thesis. Figure 4.1 shows the various components. This navigation system consists of three layers.

- Deliberative layer – indicated by red dotted lines in Figure 4.1. This layer consists of the environment map and path planner components.
- Reactive layer – indicated by the blue dotted lines in Figure 4.1. Combination of path tracking with reactive control produces the reactive system.
- Third layer – This layer consists of other modules: localisation, information extraction and sensor fusion, and low level motion control.

Deliberative and reactive navigation architectures can be classified according to the relationship between sensing, planning, and acting components in the architecture.

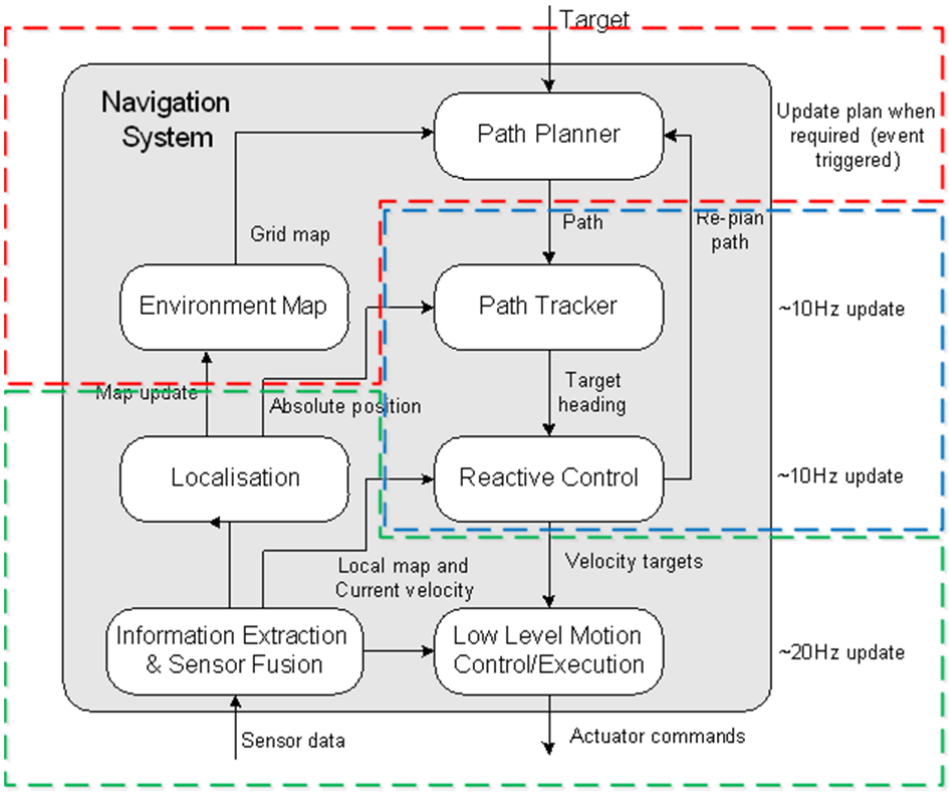

Figure 4.1 Overview of the hierarchical hybrid navigation system (Praneel Chand & Carnegie, 2011)

4.1.1 Deliberative Component

The deliberation indicates thinking in decision and action planning. Deliberative architecture enables a deliberative robot to perform high level tasks that are too difficult to perform without planning (Qureshi, Terzopoulos, & Gillett, 2004). This layer introduces the planning step between sensing and acting. The deliberate layer handles: mission planning and reasoning, localization, and path planning. Computationally, tasks in this layer can be expensive and therefore take a relatively long time. Additionally, these tasks also require complex processing.

This planning is based on a map of the environment in combination with the environment information acquired by the sensors. A rectangular occupancy grid has been selected for the deliberative component of the hierarchical navigation system (described earlier) in order to represent the robot's environment (Praneel Chand & Carnegie, 2011). This occupancy grid map is generated by dividing the environment into discrete cells and assigning unit interval values to represent occupancy probability (Praneel Chand & Carnegie, 2011). A modified A* algorithm is used for searching the occupancy grid for an optimal path.

4.1.2 Reactive Component

Reactive architectures are composed by a set of reactive behaviours and there is no planning based on a global map or model of the environment (Junior, Parikh, & Junior, 2006). In other words, sensing is directly associated with acting. In reactive architectures, control systems navigate in real-time reacting to current sensorial information that is perceived from the environment (dos Santos, 2008).

The reactive component of the proposed hybrid navigation system combines a modified version of the dynamic window approach with a polar histogram technique. A target heading angle is determined from the path tracker which is then fed to the direction sensor that produces a modified target heading as output. The modified dynamic window approach decomposes the modified target heading angle into linear and angular wheel velocities (υ, ω) which are then input to the low level motion controller.

4.2 Sensors

As stated in the previous chapter the selected mobile robot platform, MARVIN, is equipped with a variety of sensors. Depending on the operating environment, all sensors have a degree of uncertainty in their readings. This means that sensor measurements are not reliable and often require the combination of one or more sensors in order to sense the environment accurately. MARVIN has encoders and a 360° field of view network of IR sensors.

4.2.1 Odometers

Shaft encoders attached to the wheels of MARVIN accurately measure and in turn help to control the rotation of the wheels. The robot's current position can be calculated using the measurements from the encoders and kinematics. However, this task becomes challenging due to the accumulating errors that are inherent in odometry measurements. The accuracy of the measurements is limited by factors such as wheel slippage, missed digital pulse counts and transmission slop (Victorino, Rives, & Borrelly, 2000). This results in an increase in the difference between the actual distances the robot moves and the distance readings obtained from the odometry measurements.

The theoretical odometer conversion factor of 12732 pulses per metre only approximates the actual encoder counts per metre ratio. To obtain more accurate conversion factors for each wheel, experiments were performed by driving MARVIN in a straight-line motion. These experiments, discussed further in section 7.1, provided ratios between the actual distance travelled and the

distance measured by the two encoders. The average of these ratios is multiplied with the odometer conversion factor to reduce systematic error.

4.2.2 Rangefinders

The rangefinders employed on MARVIN use the process of triangulation to find a distance from a particular point to an area of interest. As illustrated in Figure 4.2, in order to determine the location of the cardboard box accurately, not only is the distance from the sensor to the box required, but also the angle, inclination and mounting position of the sensor (McClymont, 2011).

The distance measurements for the sensors are calculated using the power equation given in Equation 4.1. In this equation, x represents the dimensionless sensor data, A represents the co-efficient term, B represents the exponential term and C represents an offset (McClymont, 2011).

$$Distance = Ax^B + C$$ **Equation 4.1**

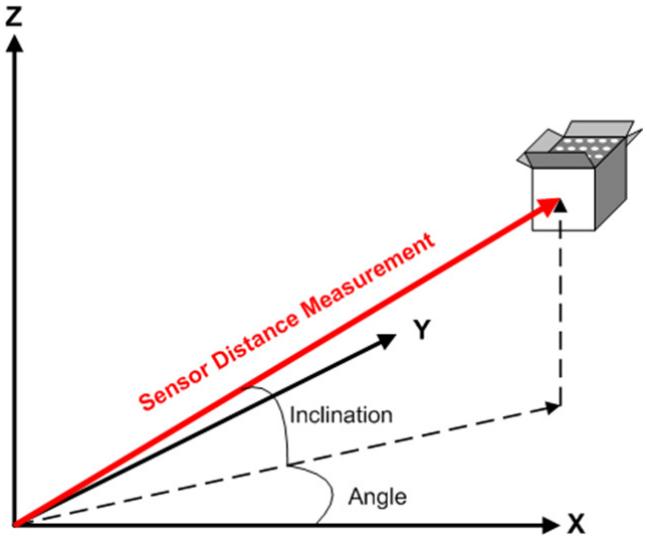

Figure 4.2 Orientation of Sensor measurement in 3-D Environment (McClymont, 2011)

4.3 Internal Representation

In order to effectively combine data from MARVIN's sensors, it must be converted into an internal representation that is established by all the sensors.

4.3.1 Position and Orientation

Assuming no wheel slippage occurs, each of MARVIN's wheel movements results in a change in MARVIN's position and/or orientation. MARVIN's position changes but the orientation or the heading remains the same when the wheels move the same distance in the same direction. It can be said that MARVIN is moving in a straight line. If each wheel moves the same distance, but in opposite directions, MARVIN's position remains the same but the heading changes. It results in a stationary turn. A combination of these two extremes will result in a moving turn shifting the robot's position and heading. Several equations given below are combined to determine these two variants.

The distance the robot's centre has travelled in an arc since the last position calculation is given by Equation 4.2.

$$Arc\ length = \frac{(left\ encoder\ count + right\ encoder\ count)}{2.0} \qquad \textbf{Equation 4.2}$$

When the robot first starts from the initial position, the left and right encoder counts are set to zero. Equation 4.3 determines the current angle the robot's centre has travelled.

$$Theta = \frac{(left\ encoder\ count - right\ encoder\ count)}{wheel\ base\ distance} \qquad \textbf{Equation 4.3}$$

The wheel base distance is the distance between the left and right wheels. It is useful to have a system that returns zero for straight drive, a positive value for clockwise rotations, and a negative value for anti-clockwise rotations. Thus, left encoder count has been set as the reference in the equation above.

Equation 4.4 provides the distance travelled by the robot's centre.

$$Distance = \frac{(Arc\ length) * \sqrt{2(1 - \cos(Theta)}}{Theta} \qquad \textbf{Equation 4.4}$$

Using Equations 4.2 – 4.4 a set of Cartesian coordinates is derived representing MARVIN's positing and heading. Equations 4.5 and 4.6 determine X and Y positions respectively.

$$X\ position = Distance \times \cos(Theta) \quad \textbf{Equation 4.5}$$

$$Y\ position = Distance \times \sin(Theta) \quad \textbf{Equation 4.6}$$

In the coordinate system, X position represents lateral motion with positive values to the right and negative values to the left. Y position represents forward motion as positive values and reverse motion represents negative values. The heading (Theta) represents rotations of the robot in radians where movement in a straight line is zero radians, positive values mean rotations in the clockwise direction, and negative values mean rotations in the anticlockwise direction.

4.3.2 Localisation

To determine the position and orientation further, the offset and the heading of the robot are extrapolated from the measured corridor wall distances. The comparison between the measured wall distances with the expected position of each wall results in an offset. The relative distance measured by multiple adjacent rangefinders is used to derive the heading.

To eliminate transient signals generated by objects momentarily blocking the sensors, the rangefinder distances are filtered by comparing each range with the mean average of the last ten measurements. The range is not used in the calculation of the position and heading if the difference is larger than 0.2 m. This prevents the robot from reacting to people walking past in the corridor.

Each rangefinder's position and orientation with respect to MARVIN's position and heading are added to the offset and heading recorded from the odometers to obtain coordinates relative to the corridor centre axis (calculated by Equations 4.7 and 4.8). These coordinates are used to predict the wall distance measurements obtained from the rangefinders. This is achieved using Equation 4.9. These resulting measurements are used to ignore the rangefinders facing away from the walls.

$$y_{IR} = y_M + x'_{IR}\cos(\theta_M) + y'_{IR}\sin(\theta_M) \quad \textbf{Equation 4.7}$$

$$\theta_{IR} = \theta_M + \theta'_{IR} \quad \textbf{Equation 4.8}$$

$$d_{IR} = \frac{y_W - y_{IR}}{\sin(\theta_{IR})} \quad \textbf{Equation 4.9}$$

The coordinates of measured objects relative to the robot's position and orientation are determined by Equations 4.10 and 4.11.

$$x_{OBJ} = x'_{IR} + d_{IR}\cos(\theta'_{IR}) \quad \textbf{Equation 4.10}$$

$$y_{OBJ} = y'_{IR} + d_{IR}\sin(\theta'_{IR}) \quad \textbf{Equation 4.11}$$

MARVIN's offset is calculated from each valid rangefinder distance using Equation 4.12.

$$y''_M = y_W - x_{OBJ}\sin(\theta_M) - y_{OBJ}\cos(\theta_M) \quad \textbf{Equation 4.12}$$

The heading obtained using valid distances from adjacent rangefinder pairs is given by Equation 4.13. Both possibilities of the resulting heading being equal or opposite to the actual heading are compared with the heading measurement obtained from the odometer, and the closest match is selected.

$$\theta''_M = -\tan^{-1}\left(\frac{y_{OBJ2} - y_{OBJ1}}{x_{OBJ2} - x_{OBJ1}}\right) \quad \textbf{Equation 4.13}$$

Figure 4.3 and the list below summaries the parameters used in the equations above.

y_M Offset from centre axis (m). Obtained from 4.6.

θ_M Heading determined by Equation 4.3.

y_{IR} Rangefinder offset (m).

θ_{IR} Rangefinder heading (rad).

x'_{IR} Rangefinder distance with respect to MARVIN's centre (m).

y'_{IR} Rangefinder offset with respect to MARVIN's centre (m).

θ'_{IR} Rangefinder heading with respect to MARVIN's centre (rad).

y_W Wall offset (m).

d_{IR} Predicted distance measured by rangefinder (m).

x_{OBJ} Distance of object (m).

y_{OBJ} Offset of object (m).

y''_M Final offset of MARVIN's offset from rangefinder (m).

θ''_M Final heading of MARVIN calculated from rangefinder (rad).

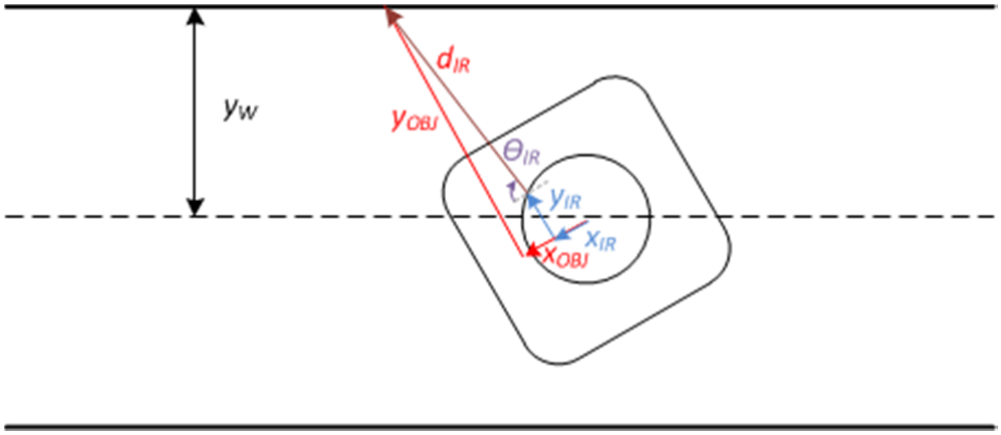

Figure 4.3 Calculating offset and heading measurements from the rangefinder with respect to MARVIN

4.4 Sensor Fusion

Sensor fusion deals with merging of sensor data from multiple sensor sources. MARVIN's multitude of sensors provides useful information however their individual importance differs with circumstance. For example, rangefinders are less accurate over short distances (as mentioned in section 3.5.1), but the errors generated from the rangefinder measurements do not increase over time. Conversely, odometers are reliable over short distances, but cumulative errors which are generated over long distances limit their long-term usefulness. The navigation system solves these problems utilising sensor redundancy which allows the sensors to provide the same information, but with different degrees of accuracy. To take advantage of each sensor's strengths and reduce the weaknesses, overlapping sensor signals are combined, or fused.

One of the simplest and most intuitive methods of signal-level fusion is to take a weighted average of redundant sensor information provided by the sensors (Kapach, Edan, & Xiao, 2007). MARVIN's sensor fusion algorithm uses a form of dynamic weighted average. For a system such as MARVIN, equipped with relatively few sensors and a comparatively simple operating environment, it was not justifiable to consider other sensor fusion implementations such as Dempster-Shafer or Bayesian techniques. These techniques do not provide enough of an improvement to justify the complexity of their implementation and they are not applicable due to the dependant nature of the two sensors, the odometers and the rangefinders. On MARVIN, dynamic weighted average

algorithm allows each of the sensors to make a contribution, however these sensors are still prioritised according to estimated uncertainties.

MARVIN's odometer weights would be much higher than the rangefinder sensors, given their accuracy over short distances. The rangefinders would correct the odometer errors over time even with low weights. The rangefinder weights could become velocity dependant, however increased weights would result in rapid changes in the offset and heading measurements. This may also have adverse effects on MARVIN's reaction. Hence, the weights will remain the same throughout normal operation. The rangefinder measurements will be temporarily zeroed in situations where the measurements can be misleading such as passing through an open door or a corridor intersection.

4.5 Summary

A three layer hierarchical hybrid navigation system is identified to perform autonomous indoor navigation of MARVIN. The navigation system is classified to deliberative, reactive and low-level motion control layers. These layers have been further simplified into components and these components interact together to form the navigation system. Environment map and the path planner components form the deliberative layer while the reactive layer comprises the reactive control and path tracking components. As part of the low-level motion control, sensor data from rangefinders and encoders has been fused together using a form of the dynamic weighted average technique to provide position and orientation. The target position and heading in conjunction with the current position and orientation of the robot are input to the deliberative and reactive layers to determine the target velocity and angular velocity of MARVIN.

Chapter 5 Software Interfaces

This chapter presents the details of the software development languages, environments and tools that were used to develop MARVIN's control and navigation systems. MARVIN's software utilises two programming languages, GNU C and C#. GNU C has been used to implement low-level applications for the Atmel AVR microcontrollers on MARVIN's hardware modules. The navigation system has been implemented in C# to run in a specialized environment called MRDS. MRDS is designed to execute on any recent Windows-based PC. The following sections will further detail these software interfaces.

5.1 MRDS

Microsoft released Robotics Developer Studio (MRDS) with the explicit goal of providing an industry software standard for robot control including a simple programming and simulation environment that allows easy creation of sophisticated, multi-agent robot systems. The MRDS environment enables software written for one robot to work on another robot with similar capabilities. It is built on a .NET/Windows-based runtime environment that facilities not only robotics development but is also applicable to a wider range of real time applications such as embedded applications. It consists of a number of components that include the development and runtime systems. Because it is built on a .NET framework it is targeted towards a number of Windows operating systems from Windows Mobile to Vista.

The applications for this project are built specifically on MRDS 2008 R3 version based on .NET framework 3.5.

Following the service-oriented architecture, every application module running on MRDS interacts as a service that subscribes to or publishes to other services, similar in nature to Web services. These services are highly decoupled, providing the ability for modular reuse of the code (Jackson, 2007). For the communication between these services, a highly abstracted and distributed messaging system is provided, which allows services to interact both on the same computer system, and across a variety of network protocols, hiding the implementation details from the developer (Kamath, 2009). This serves to provide a common interface that simplifies the basic send and receive operations, making the services messaging layer independent.

In a typical application developed on MRDS for a robot, there are usually several services running and these can be categorized as low-level and top-level. Low-level services interface directly with hardware and top-level services control the behaviour of the robot. These top-level services typically interface only with other services and are called orchestration services. Figure 5.1 illustrates an example of the low-level and top-level services. In this example, the low-level services are defined to interact with each hardware component such as a sensor or an actuator. Services are not just limited to hardware applications but can also include implementations for business logic, remote PC control and Web-based error reporting applications (Jackson, 2007). Typically these are top-level services implementing high-level behaviours such as navigation or teleoperation of a robot. Figure 5.1 shows the orchestration services at the centre interacting with the low-level services on a differential drive robot application.

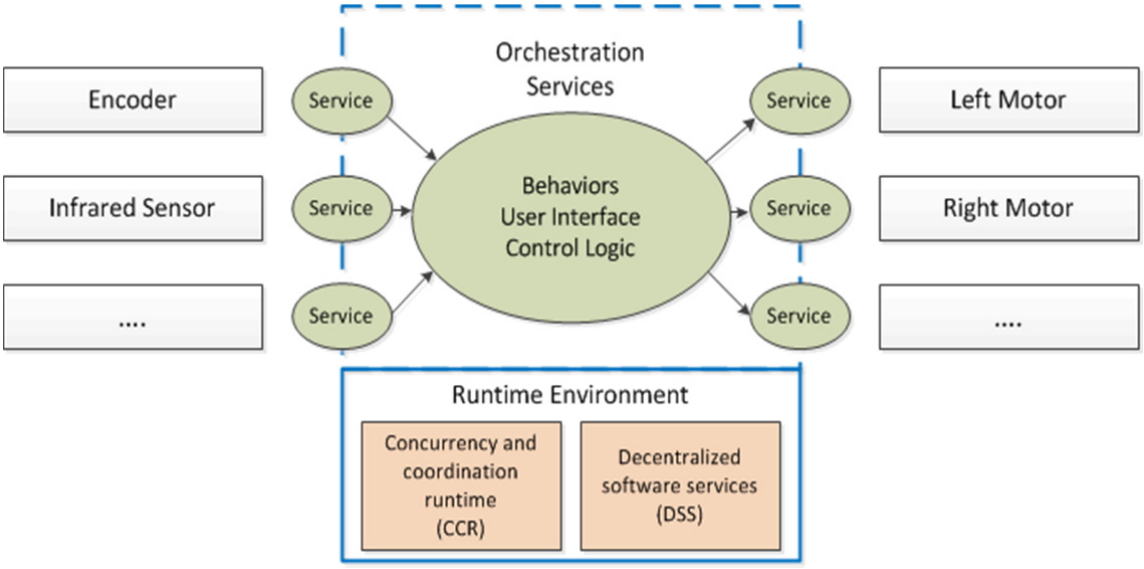

Figure 5.1 Overview of orchestration of services in a typical application

As shown in Figure 5.1, two managed libraries, Concurrency and Coordination Runtime (CCR) and Decentralized Software Services (DSS), comprise the run-time environment of MRDS. These are looked at closely in the next few sections.

MRDS defines a set of abstract services (also known as generic contracts) specifying APIs that can be used to communicate with common hardware components such as infrared sensors and motors. These services don't implement behaviour and they are available on MRDS to allow the control of a range of hardware without any programming effort. Incorporating these services with user-defined services gives the added advantage of using MRDS's built-in applications to manipulate the robots. For example, the generic differential drive contract service enables a built-in application called the

Dashboard to drive around any differential drive robot using a mouse or a joystick (Johns & Taylor, 2008). MRDS also includes several utility services, most of which automatically load whenever any service in the robotic application is started. These include:

- A control panel service which provides a Web interface to the end user displaying a collection of all the running services and providing links to configure each of these services.
- A message logging service that provides the debugging and diagnosis interface.
- A resource diagnostic service to provide additional information to assist in debugging and performance evaluation.

In addition, MRDS consists of two visual components, a 3D simulator known as Visual Simulation Environment (VSE) and a graphical programming environment known as Visual Programming Language (VPL).

The full-featured 3D simulator supports both indoor and outdoor environments and it is useful for prototyping simple robotic algorithms prior to running them on the actual hardware. The simulator comes with a variety of simulated robots and it is also extensible so that custom robots can be added (Johns & Taylor, 2008). However, given the characteristics of the custom robots the task of adding 3D images of these robots can be complex (Johns & Taylor, 2008). A screenshot of the 3D simulator with the integrated environment and the robots is shown in Figure 5.2.

Figure 5.2 Screenshot of the Simulation Environment (VSE)

VPL is a graphical tool that allows the rapid development and prototyping of orchestration services without writing any code. In VPL, programs are defined graphically in data flow diagrams rather than the typical sequence of commands and instructions found in programming languages such as C# (Johns & Taylor, 2008). Figure 5.3 shows a screenshot of the VPL environment.

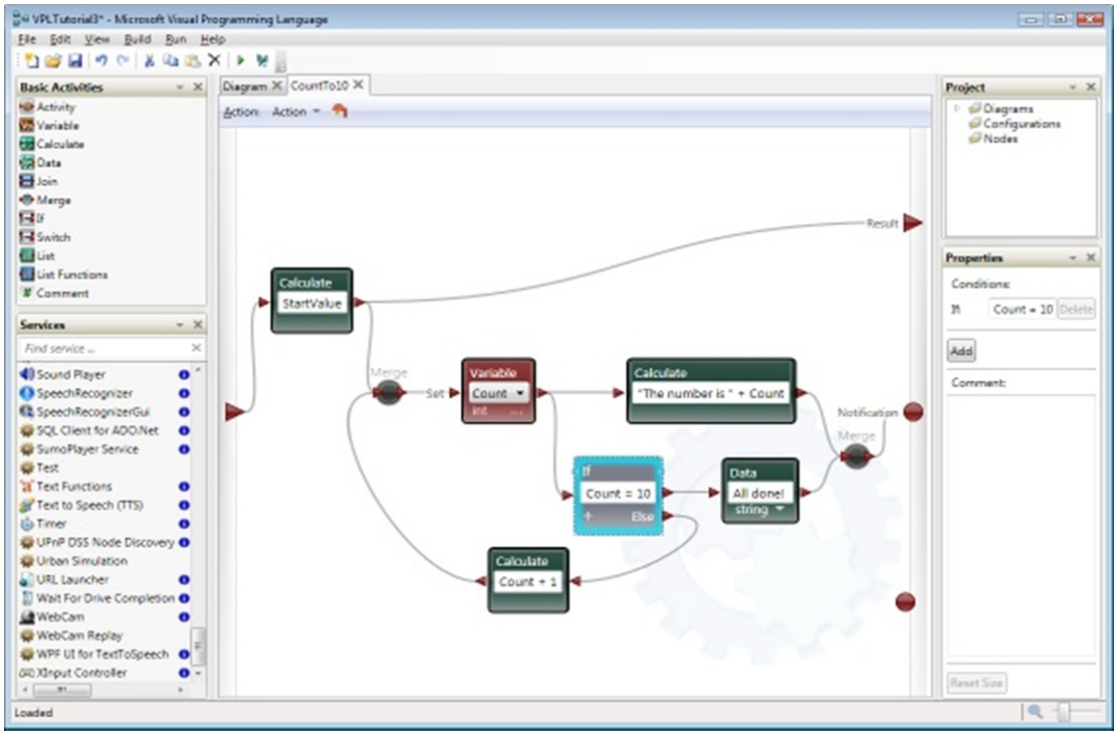

Figure 5.3 Screenshot of the VPL Environment

A decision was made to not pursue employing any of these visual environments due to the following reasons:

- At the onset of this thesis project, the simulator still has not reached its maturity at solving intricate dynamics. Given the dynamics of the real world environment together with the complexity of the hybrid navigation algorithm, it can be a very difficult task for the simulator to process and maintain the fidelity of the simulation while still allowing the navigation algorithm to update at a reasonable rate.
- The sensors in the real world often return noisy data. MARVIN includes non-generic hardware components such as the rangefinder sensor network that is not present in the simulator. Therefore, to program these components and to simulate noisy data can be a very difficult task.

5.1.1 CCR

CCR is a managed library based on .NET platform that provides classes and methods to help with asynchrony, concurrency, coordination, and failure handling. For robotics, real-time processing such as listening to sensors while controlling the actuators can be achieved with the concurrency and coordination features of CCR. These features are beneficial as they eliminate the conventional complexities such as manual multi-threading with the use of mutexes (mutual exclusions) and thus preventing deadlocks (Cepeda, Chaimowicz, & Soto, 2010). CCR enables segments of code that operate independently to pass messages and run in parallel within a single process (Johns & Taylor, 2008).

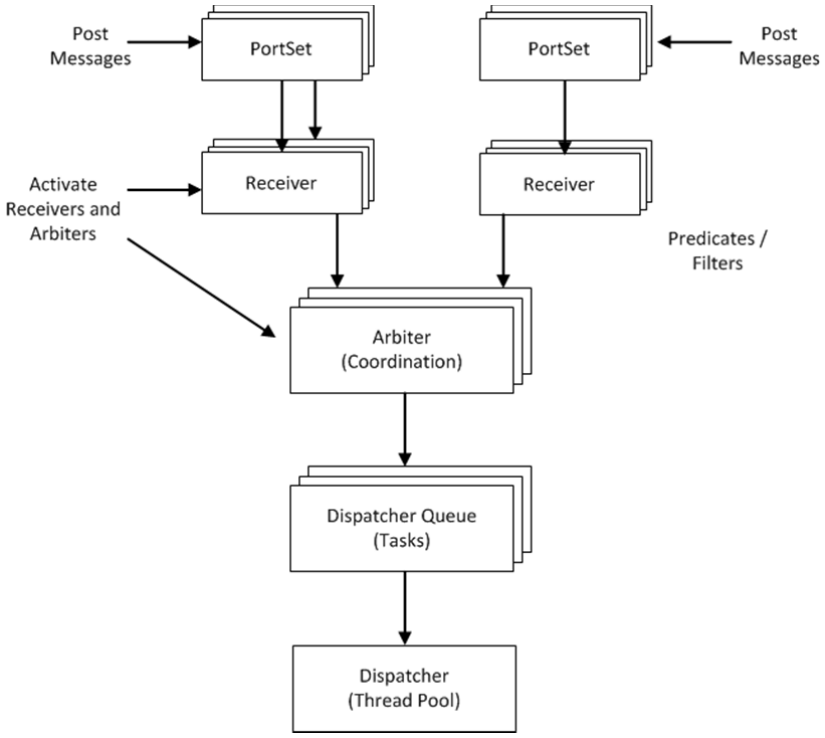

Figure 5.4 CCR Architecture (Johns & Taylor, 2008)

CCR exposes several primitives that address tasks and intra-task communication.

- *Port* - port is a structure CCR utilizes to listen (and/or modify) for messages from sensors and actuators concurrently, for developing actions and updating the robot's state. They are First-In-First-Out (FIFO) queues.
- *PortSet* – A collection of ports is called a portset. Ports can be utilized individually or as a group.

- *Receiver* – A receiver dequeues a message after it is posted to a port or a portset. Conditions can be set on receivers to create complex logical expressions, such as a *Choice* between two ports (a logical OR is created when a message arrives on either port).
- *Arbiter* – An arbiter evaluates the conditions set by a receiver.
- *Handler* – A handler is a piece of code implemented to perform a critical function such as sending a serial command to request for sensor data.
- *Task* – A task contains a reference to a handler. Once a receiver's conditions have been met, a task is queued to a dispatcher queue and then passed to a dispatcher for execution. When a task is scheduled to run, the referenced handler is executed.

These primitives are presented in Figure 5.4.

To handle failures, also called faults in CCR, in relation to web requests or exceptions, a structured approach similar to try/catch exception handling called causalities is used. Because of the multi-threaded nature of MRDS, errors could occur in a different thread from the original caller thread. Causalities enable these errors to be passed back to the original caller.

5.1.2 DSS

Like CCR, DSS is also a core runtime library of Robotics Developer Studio and it extends the concept of CCR (as described in the above section) across processes and even across machines. DSS uses a protocol called DSS Protocol (DSSP). DSSP is based on the Representational State Transfer (REST) model used for web development.

An application built with DSS consists of multiple independent services running in parallel. These services are mainly (but not limited to): hardware components such as sensors and actuators, software components such as user interfaces, orchestrators and repositories; or aggregations referring to sensor-fusion and related tasks (Cepeda et al., 2010). Also, services can be operating in a same hosting environment, or distributed over a network, giving flexibility for execution of computational expensive services in distributed computers (Cepeda et al., 2010).

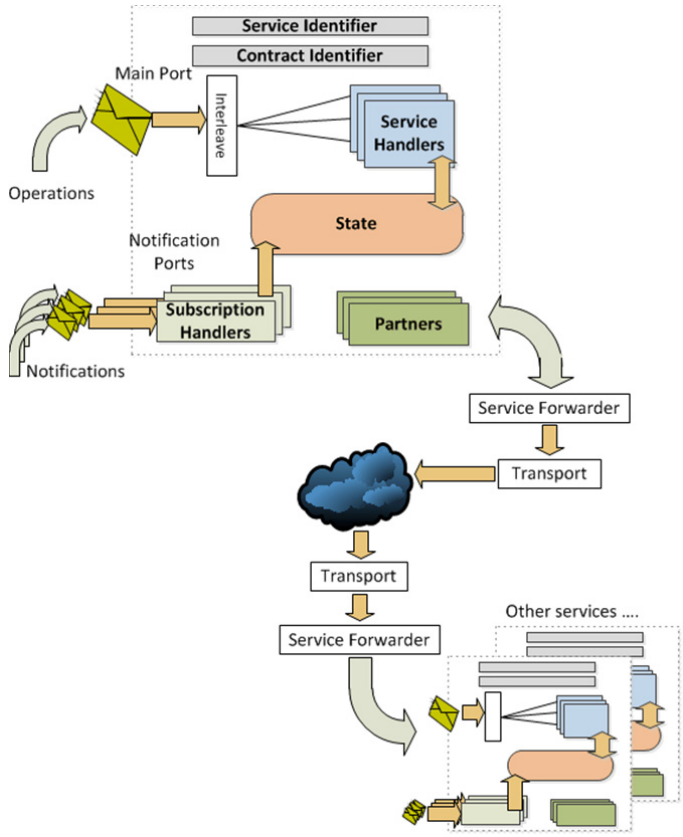

Figure 5.5 DSS Architecture (Johns & Taylor, 2008)

A graphic representation of services in DSS architecture is shown in Figure 5.5. A service consists of the following components:

- Contract – Defines the messages a service accepts, as well as a globally unique reference, called the Contract Identifier which is created, static and unique, within the service for identifying it across services. It is expressed in the form of a URI (Universal Resource Identifier). When multiple instances of a service are running in the same application, each instance will consist of the same contract identifier but a different service URI (Cepeda et al., 2010).
- State – Information that the service carries and maintains to control its own operation. For example, if the service interacts with a laser range finder, the state will carry basic information that is associated with the sensor such as distance measurements, angular range and sensor resolution (Cepeda et al., 2010).
- Behaviours – The set of operations that the service can perform and that are implemented by handlers. These handlers operate in terms of the received messages in the main port to develop specific actions in accordance to the type of port received.

- Execution context – The partnerships that the service has with other services, and its initial state. Through a component of DSS called event notification, a service can subscribe to a monitored service to receive announcements of changes to a service state. Also, each subscription will represent a message on a particular CCR port, providing differentiation between notifications and enabling for orchestration using CCR primitives (Cepeda et al., 2010). A special port known as the service forwarder (shown in Figure 5.5) can be used to link services and/or applications running in remote nodes in a process called partnering.

5.1.3 Programming Language

MRDS can be programmed in many common languages including VB, C++, C#, and VPL. However, most of the documentation and samples available in MRDS are coded in C#, thus the focus of this project will be to code in C#. It was decided to code in C# based on a number of other factors. These were:

- Complexity of the navigation system meant the programming language will have to accommodate strong typing and an object-oriented paradigm.
- Easy deployment in a distributed environment.
- Economical memory and processing power requirements.
- Preferred language for the development of DSS services.

C# is a general purpose programming language intended to be suitable for writing applications for both embedded and hosted systems.

To develop C# applications, Microsoft Visual Studio 2010 has been used as the integrated design environment (IDE). Visual Studio allows libraries, console and graphical applications to be designed, programmed, debugged and finally deployed. Following the installation of MRDS, the runtime libraries of MRDS, the build and deployment process will be setup automatically on Visual Studio.

The source code for a new service is split into several different files, a main code file, a state file, and a types file. In C#, a service comprised in the main file is normally defined as a class. Similarly, the state and the operation request types of a service are defined as two separate classes and are included in their respective source files. The state file also defines the contract of the service as a class and the contract identifier. This convention is used throughout this project.

The state and the operations port are instantiated in the service class followed by the Start method. It is required that a service class define a Start method which gets called during service creation so that the service can initialize itself. These components are described in detail in the next chapter.

5.1.4 User Interfaces

User Interfaces on MRDS can be developed by following two different approaches, Web Forms and a Graphical API in the form of Windows Forms (WinForms) and Windows Presentation Foundation (WPF).

Windows Forms (WinForms) graphical application programming interface is chosen to develop the user interface of MARVIN. Compared to Windows Presentation Foundation (WPF) API, WinForms is a mature and simple technology for the purposes of building user interfaces quickly. WinForms will only be visible on the local computer that is running the DSS node (Johns & Taylor, 2008). However, to introduce Teleoperation, a service can be used to display a WinForm and run it as a client on a remote computer to communicate with the main service.

Web Forms can also be used to display service parameters and it is the preferred method to make changes to these parameters remotely. This is achieved with HTTP requests HTTP Get (HttpGet) and HTTP Post (HttpPost). To display the parameters the data is required to be formatted using XSLT (Extensible Stylesheet Language Transformation).

5.2 Microcontroller Interface

5.2.1 GCC C

Firmware for the Atmel AVR microcontrollers (that is embedded in all the hardware modules) has been written in GNU C which is an open source programming language provided by the AVR Libc package. The AVR Libc package provides a subset of the standard C library and a set of tools to develop software for the Atmel AVR microcontrollers. The open source community has adapted the AVR Libc package to run on various operating systems such as Windows and Linux. Such an adaptation is WinAVR which is a suit of development tools that includes the AVR-GCC compiler and a development environment for the Windows operating system. The firmware for MARVIN's hardware modules is developed on WinAVR.

Johnny McClymont has developed a protocol for communication between MARVIN's hardware modules and a PC (McClymont, 2011). The firmware is written in GNU C and apart from the

modifications addressed in chapter 3 no other changes were done. Mr McClymont has also developed user interfaces, for the PC side, to control and test the hardware in C#.

5.2.2 Communication Protocol

Instructions are delivered to the microcontroller based on a request – response protocol. The modules will communicate on the USB interface only in response to a request from the control computer. A packet format is utilized to ensure correct operation of the interface and to guarantee data integrity (McClymont, 2011).

The command interface uses the ASCII control characters Data Link Escape (DLE) and End of Text (ETX) for packet framing. New frames begin with a single DLE character while frames end with a single DLE character followed by an ETX character. Byte stuffing is used to prevent DLE characters within the data payload or packet header bytes from being interpreted as a framing character (McClymont, 2011).

Byte Offset	Packet Field
0	Data Link Escape
1	Packet ID
2	Error Code
3	Packet Size
4	Data Payload
4 - 132	Packet Checksum
132 - 133	Data Link Escape
133 - 134	End of Text

Figure 5.6 Packet Structure for Command Interface (McClymont, 2011)

The command interface packet, shown in Figure 5.6, consists of the following fields (McClymont, 2011):

- Packet ID – identifies the type of command packet requested by the control computer.

- Error Code – holds the error code indicating any errors during the transmission of the packet.
- Packet Size – indicates the number of bytes in the packet.
- Data Payload – contains the data payload of the command. When issued by the control computer it contains command parameters and when issued by MARVIN's hardware modules it contains the requested status data.
- Packet Checksum – contains a checksum byte to indicate a corruption during transmission.

MARVIN's hardware modules expose unique command packets (different command packet IDs) and also common packets. Two of these common packets are Dummy and Reset shown in the table below. For the purpose of checking if a connection is active with a hardware module, the control PC can send a Dummy packet and identify the response Error Code. If the Error Code indicates an error, the control PC can send a Reset packet immediately to reset the hardware module.

Table 5.1 Common Command Packet types (McClymont, 2011)

Packet ID	Packet Type	Parameters	Description
0x01	Dummy	-	This packet has no data payload and is primarily used to check correct communication on the command interface.
0x02	Reset	-	This packet resets the Rhino Driver MCU module and re-initialises the Rhino Drivers.

5.3 Summary

This chapter details the runtime libraries of MRDS, CCR and DSS, and the contribution of each of these components towards developing the software architecture. CCR provides a feature set to enable segments of code to operate independently within an application. DSS extends CCR concepts by introducing functionality to develop service-oriented applications that can run across a network or on a single machine. C# is chosen as the primary programming language to develop these applications.

Chapter 6 Software Framework Methodology

In this chapter, the design and implementation of a software framework is presented. The software framework integrates all the essential components of the hierarchical hybrid navigation system described in chapter 4. After giving an overview of the functional model, it proceeds with a thorough description of the implemented software modules in the framework. It is the aim of this thesis to design the software framework with flexibility and reusability in mind.

6.1 Software Architecture

The software framework closely follows the Service-Oriented Architecture (SOA) and an event-driven programming approach (Chang, He, & Castro-Leon, 2006; Chen, 2006; Chen & Tsai, 2008). Traditionally, SOA has been applied in Web-based applications and was not considered to be suitable for robotic applications and embedded systems. This perception was changed after more efficient technologies were proposed, in particular after Microsoft released the SOA-based MRDS (Chen & Bai, 2008).

Compared to Object-Oriented Computing (OOC), SOA makes it easier to develop recomposable robotic applications taking care of the versatility and extensibility when using different kinds of sensors and actuators. It has been perceived that SOA applications are less efficient than OOC applications because of the additional layer of the standard interface, which makes it possible for language and platform independent communication and remote invocation implemented in Web services (Chen & Bai, 2008). Thus, SOA was considered to be an unsuitable paradigm for developing robotic applications as real-time performance is often required. However, MRDS's DSS runtime environment, which is SOA-based, has been designed to use Decentralized Software Services Protocol (DSSP) and Hypertext Transfer Protocol (HTTP) to provide a lightweight protocol for building high performance applications. Some of the key benefits of the adoption of the SOA paradigm are the reduced programming complexity, development time and maintenance costs.

MARVIN's navigation software framework was developed following the SOA paradigm. As shown in Figure 6.1, MARVIN's navigation software framework consists of three main components; Plant, Navigation System and User Interface, which are coupled with well-defined interfaces. The Plant is the low-level hardware control. It is responsible for communicating sensor data from the sensors and manipulating the actuators.

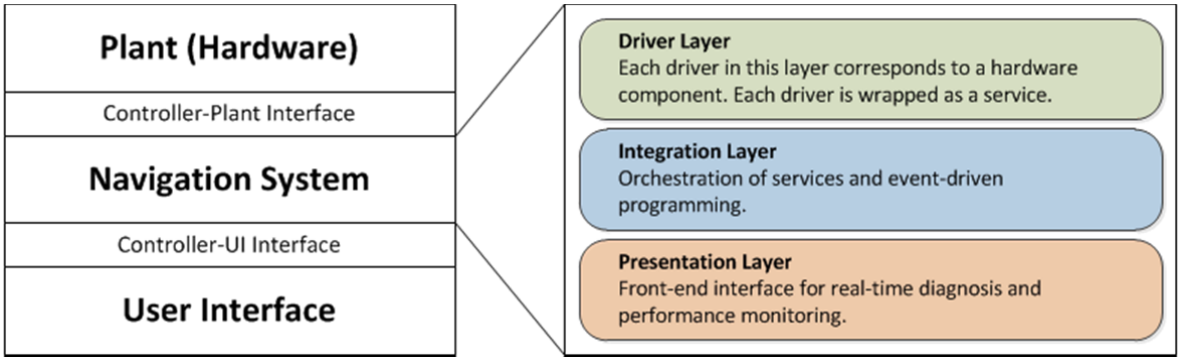

Figure 6.1 Overview of the software architecture

The second component is the heart of the navigation architecture where each of MARVIN's hardware components, Rhino drive module and sensor network module, is wrapped as a service and all these services are loosely coupled together. Figure 6.1 visualizes how the services are integrated as layers to make the system. This system is designed as 3-layer architecture. The bottom layer is the abstract layer for MARVIN's hardware. This provides the drivers as services that communicate directly with the sensors and actuators. The middle layer or the application layer combines all the services to provide orchestration. The top-most layer which is the visual interface layer is used for real-time diagnosis.

The third component provides the user interface as an additional service layer that complements the navigation system. It provides a graphical user interface for teleoperation of MARVIN.

Controller-Plant interface specifies the communication protocol between the Plant and the Navigation System. It is the Navigation System that initiates the communication by sending request commands to the Plant. In response, the Plant returns the current odometer and sensor information.

Controller-UI interface defines the communication protocol between the Navigation System and the User Interface components and it employs HTTP and SOAP protocols to establish this interaction.

Both the Navigation System and the User Interface will be discussed in detail later in this chapter.

6.2 Functional Model

Figure 6.2 visualizes the use cases of the implemented software framework, where Plant and User Interface are the identified actors. It is the User Interface that initiates the use cases for both manual and autonomous modes. The ***manual mode use case*** uses three use cases to drive straight and at an angle respectively, and setting the speed. Inheriting the functionality of the ***manul mode use case***, the ***autonomous mode use case*** includes the use cases for the following; starting and stopping autonomous mode, uploading a map of the environment, determining the current position of

MARVIN, setting the initial and target positions. These use cases are discussed in detail in the following subsections.

6.2.1 Use case "Manual Mode"

User Interface transmits nominal values of driving straight and at an angle and also speed to the Navigation System. Subsequently, applying normalization of straight drive, steering angle and speed to these nominal values, the Navigation System sends the new nominal values to the Plant. Plant periodically returns sensor data to the Navigation System, which then forwards the data to the User Interface.

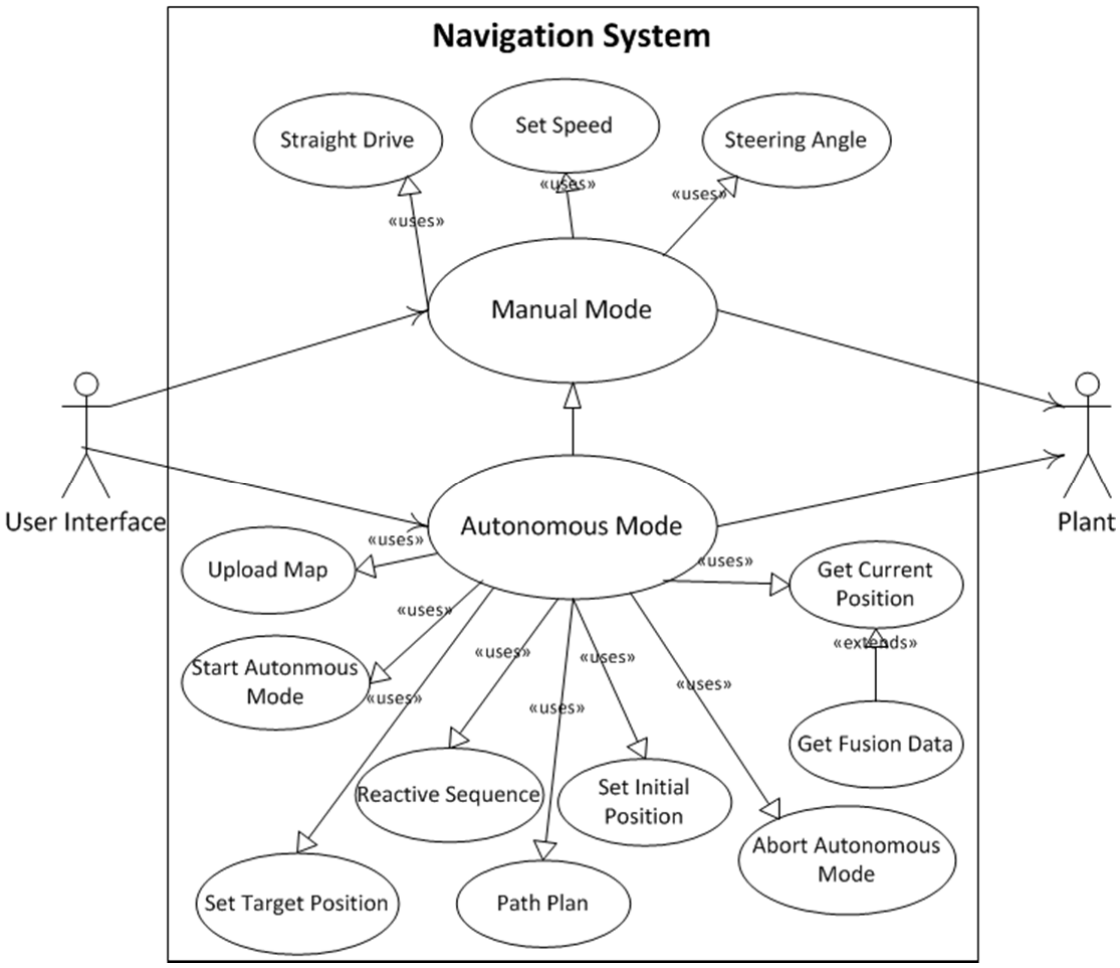

Figure 6.2 Use cases of the Navigation System

- *Set Speed* - The speed at which MARVIN is travelling can be set from the User Interface.
- *Straight Drive* – Driving straight means the wheels of MARVIN will be turning at the same speed and direction. The Navigation System applies the normalization needed to the received nominal values. The normalization process converts the desired velocity (metres

per second) to the low-level velocities of left and right wheels (encoder counts per second). This is achieved when the velocity is divided by the distance each wheel travels per encoder count and multiplying the result with the duration of the control loop sample period.

- *Steering Angle* – Steering angle means MARVIN's wheels will be turning in the same direction but at different speeds. The normalization is applied to the received nominal values before it is sent to the Plant. The normalization is achieved by converting the given angle to the encoder counts of the left and right wheels appropriately.

6.2.2 Use case "Autonomous Mode"

The functionality of the manual mode is inherited by ***autonomous mode use case***. Following the environment map upload, User Interface can set a start position and a target position. Navigation System will take over after receiving the start message for conducting a particular path in a given map.

- *Start Autonomous Mode* – The User Interface sends a message (which contains the mapping data) to start the autonomous mode, to the Navigation System. After successfully analysing the mapping data, Navigation System applies the use case Tracking Path, manoeuvring MARVIN along a defined path. If the mapping data is not valid, it will fall back to the manual mode.
- *Abort Autonomous Mode* – User Interface sends an abort message to exit the autonomous mode to the Navigation System, which instantly switches to manual control.
- *Upload Map* – User Interface reads a specified environment map file and an occupancy grid map is generated. This grid map is then transmitted to Navigation System for further analysis.
- *Get Fusion Data* – Fuses the correction data from IR sensors (measuring wall distances) with the odometry data.
- *Get Current Position* – Navigation System determines the current position and orientation after applying the use case Get Fusion Data.
- *Set Initial Position* – User Interface sends nominal values to set the initial position of MARVIN on the map.
- *Set Target Position* – User Interface sends nominal values to set the target position on the map which MARVIN should reach.
- *Path Plan* – Following the analysis of the grid map data, Navigation System plans the path of the robot using a modified A* algorithm.

- *Reactive Sequence* – This is the reactive control stage where a modified dynamic window with a polar histogram technique is implemented. This provides a target heading and a velocity which are used to instruct the robot to reach the goal.

6.3 Environment Map

The environment map which is constructed from recorded measurements of the third floor of LABY building's corridors forms the basis for MARVIN's navigation environment. It is measured at 1.5 × 11.4 metres. The environment is represented by a rectangular occupancy grid, generated by dividing the environment into discrete cells. As the dimensions of MARVIN's base are 0.58 × 0.52 metres, initially the size of the grids were set to 0.6 × 0.6 metres giving a resolution of 2 × 19 grids. This allowed MARVIN to be represented on the grid map by a minimum of one grid. However, the resolution was insufficient resulting in a poor representation of the environment. Therefore, it was decided to lower the grid size to 0.3 × 0.3 metres which resembles a resolution of 5 × 38 grids and representing MARVIN by a minimum of four grids.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				
10																				
11																				
12																				
13																				
14																				
15																				
16	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 6.3 Segment of the environment map implemented in the navigation system

As part of the navigation system, this map was converted into a two dimensional array (List of Lists in C#) of size 5 × 38 with a "1" depicting a wall and a "0" representing an unoccupied space. Figure 6.3 illustrates a segment of the map implemented in the navigation system. The walls are indicated by the thick lines. The grey region on the figure shows the environment as perceived by MARVIN.

6.4 Navigation System

As mentioned earlier (and shown by Figure 6.1), the navigation system consists of three main components, the driver layer, the integration layer and the presentation layer. The top two layers present the core functionality of the navigation system while the presentation layer is for the purposes of diagnosis and testing. The driver layer functionality is specific to the robotic platform the navigation system is designed to run on where as the integration and the presentation layers are common to all platforms.

6.4.1 Driver Layer

This section describes the abstract layer that represents MARVIN's core drivers. To navigate successfully, MARVIN's actuators and sensors had to be driven independently. This was achieved by introducing a set of services to send control commands and receive responses to and from each of the hardware modules.

Two main services were defined to manipulate the actuators and sensors of MARVIN,

- Rhino Drive service – For controlling the Rhino drive hardware module.
- Sensor Network service – For controlling MARVIN's 360° sensor network.

Because the hardware modules are operated by means of an identical communication medium, i.e. USB, the services controlling them have overlapping data transmission functionality. In other words, USB communication operations such as opening and closing of a serial port and transmission of data are identical in all the driver services. To avoid this unnecessary code replication, it was decided to introduce a third service called a Generic Module service.

6.4.1.1 Generic Module service

Generic Module service forms the basis for the above mentioned services. The purpose of this service is merely to communicate with hardware without any dependency on the commands defined in MARVIN's communication protocol. The concept of code reuse is adopted to create an internal abstraction of the USB communication operations. In addition to the basic operations, this service

will also perform simple data integrity checks to ensure the data transmitted to and from MARVIN is valid.

As illustrated in the Figure 6.4, each of the core services, Rhino Drive and Sensor Network, can partner with an instance of the Generic Module service to complete the interface between the hardware and the navigation system. The abstraction is achieved when the Generic Module service hides away the serial communication with the hardware modules, from the core services.

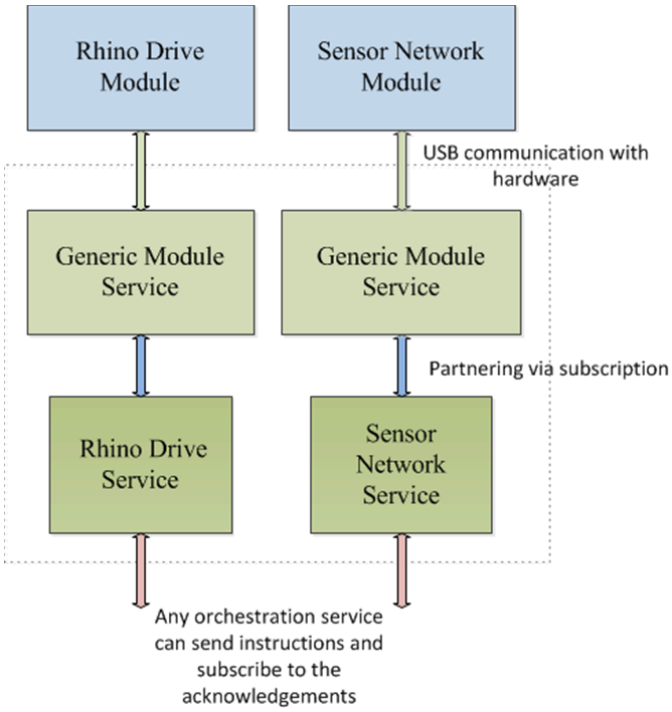

Figure 6.4 Overview of Generic Module service partnering with the core services

The control computer accesses each hardware module board over a USB serial connection which appears on the Windows OS as a virtual serial port. Generic Module service utilizes the .NET framework's *SerialPort* class to access these serial ports to provide serial communication.

Generic Module service is built on top of *GenericModuleService* class and several other C# classes that provide encapsulated functionality of MRDS and additional facilities to ensure that the serial communication meets the specification of the command interface. These classes and their dependencies are shown in Figure 6.5. *GenericModuleCoreOperations* and *GenericModuleCoreState* classes define the state, contract and operations of Generic Module service. *PacketProcessor*, *Packet* and *SerialManager* classes provide all the serial port functionality required to create, transmit and receive packets over the virtual serial port. *SerialManager* class

extends the *SerialPort* class functionality through multiple helper methods. It also uses the *PacketStore* class to queue the incoming response packets from the hardware modules so they can be processed in sequence.

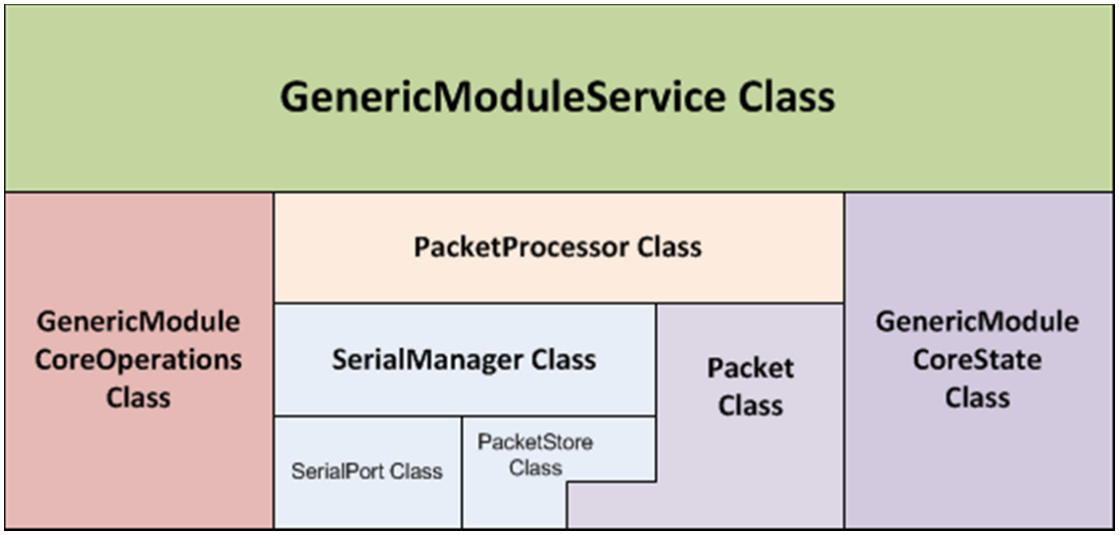

Figure 6.5 Overview of the Generic Module service class structure

Packet class

The *Packet* class defines basic methods to create packets that meet the specification of the command interface in each of the hardware modules. This class has static members to simplify the process of creating Dummy and Reset packets. It also has functions to determine the packet checksum and to write packets to a serial port.

The packet interface must be able to handle two types of packets, request and response. A request packet is a generated packet to be sent to the hardware modules while a response packet is a packet that is sent by the hardware modules to the software interface in reply. Although these packets have an identical format, they are processed differently by the interface. For example, when a new request packet is created, the checksum field is automatically computed and therefore it is used as a read-only field. In contrast, the response serial data received from the hardware module is used to construct a new packet and therefore, the checksum field is set as a writable attribute.

The table below shows the fields of the *Packet* class. This applies to both request and response packets. The fields shown have private attributes and therefore are not accessible externally. To make these fields publicly available, the class declares accessor methods. This concept is followed throughout other classes to standardize the code.

Table 6.1 Listing of request/response packet fields

Packet Field	Data Type	Description
_packetID	byte	This is the packet ID.
_packetByteCount	byte	This is a read-only field to keep the packet size. This field includes the 4 head bytes (packet ID, packet size, error code and checksum) and the payload size.
_errorCode	byte	This consists of a transmission error code. For request packets, the error code is always set to 0x01 as the error code for responses is set by the hardware modules.
_dataPayload	byte array	This is a read-only field for request packets and a read/write field for response packets. For request packets the error code is always set to 0x01 as the error code field is only utilised in response packets.
_checkSum	byte	Contains the checksum value.

The *Packet* class provides several functions to generate packets and help with the transmission of the data in the packets. It also defines several *Packet* constructors.

public Packet(byte[] databuffer)

Figure 6.6 illustrates one of the constructors which creates a new *Packet* object.

```
public Packet(byte[] databuffer)
{
    // Create a shallow copy
    _dataPayload = (byte[])databuffer.Clone();

    _packetID = 0x00;
    _errorCode = 0x01;
    _packetByteCount = (byte)(_dataPayload.Length + 4); // + 4 for the main
bytes
    _checkSum = CalcCheckSum();
}
```

Figure 6.6 Constructor for Packet object

It takes data payload as a parameter. A shallow copy of the data payload is created to avoid any changes to the original payload array. The `packet ID` and `error code` is set to the default values of 0x00 and 0x01 respectively.

public Packet(byte packetid, byte errorcode, byte[] databuffer)

This is the second constructor (Figure 6.7) which accepts three parameters, `packet ID`, `error code`, and `data buffer`. This is very similar to the previous constructor with the exception of setting the `packet ID` and `error code` to the given parameters.

```
public Packet(byte packetid, byte errorcode, byte[] databuffer)
{
    // Create a shallow copy
    _dataPayload = (byte[])databuffer.Clone();

    // Initialize the local variables
    _packetID = packetid;
    _errorCode = errorcode;
    _packetByteCount = (byte)(databuffer.Length + 4);
    _checkSum = CalcCheckSum();
}
```

Figure 6.7 Constructor of Packet object accepting three parameters

private static Packet Generate(byte packetid, byte errorcode, byte[] buffer)

This is a static helper function to create an instance of the *Packet* object from the given parameters. This is useful in instances where a constructor is not needed.

private byte CalcCheckSum()

This function described by Figure 6.8, calculates the checksum of the packet using a very simple algorithm of adding together the `packet ID`, `error code`, `packet size` and the `data payload`. The final sum is then bitwise inverted (XOR'ed) with 0xFF hex value. This is used to validate the contents of the packet.

```
private byte CalcCheckSum()
{
    // Checksum calculated as sum of all bytes XORed with 0xFF.
    byte calculatedCheckSum = 0;
    calculatedCheckSum += _packetID;
    calculatedCheckSum += _errorCode;
    calculatedCheckSum += _packetByteCount;
    for (int i = 0; i < _dataPayload.Length; i++)
    {
        calculatedCheckSum += _dataPayload[i];
    }
    calculatedCheckSum = (byte)(calculatedCheckSum ^
(byte)(0xff));

    return (byte)calculatedCheckSum;
}
```

Figure 6.8 Checksum function

protected WriteComPort()

This function of Figure 6.9 writes a single byte of data to the serial port using the *SerialPort* object's write function.

```
protected void WriteComPort(SerialPort port, int data)
{
    byte[] buffer = new byte[1];
    buffer[0] = Convert.ToByte(data);
    port.Write(buffer, 0, 1);
}
```

Figure 6.9 Function to write byte of data to the serial port

PacketStore class

PacketStore class has been implemented to process and store response packets from hardware modules in a queue (a FIFO data structure). It uses the *Packet* class to create the packets. If there is a steady stream of packets through the serial port, *PacketStore* can queue these packets without loss of any data. *SerialManager* class instantiates a *PacketStore* object and uses it to read the received packets.

public PacketStore()

This has no user defined constructor. The default constructor is generated by the compiler.

PacketStore class defines the following main functions to achieve this serial data queuing.

public AddPacket(byte[] data, int length)

This is one of the main functions of this class. Given an array of data and the length, it reads the array and identifies the packet fields. It then creates the packet, calculates the checksum and validates against the checksum of the received packet. If the validation result is a failure, the error code of the packet will be set to 0x04. Regardless of the validity of the packet, it will be queued. This process is repeated for any subsequent data in the given array.

public Packet RemovePacket()

This method helps with dequeuing of the packets starting with the first in the queue. Once the packet is read from the queue it is removed automatically.

SerialManager class

The *SerialManager* class is responsible for creating and closing a serial port, as well as the ability to send and receive raw packets. As mentioned earlier, *SerialManager* class utilises the .NET *SerialPort* class for serial port communication. In addition, it also utilises the convenience class *PacketStore* which processes, validates and queues the received serial data into packets.

SerialManager class has several public and private functions that extend the functionality of the *SerialPort* class.

public SerialManager()

This constructor (Figure 6.10) is called whenever a new *SerialManager* is instantiated and it creates a new *PacketStore* object and calls the CreatePort function to create a *SerialPort* object. User specified serial port properties are passed onto the CreatePort function to create the specified serial port. It also defines arbiters that execute handlers containing serial port operations, send, receive and close.

```
public SerialManager(DispatcherQueue dispatcherQueue, string portName, int baudRate,
int readTimeout, int writeTimeout)
            : base(dispatcherQueue)
{
    _builder = new PacketStore();
    CreatePort(portName, baudRate, readTimeout, writeTimeout);

    Activate(Arbiter.Interleave(
        new TeardownReceiverGroup(
            Arbiter.Receive<Close>(false, Operations, CloseHandler)
        ),
        new ExclusiveReceiverGroup(
            Arbiter.Receive<SendPKT>(true, Operations, SendPKTHandler),
            Arbiter.Receive<ReceivePKT>(true, Operations, ReceivePKTHandler)
        ),
        new ConcurrentReceiverGroup()
    ));
}
```

Figure 6.10 Constructor for SerialManager object

private CreatePort()

Initially, this function creates the *SerialPort* object, and then configures it, followed by calling the open function of *SerialPort* object to open the serial port. This also sets read and write timeouts to stop the Generic Module service from hanging indefinitely if the serial communication link fails.

private sp_DataErrorHandler()

This is a helper function which gets called when an error event is triggered in the *SerialPort* object. An error message is then posted and notified to any listening arbiters.

private sp_DataReceived()

When there is data waiting to be read by the *SerialPort* object, this function gets triggered and posts a `ReceivePKT` message to the Operations port which is then received by the respective arbiter.

Classes for three message types are defined to provide serial functions; send, receive and serial port close. These are `ReceivePKT`, `SendPKT` and `Close` respectively. A set of corresponding handlers (given below) are defined to process these message types when they are posted on the Operations port. *SerialManager* class also defines an additional message type for error handling, Error. Any class instance listening for this message type can define its own version of the handler to process the errors appropriately hence a handler is not defined in the *SerialManager* class.

```
void ReceivePKTHandler(ReceivePKT recv)
{
    try
    {
        if (_serialPort.BytesToRead <= 0)
        {
            return;
        }
        byte[] data = new byte[_serialPort.BytesToRead];

        int read = _serialPort.Read(data, 0, data.Length);

        if (read <= 0)
        {
            return;
        }

        // add the received packet to the processing queue
        _builder.AddPacket(data, read);
        // process the queue
        while (_builder.HasPacket)
        {
            // read packet and remove from queue
            Response.Post(_builder.RemovePacket());
        }
    }
    catch (Exception e)
    {
        recv.ResponsePort.Post(new Exception(e.Message));
        Response.Post(new Error(e));
    }
}
```

Figure 6.11 The content of the ReceivePKTHandler handler method

private ReceivePKTHandler()

As shown in Figure 6.11, this handler reads the raw serial data and creates the packets using the *PacketStore* object. It then checks for any valid packets in the queue and forwards the packets to any listeners. Any errors will be notified via the Error message type.

private SendPKTHandler()

SendPKTHandler simply uses the Write function of the *Packet* object to transmit the raw serial data. If an error occurs in the process it will generate an Error message type and will be notified to any listeners.

private CloseHandler()

This is responsible for closing the port.

PacketProcessor class

PacketProcessor class has been implemented as a state machine to carry out the task of passing messages between other services and the *SerialManager* class. *PacketProcessor* accomplishes this task by defining several message types as classes and corresponding receivers. Setting up these receivers is the job of an interleave, thus *PacketProcessor* class also defines several interleaves.

The following functions are defined in the class,

private PacketProcessor()

This is the *PacketProcessor* constructor. The main job of this is to activate and start the *InitializeTask* interleave.

private OpenSerialManager(int portnumber, int baudrate, int readtimeout, int writetimeout)

This function is defined to instantiate *SerialManager* object. It then proceeds to send a Dummy packet to the hardware modules to check the connection status.

private SendDummy()

This posts a Dummy packet via the serial port and activates *WaitForDummy* interleave.

private ResetSenderTask()

This function is defined to reset the *SenderTask*. It uses CCR's *EmptyValue* object to signal the listening *SenderTask* interleave to exit from any current activities it is performing.

private OpenHandler(Open open)

This processes the internally defined `Open` message to activate both *SenderTask* and *PacketListenerTask*.

private CloseHandler(Close close)

This processes the internally defined `Close` message to reset the *SenderTask* interleave by calling the ResetSenderTask function and then it activates *ConnectTask* interleave.

private SendingPacketHandler(Send send)

This function is responsible for sending the packets by posting a `Packet` message to the *SenderTask* interleave.

private ReceivingPacketHandler(Packet packet)

Following the receipt of a response packet from the hardware modules, this function forwards the packet to *GenericModuleService* class.

private SerialManagerErrorHandler(SerialManager.Error error)

As the name suggests, this function is responsible for handling the exceptions being thrown by the *SerialManager* class. These exceptions are forwarded to the *GenericModuleService* class.

PacketProcessor sets up several interleaves to assist with the flow of messages. Figure 6.12 shows a flowchart of the flow of messages and how the interleaves interact. Following is a list of all the interleaves and their functionalities.

- *InitializeTask* – The start of the state machine. It defines two receivers for `PortConfig` and `Reset` messages. When a `PortConfig` message from *GenericModuleService* is received, OpenSerialManager instantiates the *SerialManager* object. When a `Reset` message is received, the ResetSenderTask method is executed and the *InitializeTask* interleave is reactivated.

- *ConnectTask* – Accepts only `Open` and `Reset` messages. Similar to *InitializeTask* interleave, this interleave calls ResetSenderTask function and reactivates *InitializeTask* interleave. `Open` message causes the execution of the OpenHandler function.
- *SenderTask* – This interleave is responsible for transmitting the packets through the serial port to the hardware modules. It accepts `Packet` message for sending the packets, `EmptyValue` and `DateTime` message types. `EmptyValue` signals when this interleave should stop its current actions and reset. `DateTime` message types triggers a timeout and causes a reactivation of this interleave.
- *PacketListenerTask* – Accepts 5 message types, `Close`, `Reset`, `Send`, `Packet` and `SerialManager.Error`. `Close` message type resets the *SenderTask* interleave and reactivates *ConnectTask* interleave. `Reset` message type behaves similar to other interleaves. `Send` message type passes the request packet to the *SenderTask*. When a response is received a `Packet` message type executes ReceivingPacketHandler to process the packet. `SerialManager.Error` message type calls the SerialManagerErrorHandler function to process any *SerialManager* errors.
- *WaitForDummy* – This interleave waits for the dummy response packet to arrive (response to the dummy packet sent by OpenSerialManager method) from the hardware modules. It also accepts `Reset` message for resetting *SenderTask* and `DateTime` message for a timeout. `DateTime` message also causes a reactivation of the *InitializeTask*.

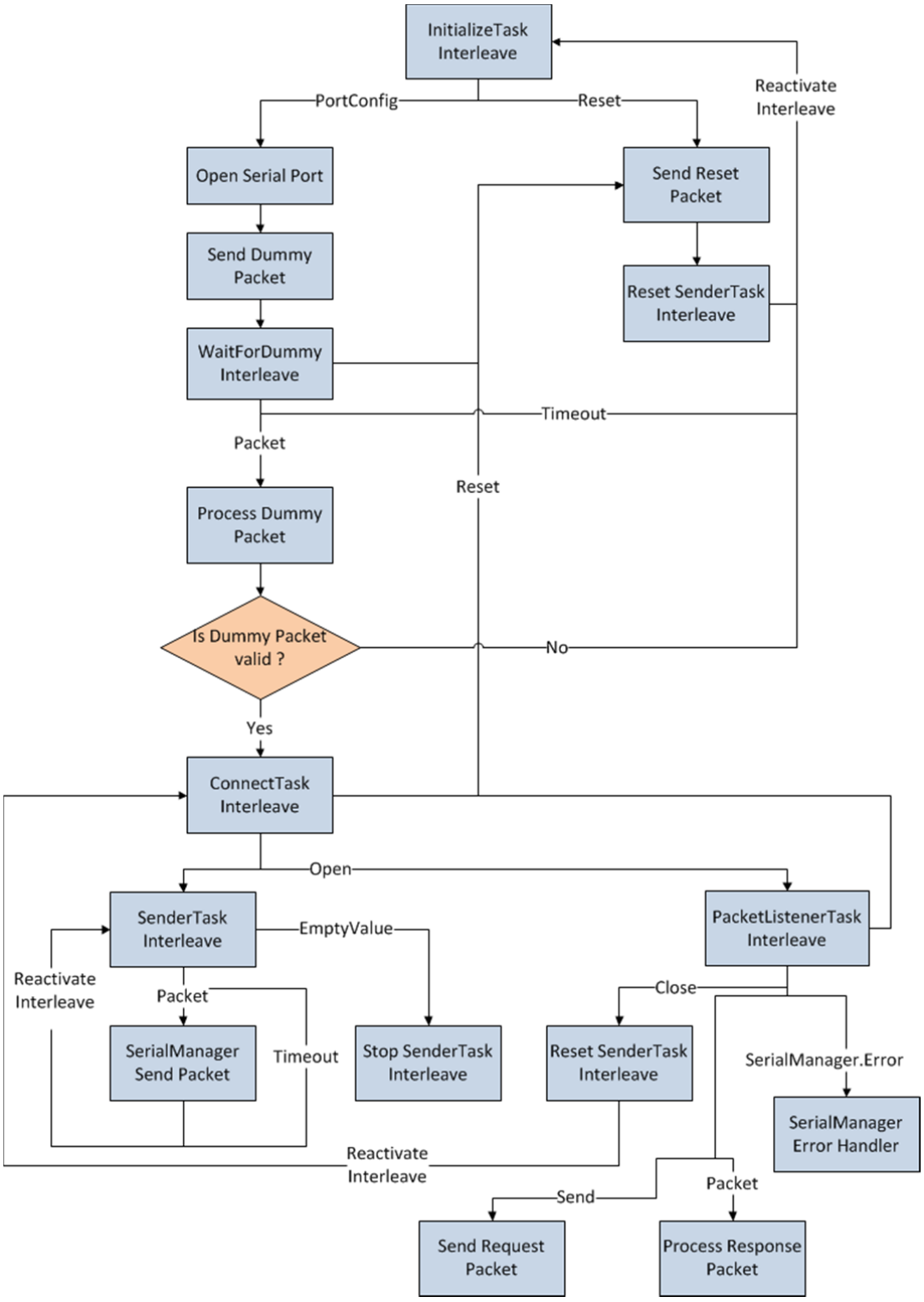

Figure 6.12 Flowchart of the messages within PacketProcessor class

GenericModuleCoreState class

State is one of the essential components of a service class. Any information retrieved from a service must be part of the state hence the state should contain all the necessary information to enable it to be saved. This class defines the state and the information that the service will expose.

This class defines the following fields,

bool openOnStart

When this flag is set to true, Generic Module service will automatically send a `PortConfig` message when the service is started.

SerialPortConfig config

A SerialPortConfig object is created to keep the current configuration of the serial port. This object consists of the port number, baud rate, read and write timeouts.

bool portStatus

This field stores the status of the serial port. Initially it is set to true.

ReceivedPacket receivedPacket

This field stores the last received response packet.

GenericModuleCoreOperations class

This class defines the service contract and the service operations for the main port. The service operations include operations from DSSP service model and custom written operations. Service operations allow other services to interact with this service over the main port. The *GenericModuleCoreOperations* constructor illustrated in Figure 6.13 lists the following operations;

- *DsspDefaultLookup* – Enable other services to find out information about this service.
- *DsspDefaultDrop* – Used to shutdown the service.
- *Get* – Retrieve a copy of the GenericModule's state.
- *Replace* – Operation to replace the entire state.
- *Subscribe* – Other services can request notification of all state changes by subscribing to this service.

- *ReliableSubscribe* – Similar to the subscribe operation, however this service will stop sending notifications if the receiving end becomes unreachable.
- *OpenSP* – Operation sends an `Open` message to *PacketProcessor* object.
- *CloseSP* – Sends a `Close` message to *PacketProcessor* object.
- *UpdateSPConfig* – Allow other services to update the serial port configuration.
- *ProcessCommand* – An internal operation to send packets sequentially. Because the service is run in a multi-threaded environment, this operation will ensure that the packets are transmitted safely through the serial port.
- *PortStatusChanged* – This operation updates the port status in the state, `portStatus`.

```
public GenericModuleCoreOperations()
    : base(
    typeof(DsspDefaultLookup),
    typeof(DsspDefaultDrop),
    typeof(Get),                    // Get entire state
    typeof(Replace),                // Replace state
    typeof(Subscribe),              // Services can subscribe for any changes
    typeof(ReliableSubscribe),      // Reliable subscribe
    typeof(OpenSP),                 // Open service port/manager
    typeof(CloseSP),                // Close service port/manager
    typeof(UpdateSPConfig),         // Configuration update for service port
    typeof(ProcessCommand),         // Send commands
    typeof(PortStatusChanged)
    )
{
}
```

Figure 6.13 GenericModuleCoreOperations constructor listing the operations

GenericModuleService class

This is the main class of the Generic Module service. Initially, it creates an instance of *GenericModuleCoreState* and two instances of *GenericModuleCoreOperations*. The reason for creating two different instances is due to the fact that a service can operate on internal and external operations. As described earlier, the *GenericModuleService* class has a defined *ProcessCommand* operation which operates internally within the Generic Module service. Conversely, other operations are accessed externally by other services. The Generic Module service defines the subscription manager which enables other services to subscribe for notification messages. Figure 6.14 illustrates these key components of the service.

```
/// <summary>
/// Service state
/// </summary>
[InitialStatePartner(Optional = false, ServiceUri = "MarvinGenericModule.Config.xml")]
GenericModuleCoreState _state = null;

/// <summary>
/// Main service operations port
/// </summary>
[ServicePort("/MarvinGenericModule", AllowMultipleInstances = false)]
GenericModuleCoreOperations _mainPort = new GenericModuleCoreOperations();

/// <summary>
/// Pending port to direct serial packets (messages)
/// </summary>
GenericModuleCoreOperations _pendingPort = new GenericModuleCoreOperations();

/// <summary>
/// Native subscription manager
/// </summary>
[Partner(Partners.SubscriptionManagerString, Contract = submgr.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.CreateAlways)]
submgr.SubscriptionManagerPort _subMgrPort = new submgr.SubscriptionManagerPort();

/// <summary>
/// Service constructor
/// </summary>
public GenericModuleService(DsspServiceCreationPort creationPort)
       : base(creationPort)
{
}
```

Figure 6.14 Instantiation of state and operation port of Generic Module service

It next defines the Start method which gets called during the initialization of the service. Start method initializes the state and instantiates the *PacketProcessor* object. When a new service class is generated in the IDE, MRDS by default includes base.Start() method which sets up the handlers on the main port and adds the service to the service directory. The start method also implements several interleaves as shown in Figure 6.15. Finally the serial port connection process is started with the function ConnectModule.

The main functions defined in *GenericModuleService* class are listed below.

private InitializeState()

Generic Module service state is persisted in a configuration file called *MarvinGenericModule.Config.xml*. When the service is loaded for the first time this file is empty hence this function is used to instantiate and populate the state object. SaveState, a built-in function performs the task of persisting the state.

private ConnectModule()

Executed by the Start method (as shown in Figure 6.15), this method posts a `PortConfig` message and waits for the result. If the message is posted successfully this method waits for 0.5 seconds, giving time to send request packets and receive response packets to and from the hardware modules respectively, before it posts an `Open` message to the *PacketProcessor* object. If a failure occurs on either of the stages, the ConnectModule is executed again after 1 second.

private InternalCommandHandler()

This function represents an internal handler defined to process the ProcessCommand messages sequentially. Essentially this function queues the incoming messages and enables the service to process each message one after another. Each message includes the serial packet for transmission to the underlying hardware. As shown in Figure 6.15, this is executed by the receiver on the duplicate *GenericModuleCoreOperations* port. It generates a new *Packet* object from packet ID and the pay load and then sends the packet to *PacketProcessor* for further processing.

private PacketReceiveHandler()

When a ReceivedPacket message is posted by *PacketProcessor* object, this handler performs the task of updating the state field, `receivedPacket`, and notifies any subscribers of the new state information.

```
/// <summary>
/// Service start
/// </summary>
protected override void Start()
{
    // Obviously we are initializing the state here
    InitializeState();

    // Setup the packet processor
    _packetproc = new PacketProcessor(TaskQueue, _receivePort, _portStatusChange);

    // Add ourselves to the service directory
    base.Start();

    // Listen on the ports for requests and call the appropriate handler.
    base.MainPortInterleave.CombineWith(
        new Interleave(
            new ExclusiveReceiverGroup
            (
                Arbiter.Receive<ReceivedPacket>(true, _receivePort,
PacketReceiveHandler),
                Arbiter.Receive<Exception>(true, _receivePort, ExceptionHandler),
                Arbiter.Receive<bool>(true, _portStatusChange, PortStatusChangeHandler)
            ),
            new ConcurrentReceiverGroup()
        )
    );

    // Messages are serialized here in the sense that they will be queued for processing
    Activate(
        Arbiter.Interleave(
             new TeardownReceiverGroup(),
             new ExclusiveReceiverGroup
             (
                 Arbiter.ReceiveWithIterator<ProcessCommand>(true, _pendingPort,
InternalCommandHandler)
             ),
             new ConcurrentReceiverGroup()
        )
    );

    // Setup Serial Manager and start the interleaves
    if (_state.OpenOnStart)
    {
        ConnectModule();
    }

    LogInfo("Starting Marvin Generic Module service");
}
```

Figure 6.15 Start method of generic module service

6.4.1.2 Rhino Drive service

Rhino Drive service is one of the driver level core services which controls MARVIN's differential drive control system. It is built on top of MRDS's Generic Differential Drive (GDD) service. GDD provides a common specification and all robots that conform to this specification can be controlled

by applications that ship with MRDS such as Simple Dashboard. Rhino Drive service also wraps around Generic Module service by subscribing to it.

Rhino Drive service is defined by three classes: the *RhinoDrive* class as the main class, *RhinoDriveState* class defines the state, and *RhinoDriveOperations* class defines the operation types.

RhinoDriveState class

RhinoDriveState class defines the state of the Rhino Drive service. This consists of fields representing the features of the differential drive system of MARVIN. The main fields of the state are:

bool Enabled

This enables or disables the GDD service.

double ControlLoopTime

This represents the frequency of the control loop in seconds.

int RighVel

This holds the velocity of the right wheel in metres per second.

int LeftVel

This holds the velocity of the left wheel in metres per second.

int RightEnc

This field represents the last generated encoder ticks in the right side.

int LeftEnc

This field represents the last generated encoder ticks in the left side.

int RightWheelDistance

This field represents the total distance travelled by the right wheel from the start in metres.

int LeftWheelDistance

This field represents the total distance travelled by the left wheel from the start in metres.

WheelConfiguration RightWheel

WheelConfiguration class has been implemented to store the characteristics of the wheel such as motor gear ratio and wheel radius. This field stores the properties of the right side wheel.

WheelConfiguration LeftWheel

This field stores the properties of the left side wheel.

RhinoDriveOperations class

Rhino Service defines a set of operations enabling external services to access the differential drive system of the robot. These operations allow other services, which are not using the GDD service, to control the differential drive. The key operations are detailed below.

- *EnableDrive* – This operation allows the activation and deactivation of GDD.
- *Stop* – This is defined for emergency stops which results in the robot stopping immediately.
- *SetVelocity* – This sets the velocity of the robot and causes it to drive straight.
- *SetRotation* – This is to enable the rotation of the robot in an arc.

RhinoDriveService class

The core functionality of the Rhino Drive service is defined in this class.

As shown in Figure 6.16 initially the Rhino Drive service state and GDD state are instantiated followed by the operation ports of Rhino Drive and GDD. Most importantly the subscription managers are instantiated at the end to allow other services to subscribe to the Rhino Drive service. This class defines handlers for the GDD operations and the operations stated in RhinoDriveOperations class. GDD operations allow services conforming to the GDD specification to send messages to the Rhino Drive service.

The navigation system primarily uses *SetVelocity* and *SetRotation* operations to move straight and to turn in an arc respectively. The navigation system calculates the velocities in metres per second, however the low level hardware requires the velocities in encoder ticks. Therefore, these velocities are normalized before they are transmitted as commands to the differential drive hardware. The velocities are normalized by converting from metres per second to encoder counts per second for

the left and right wheels. Similarly, *SetRotation* measurements are given in radians per second and this is converted to linear velocity before the velocities of the right and left wheels are sent as request commands.

```
/// <summary>
/// Service state
/// </summary>
[InitialStatePartner(Optional = true, ServiceUri = "RhinoDrive.Config.xml")]
private RhinoDriveState _state = new RhinoDriveState();

/// <summary>
/// Generic Differntial Drive state instance
/// </summary>
private gddrive.DriveDifferentialTwoWheelState _gddState = new
gddrive.DriveDifferentialTwoWheelState();

/// <summary>
/// Main service port
/// </summary>
[ServicePort("/MarvinRhinoDrive", AllowMultipleInstances = false)]
private RhinoDriveOperations _mainPort = new RhinoDriveOperations();

/// <summary>
/// GDD (MRS Generic Differential Drive) port
/// </summary>
[AlternateServicePort("/Drive", AllowMultipleInstances = false, AlternateContract =
gddrive.Contract.Identifier)]
private gddrive.DriveOperations _gddDrivePort = new gddrive.DriveOperations();

/// <summary>
/// Subscription manager partner for the native notifications
/// </summary>
[Partner(Partners.SubscriptionManagerString, Contract = submgr.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.CreateAlways)]
private submgr.SubscriptionManagerPort _subMgrPort = new
submgr.SubscriptionManagerPort();

/// <summary>
/// Subscription manager partner for GDD notifications
/// </summary>
 [Partner(Partners.SubscriptionManagerString + "/GenericDrive" ,
            Contract = submgr.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.CreateAlways)]
private submgr.SubscriptionManagerPort _gddSubMgrPort = new
submgr.SubscriptionManagerPort();
```

Figure 6.16 Data members of the Rhino Drive service

Each of the response packets from the hardware is sent as a `Replace` message by the Generic Module service. This happens as a result of the subscription to the Generic Module service. The *Replace* handler reads the content of each of the responses however, it only processes the *DriveStatus* packet. *DriveStatus* packet includes the information such as encoder counts and flags to indicate if the robot is moving. The *Replace* handler also performs the task of calculating the velocity information from the received encoder counts.

Figure 6.17 illustrates the implementation of the handler for *SetVelocity* operation. This handler first normalizes the given velocity into the respective wheel velocities and sends the right and left velocities to the low-level hardware as a command.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> SetVelocityHandler(SetVelocity velocity)
{
    // Normalize velocity
    int right = (int)(Math.Round((((velocity.Body.RightWheelSpeed * 1000) /
_state.RightWheel.WheelmmPerCount) * _state.ControlLoopTime)));
    int left = (int)(Math.Round((((velocity.Body.LeftWheelSpeed * 1000) /
_state.LeftWheel.WheelmmPerCount) * _state.ControlLoopTime)));

    Activate(
        Arbiter.Choice(
            SetVelocity(right, left),
            delegate(DefaultUpdateResponseType response)
            {
                velocity.ResponsePort.Post(DefaultSubmitResponseType.Instance);
            },
            delegate(Fault fault)
            {
                velocity.ResponsePort.Post(fault);
            })
    );

    yield break;
}
```

Figure 6.17 SetVelocity handler

6.4.1.3 Sensor Network service

Sensor Network service controls the rangefinder sensors on MARVIN. Unfortunately, there was no existing generic driver service on MRDS that provided a similar functionality hence this service had to be designed from ground up. This provides functionality to reset the sensor network and to gather measurement data. The main task of the service is to poll the sensor data at a given frequency.

Sensor Network service is defined by three classes: the *SensorNetwork* class as the main class, *SensorNetworkState* class defines the state, and *SensorNetworkOperations* class defines the operation types.

SensorNetworkState class

This class defines the state of the Sensor Network service. The main fields are listed below.

int updateFrenquency

This specifies the frequency at which the sensors should be polled. The default frequency is set to 100 milliseconds.

int numberOfConfiguredNodes

This field represents the total number of configurable sensors in the network.

int numberOfNetworkNodes

This field represents the total number of active sensors in the network.

List<Node> networkNodes

This field contains the up-to-date sensor information including the calibration data.

SensorNetworkOperations class

This class has the definitions for message operations. In addition to the built-in DSSP operations this class consists of the following main operations:

- *UpdateConfiguration* – This operation allows external services to update the `updateFrequency` field mentioned earlier.
- *UpdateFullNetworkDefinition* – During the initialization of the service, the `networkNodes` state field is instantiated and populated using this operation.
- *UpdateNetworkDefinition* – This operation allows the service to update the network definitions of the sensor nodes which are part of the `networkNodes` field.
- *UpdateSensorData* – This operation updates the distance measurement information.

SensorNetworkService class

Similar to the Rhino Drive service, the *SensorNetworkService* class creates instances of *SensorNetworkState* and *SensorNetworkOperations* classes. Operation ports for the partner services, Generic Module and Subscription Manager are then instantiated to enable other services to control the Sensor Network service. The sensor configuration data is kept in an XML file which provides sensor calibration information. To read this configuration file, this class implements a static constructor, shown in Figure 6.18. This is to allow the class to read the configuration data by decomposing the XML file prior to creating an instance of the class.

```
static SensorNetworkService()
{
    const string ParentTag = "SensorNetworkConfig";
    bool valid = XMLConfigReader.SetConfiguration("C:\\SensorNetworkConfiguration.xml",
ParentTag);

    // Check if the configuration XML file is valid
    if (valid)
    {
        // Get the Node total
        if (XMLConfigReader.GetAsInt("NodeTotal", ref NodeTotal) == false)
        {
            XMLConfigReader.HandleError("Failed to get " + ParentTag + "/NodeTotal from
config file");
        }

        // Get Node data and Calibration data
        string Node = string.Empty;
        for (int i = 0; i < NodeTotal; i++)
        {
            if (XMLConfigReader.GetAsString("NodeList/Node_" + (i + 1), ref Node) ==
true)
            {
                // Add the Node data
                Nodes.Add(Node);

                // Now lets try to find the Calibration ID
                string CaliData = string.Empty;
                string[] nodeData = Node.Split(new char[] { ',' });
                string CaliID = nodeData[nodeData.Length-1];
                if (XMLConfigReader.GetAsString("CaliData/" + CaliID, ref CaliData) ==
true)
                {
                    CalibrationData.Add(CaliID, CaliData);
                }
                else
                {
                    XMLConfigReader.HandleError("Failed to get " + ParentTag +
"/CaliData/" + CaliID + " from config file");
                }
            }
            else
            {
                XMLConfigReader.HandleError("Failed to get " + ParentTag +
"/NodeList/Node_" + (i + 1) + " from config file");
            }
        }
    }
}
```

Figure 6.18 Static constructor of the SensorNetworkService class

Once an instance of the class is created, the Start method is executed. The Start method initializes the state, sets the handlers for the operations, initializes the network nodes data structure for the storage of sensor data, and finally starts the polling timer. The polling timer constantly sends a command to the sensor network hardware module requesting the sensor information. When new sensor information is available the service will notify any subscribers.

As described earlier, the sensor information is stored in a List data structure. This allows for easy accessibility and automatic memory allocation. This data is part of the state thus will be presented to any subscribers immediately.

6.4.2 Integration Layer

This layer constitutes the Hybrid Navigation service, which implements all of the components of the

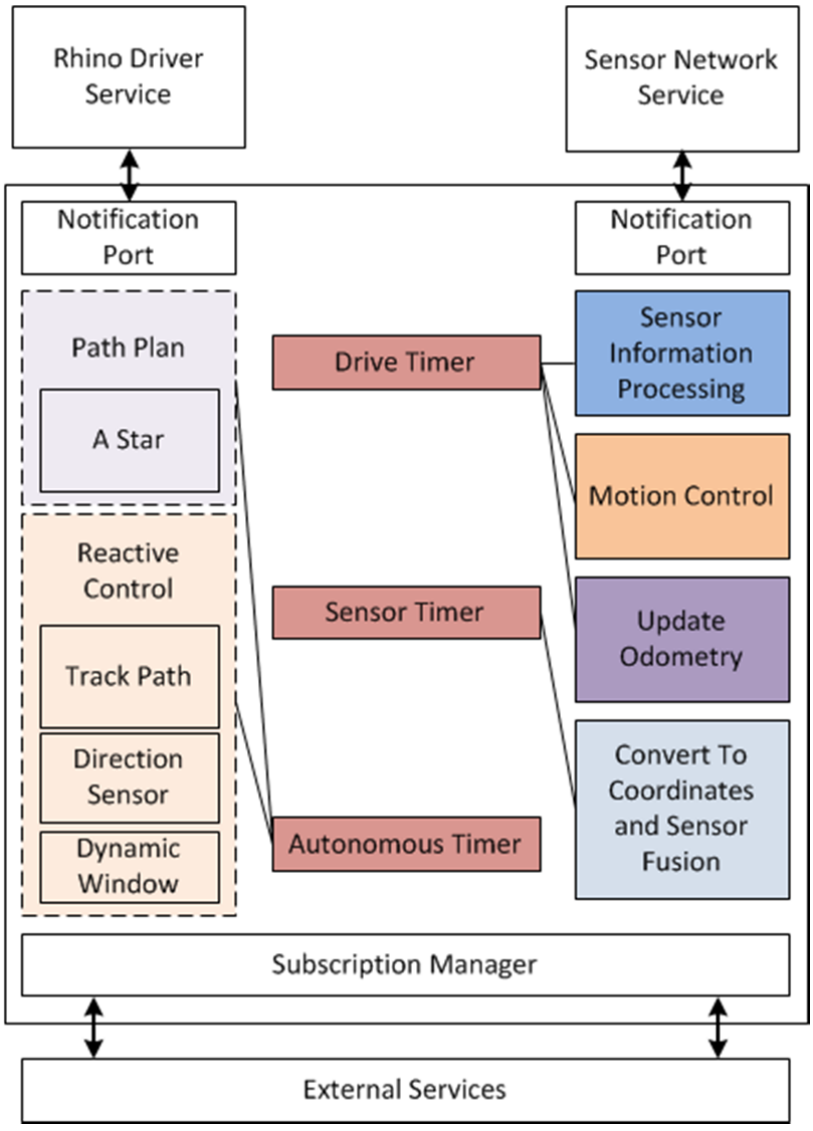

Figure 6.19 Components of the Navigation service

hybrid hierarchical navigation system described in chapter 4. Figure 6.19 illustrates the main components of the service. It should be noted that the coloured blocks shown in the figure represent functions defined within the service class.

Two control modes have been defined to activate and deactivate the navigation functionality. In manual mode, the navigation system is deactivated and in autonomous mode the system is activated.

The service partners with the Rhino Drive and Sensor Network services to provide perception and low level motion control. Each of these services has a defined notification port to listen to subscription messages.

The navigation service also consists of three timers to execute different tasks of the navigation system namely: Drive, Sensor and Autonomous.

Similar to other services, the integration layer is implemented as a service and it splits the functionality into three class files: the main code, the state and the operation types. Each of these components will be discussed further in the next few sections.

HybridNavigationState class

The state class defines the property fields of the navigation service. Several of the main fields are listed below.

Pose initPose

This field stores the initial pose of the robot. The initial pose is defined as x, y and theta.

Pose targetPose

This field provides the target pose for the robot.

Pose fusedPose

This holds the pose from the resulting fused sensor data. It determines where the robot is in a given coordinate space and it is used to drive the robot towards the given target.

List<List<double>> gridObsts

This multi-dimensional field stores the occupancy grid data of the environment map.

List<MapPointData> mapPointData
List<MapPointConnectivityData> mapPointConnectivityData

Both these fields store the points outlining the environment map and the relationship between these points.

PathPlan pathPlan

This field is an instance of the *PathPlan* object which stores the `pathPlanFlag` flag. Setting this flag starts the path plan process of the navigation system.

ReactiveControl reactiveControl

This stores the dynamic window parameters such as angular velocity limits (ω_{min} and ω_{max}) so that these can be set by an external service.

HybridNavigationOperations class

All the service operations of the Hybrid Navigation service are implemented in this class. These operations are intended to be used in conjunction with other services. In addition to the generic MRDS operations such as Get, Replace and Subscribe; the following operations are defined to support the main service class:

- *UpdatePathPlanCheck* – This operation updates the path plan flag.
- *UpdateGridResolution* – The occupancy grid map of the navigation system is generated when this operation is received. The message contains the properties for generating the map such as grid resolution and the number of grid cells.
- *UpdateMapPointData* – This operation sets the `mapPointData` and `mapPointConnectivityData` fields.
- *UpdateDriveMode* – This message sets the control mode to either manual or autonomous, thus enables the activation of the hybrid navigation system.
- *UpdateInitTargetPose* – This updates the initial and target poses defined in the state. After updating the state, it resets the navigation system to reflect the changes.
- *UpdateRCSettings* – Posting this operation type to the Hybrid Navigation service results in setting the `reactiveControl` field and updating the dynamic window parameters.

HybridNavigationService class

This is the main service class which implements the core functionality of the hybrid navigation system. Initially, it creates an instance of the service state (defined as *HybridNavigationState*) and the main operations port (defined as *HybridNavigationOperations*). It defines partnerships for both,

Rhino Drive service and Sensor Network service, to provide access to the sensor data and controlling the differential drive of MARVIN. A subscription manager is also defined to allow external services to listen for state changes.

As depicted in Figure 6.19, this class defines three timers which perform the critical tasks of the service. This is further described by Figure 6.20. A call to the start method, during the initialization of the service, triggers the three timers.

- Drive timer – This timer initially computes the odometry data and then periodically controls the motion of MARVIN by feeding the Rhino Drive service with the most current velocity and the angular velocity. The timer is executed at a frequency of 20 Hz.
- Sensor timer – At a frequency of 20 Hz, this timer constantly determines the position and orientation of the robot by translating the rangefinder data followed by fusing the odometry information.
- Autonomous timer – This timer is responsible for running the path planner and reactive control functionality. If enabled, the path planner is executed and the path planner status flag is set. Unless the path planner status flag indicates that the path planner has failed (path planner flag not equal to 2), the reactive control sequence is run which includes the path tracker, the direction sensor and the dynamic window. The timer is activated strictly in the autonomous mode and it executes these tasks at a frequency of 10 Hz.

The service class defines the corresponding handlers to the operation message types described in in the previous section. These can be posted on the main port of the service to update the state of the service. For the purposes of analysis and diagnosis, a logging function is implemented. This function constantly logs the vital information such as the current velocity of MARVIN.

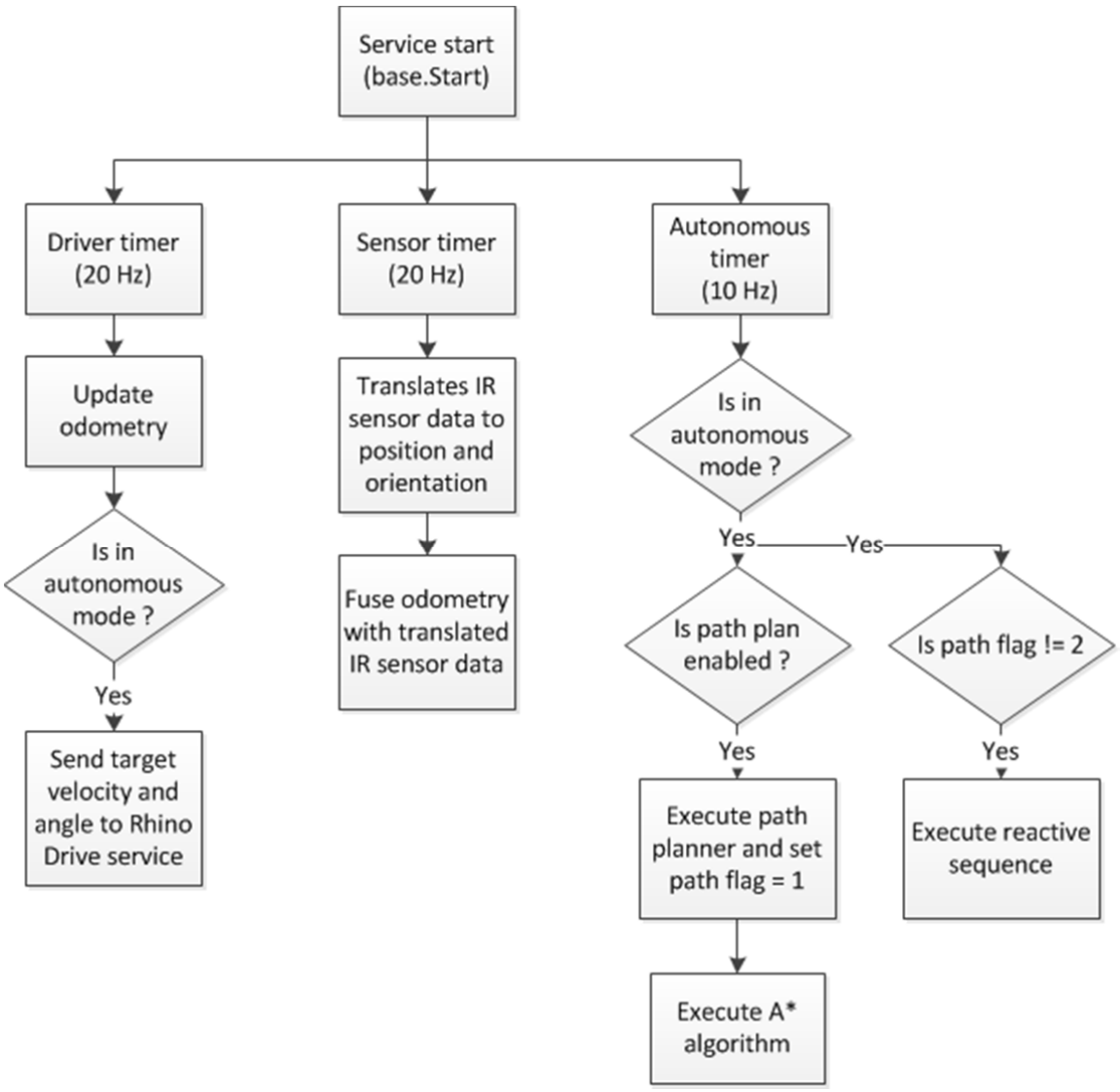

Figure 6.20 Timer tasks of the Hybrid Navigation service

6.5 User Interface

A graphical user interface is implemented in C# Windows Forms (WinForms) to complement the navigation system. This GUI has been designed to perform the following main tasks:

- Display sensor feedback from the 360° rangefinder network and encoders.
- Enabling the selection of either manual or autonomous control modes.
- Achieving manual control of MARVIN by utilizing a joystick interface.
- Uploading the environment map data to the navigation service. This reads two files which specify map points and the relationship between these points.
- Functionality to generate the occupancy grid map.

- Configuration of the dynamic window by posting *UpdateRCSettings* operation messages to the navigation service.

The interface provides three tabs (shown in Figures 6.21 – 6.22) to accomplish the above mentioned tasks. Figure 6.21 shows the main screen which displays the control mode selection (manual or autonomous), joystick control and display data from sensor feedback. The interface displays the sensor data as a map, with each sensor data value plotted as a vector (indicated in Figure 6.21 by the red lines). In the autonomous mode, a button is provided to start the navigation of the robot. A stop button is implemented to stop the robot immediately in an emergency.

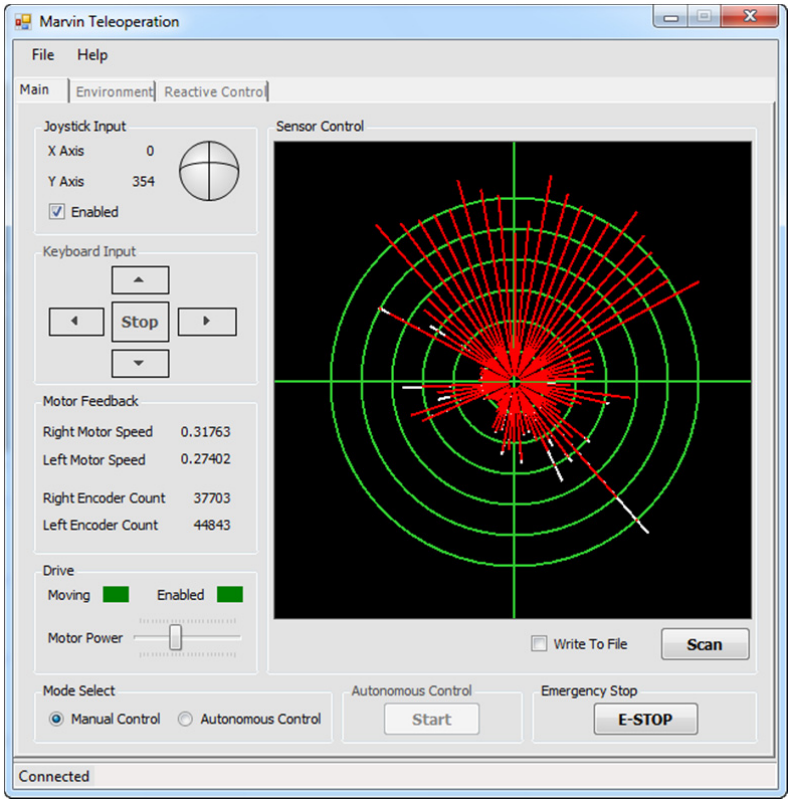

Figure 6.21 The main tab of the interface

The second tab is illustrated in Figure 6.23 which enables the grid map generation, activation of the path planner and displays the environment map. The environment map displays the environment boundary (shown by the purple straight lines), current position (shown by the red circle), and the target position of the robot (shown by the green circle).

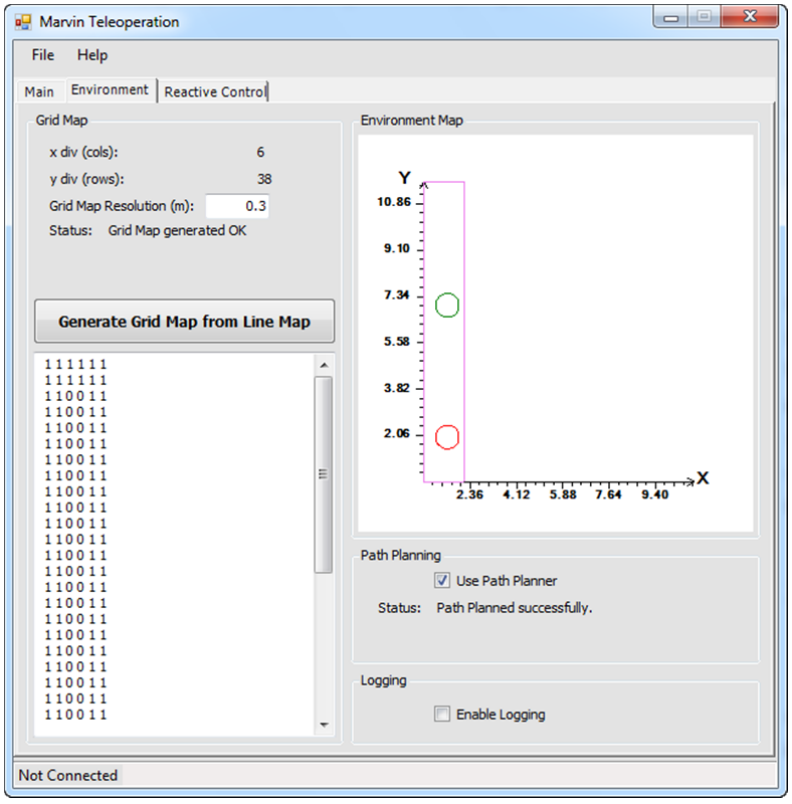

Figure 6.23 A screenshot of the environment tab of the interface

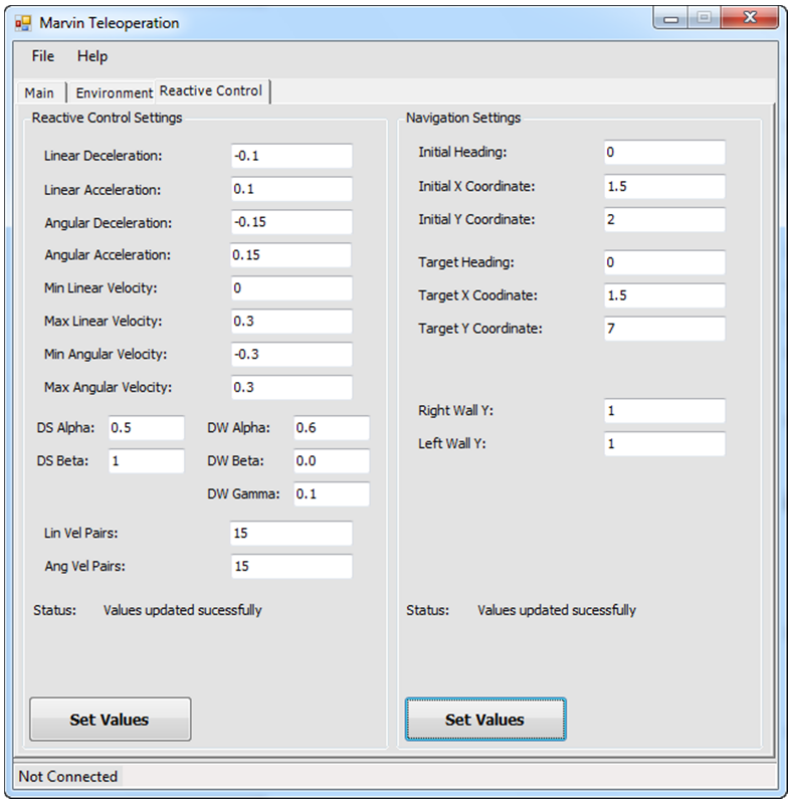

Figure 6.22 A screenshot of the reactive control tab

The last tab, as shown in Figure 6.22, facilitates the configuration of the dynamic window, the initial and the target positions of the robot. These tabs are inactive until a connection is established with the remote PC. If a failure occurs in the connection process, the interface will indicate that the connection is pending. Until a successful connection is made the tabs will remain inactive. The connection status is indicated at the bottom of the interface.

6.6 Summary

Following the SOA architecture, the hybrid navigation system has been developed as a software framework to provide autonomous navigation for MARVIN. At the early stages of the development process, a functional model was adopted which separates the functionality into two main layers, driver layer and integration layer. The driver layer consists of three services which include the Generic Module service, the Rhino Drive service and the Sensor Network service. Rhino Drive and Sensor Network services have implemented functionality to control the actuators and sensors respectively. Generic Module service allows the abstraction of the USB communication functionality with the hardware modules such as sending and receiving commands. The integration layer comprises of the Hybrid Navigation service which implements the components of the hybrid hierarchical navigation system. This service integrates the driver services to perform the tasks of low-level motion control and sensor information extraction. A user interface was introduced to enable manual or autonomous control of the robot. In the manual mode, the robot can be driven using either a joystick or a keyboard. When the mode is switched to autonomous, a user can specify the environment map, navigation settings, the initial and the target headings. This activates the hybrid navigation system allowing the robot to navigate a given environment autonomously.

Chapter 7 Results

7.1 Odometer Calibration

Initially, a decision was made to limit the digital potentiometer values sent to the RHINO H-bridge motor drivers so that the motors are not over-strained at high speeds. The maximum digital potentiometer value that can be set is 250. By limiting the potentiometer setting to 78% of this value, a top speed of 1.1 metres per second was achieved. This maximum speed was adequate to drive the robot safely in an indoor environment. It was determined through trial and testing that driving the robot at any speed beyond this top speed was not only dangerous to the robot itself but to its surroundings as well. Therefore, the maximum speed was maintained at 1.1 metres per second throughout the testing phase.

During the initial phase of resurrecting MARVIN's drive system, a conversion factor of 12731 pulses per metre was calculated to convert the odometry counts to a distance value in metres (section 3.5.2). This factor was calculated by multiplying the drive system reduction ratio (56.51) with the number of slots on the odometer code wheel (256) and dividing the result by the circumference of the wheels (1.037 m), giving the number of pulses per metre. This value was proved to be inaccurate following an experiment that was carried out by driving the robot through a distance of 2 metres in the postgraduate labs (detailed below). This inaccuracy could be due to changes in the tyre pressures, altering the circumference of the wheels.

This experiment involved instructing the mechatron to travel a distance of 2 metres over various velocities, measuring the actual travelled distances with a tape measure fixed to the floor. The distance the robot thought it had travelled was then divided by the measured distance of 2 metres, giving a distance ratio that was plotted against the velocity. The results from this experiment are depicted in Figure 7.1.

An average line (dotted) is indicated on the diagram to show the average distance ratio of MARVIN (1.03078) for velocity values greater than 0.3 metres per second. At its slowest speed, which is at 0.1 metres per second, the robot was not very responsive and the behaviour was not consistent. Therefore, it was decided to measure the distances at speeds higher than 0.2 metres per second. As the average distance ratio was greater than one, the mechatron thought it was travelling further than it was expected to travel. At velocities closer to the minimum, the mechatron was travelling closer

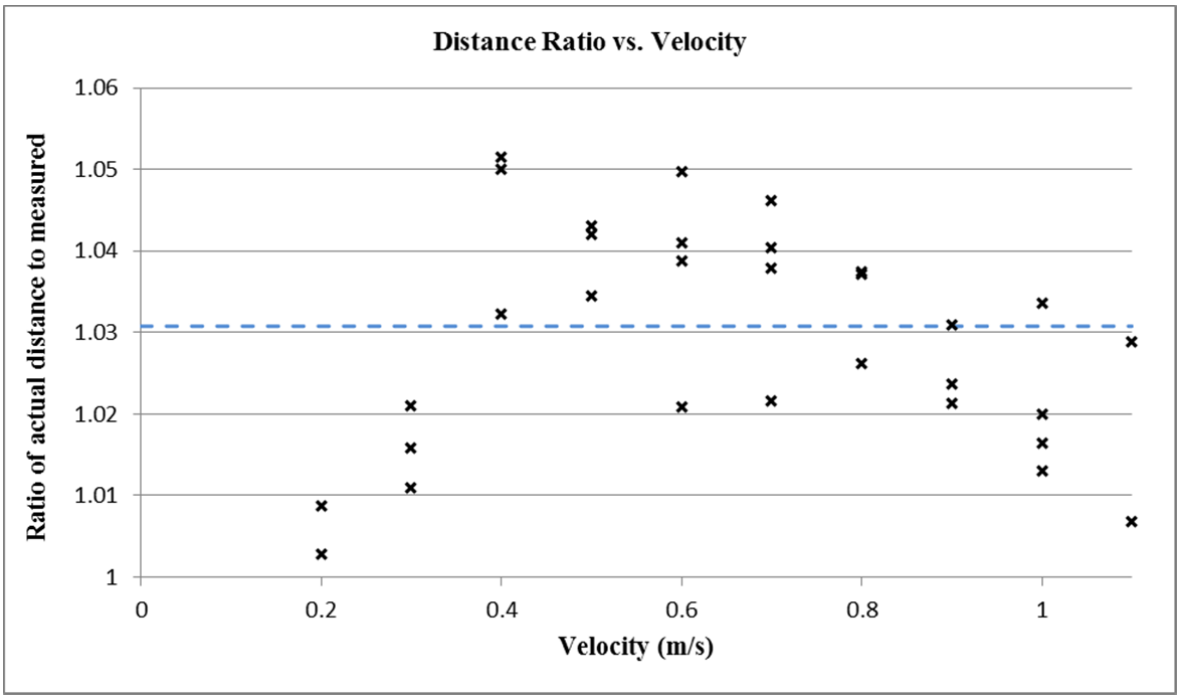

Figure 7.1 MARVIN's distance ratio vs velocity plot

to the intended set distance. This could be due to the higher degree of wheel slippage at speeds greater than 0.3 metres per second. As the robot is intended to operate at speeds between 0.3 and 0.7 metres per second, the average ratio of 1.03078 is used as a multiplier to find a more suitable conversion factor of 13122 pulses per metre.

7.2 PID Tuning

The trial and error method was used to tune the closed loop system which included the feedback from the motor encoders. The output from the PID controller was plotted on Matlab which helped to identify the interaction of different errors, process variable and manipulated variable.

The trial and error method approach first puts the system into a rough solution from which small tweaks are performed to perfect the response of the system. Initially, each coefficient of the PID controller is set to zero. Then, the proportional component is increased until a steady oscillation is

obtained. Once the proportional gain has been set to obtain a desired fast response, the integral coefficient is increased to stop the oscillations. The integral term is tweaked to achieve a minimal steady state error. After the proportional and integral coefficients are set to get the desired fast response with minimal steady state error, the derivative term is increased until the control loop is acceptably quick to reach the set point. This process is performed to find the coefficients for both the left and right control loops until the desired values are obtained. Figure 7.2 illustrates the output of the PID controller where the set-point variable (SP), process variable (PV), and the manipulated variable (MV) are plotted against time. As shown in the plot, the process variable takes about 0.6 seconds to settle to a steady state. However, given the intended speeds of MARVIN this value is not very significant.

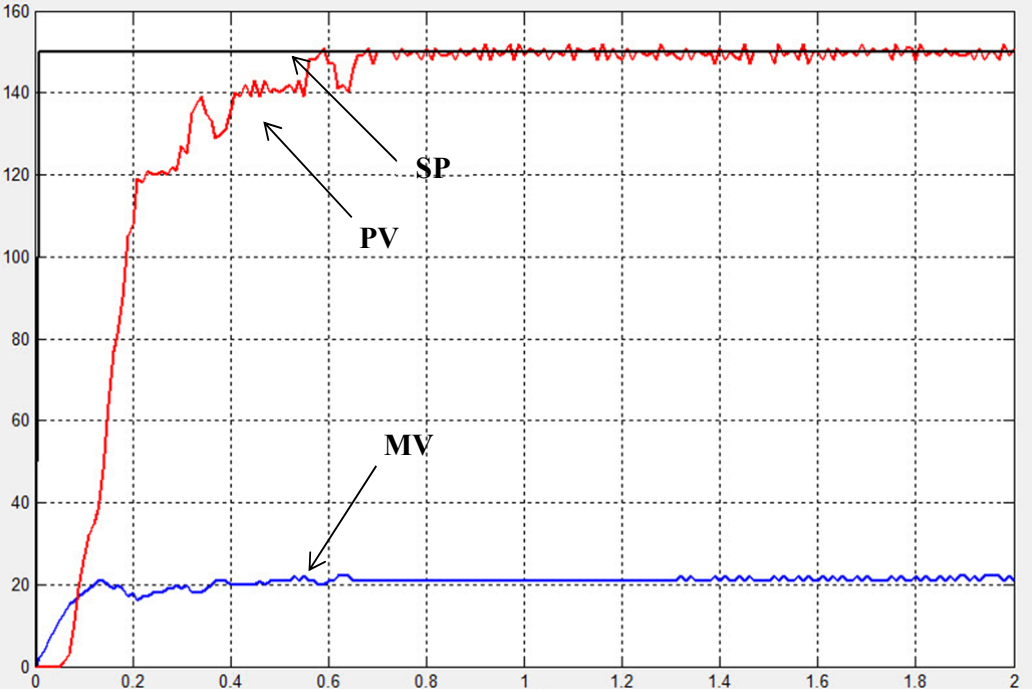

Figure 7.2 Plot of the tuned PID controller

7.3 Corridor Environment Test Results

The effectiveness of the navigation algorithm was tested by conducting several experiments in the corridors of the third floor of the Laby building. These tests conducted include:

- Navigating MARVIN down the length of the corridor from the postgraduate lab (LB313) to stop near the workshop door at the end of the corridor.
- Performing a left turn in to the postgraduate lab (LB313).
- And finally, instructing MARVIN to travel a fixed distance avoiding obstacles.

7.3.1 Navigating down the length of the corridor

In this test, results from a sample journey of three velocities (0.3, 0.5, and 0.7 metres per second respectively) were obtained and can be seen in the Figures 7.3 - 7.5 along with the generated velocity profiles. The wall positions are indicated in blue lines in each of the figures.

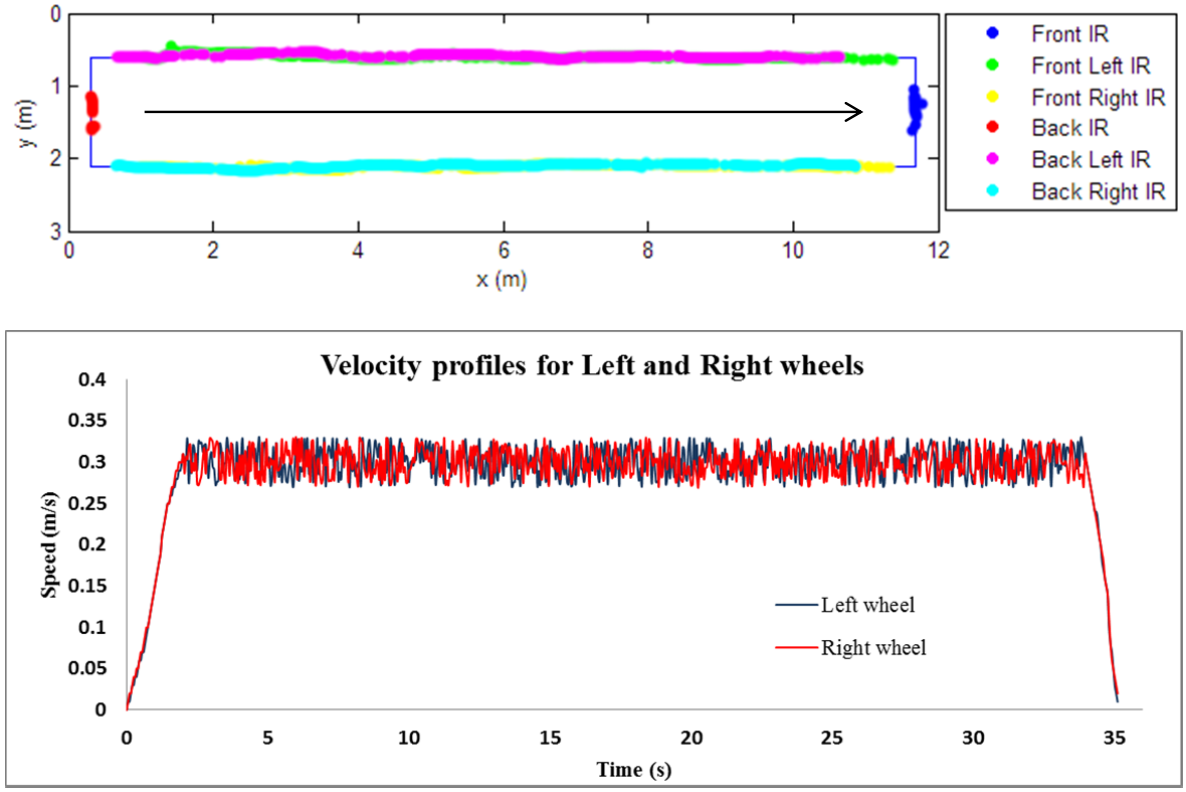

Figure 7.3 Journey of MARVIN along the corridor at 0.3 m/s

In each of these cases, the mechatron was instructed to travel a straight distance of 10.5 metres in the "x" direction (indicated by the black arrows in each of the maps) from a fixed starting point of

(0.5, 1.36) metres with respect to the origin of the map. It should be noted here that the values of x and y-axis have been swapped to change the orientation of the map.

As illustrated in the sample journeys, the infrared sensors show the wall positions for the entire journey in each sample with MARVIN successfully stopping near the workshop room (target position). The velocity profile plots shown below each map in the Figures 7.3 - 7.5 depict how MARVIN adhered to the intended speeds for each journey. In each of the three sample journeys, MARVIN adhered to the target trajectory keeping to the middle of the corridor. The generated velocity profile in each case indicates MARVIN's deceleration when the goal is reached. At the speed of 0.3 m/s, it took much less time to come to a halt compared to the speed of 0.7 m/s.

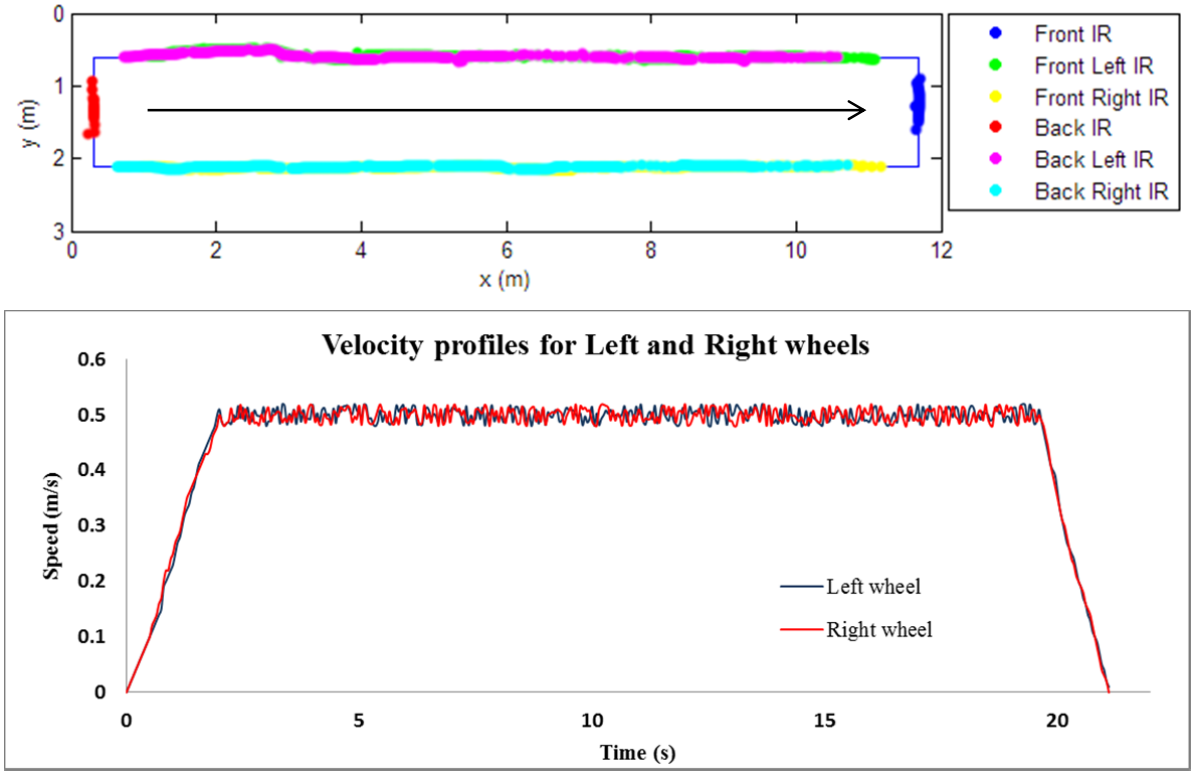

Figure 7.4 Journey of MARVIN along the corridor at 0.5 m/s

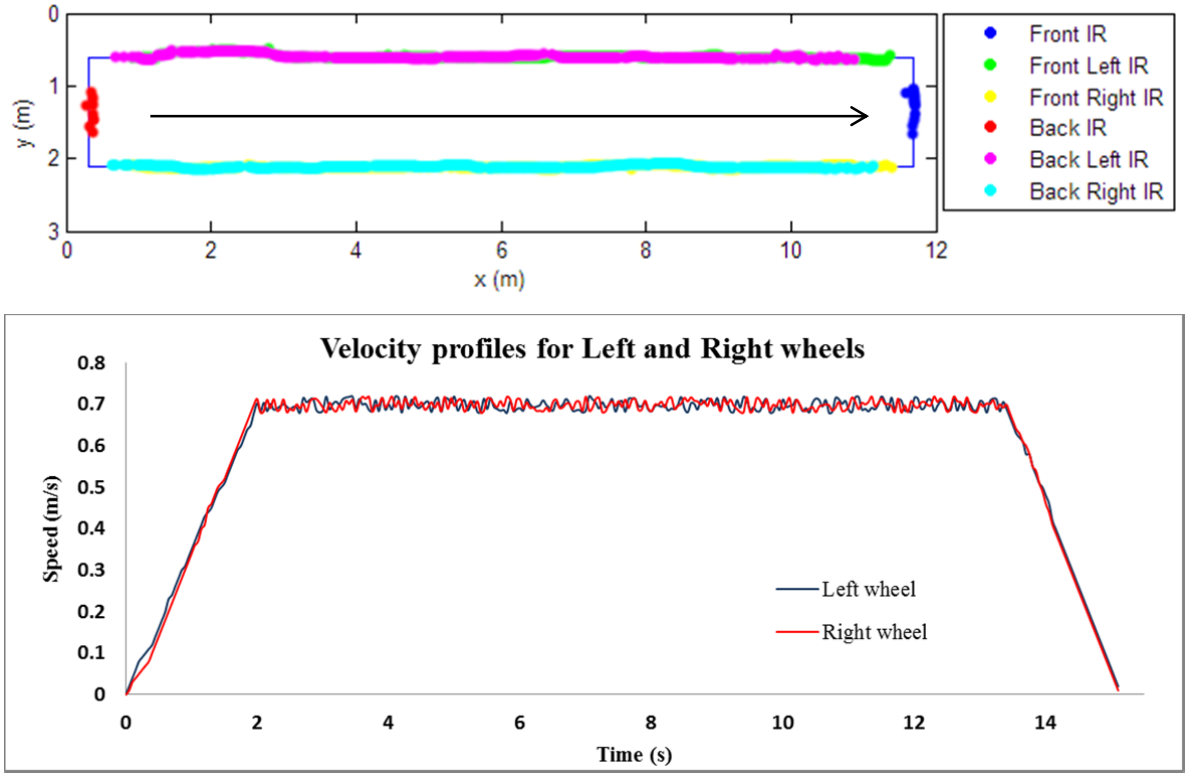

Figure 7.5 Journey of MARVIN along the corridor at 0.7 m/s

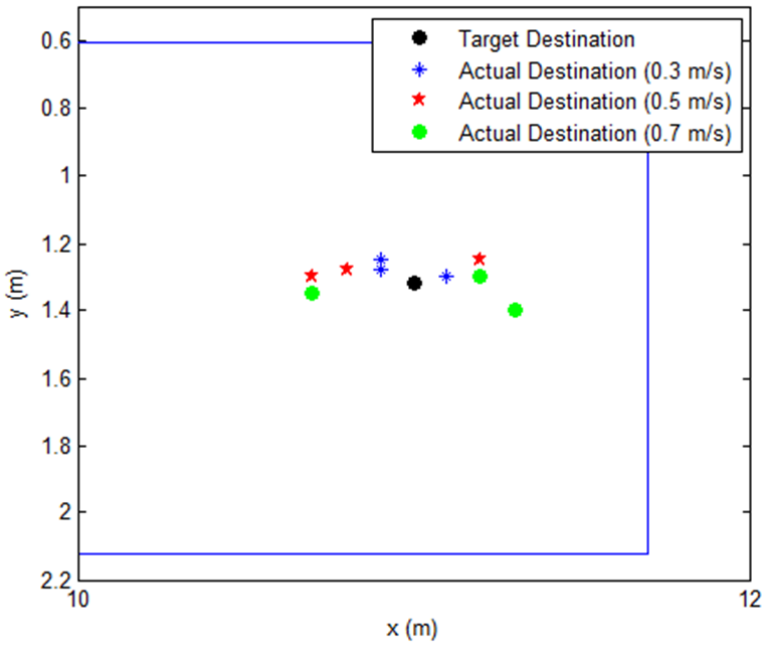

Figure 7.6 Comparison of the target and actual positions s resulting from a journey along the corridor

Figure 7.6 illustrates MARVIN's expected target position and the actual target position after the completion of the designated task for each of the instructed velocities on a zoomed-in view of the main environment. The target position of MARVIN was set at the point (11, 1.36) metres with respect to the origin of the map. It can be seen from the figure that MARVIN performed better at a lower speed of 0.3 metres per second while falling slightly short of reaching the target destination when maintaining the other two velocities. This may be due to the minor inaccuracies in the localization and the landmark detection measurements. The furthest point MARVIN travelled was 0.3 metres away at 0.7 metres per second in the x direction, which is very close to the grid space of the target point.

7.3.2 Performing a left turn into the postgraduate lab (LB313)

The second test conducted was to navigate MARVIN down the corridor from the workshop end, performing a left hand turn into the postgraduate lab (LB313) and stopping inside the lab. A temporary wall was erected in the lab to create a boundary for MARVIN to navigate effectively. Figure 7.7 illustrates the map of the corridor with the postgraduate lab (LB313) and it indicates the direction of the turn MARVIN is instructed to do.

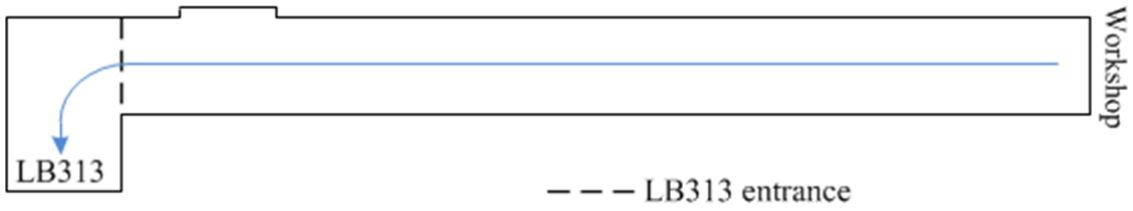

Figure 7.7 Map of the corridor and the postgraduate lab (LB313)

Figures 7.8 and 7.9 illustrate sample journeys of each of the two velocities (0.3 and 0.5 metres per second) used along with the generated velocity profile. This test was not conducted at the top speed of 0.7 metres per second due to the distance restrictions in the environment. In each of these sample journeys, MARVIN was instructed to navigate 11.4 metres along the corridor from the workshop room end with a starting point of (12.5, 1.36) metres towards the postgraduate lab (LB313), then to move 1.25 metres in the y direction. A black arrow is indicated on each of the figures representing the path that MARVIN took to complete the journey.

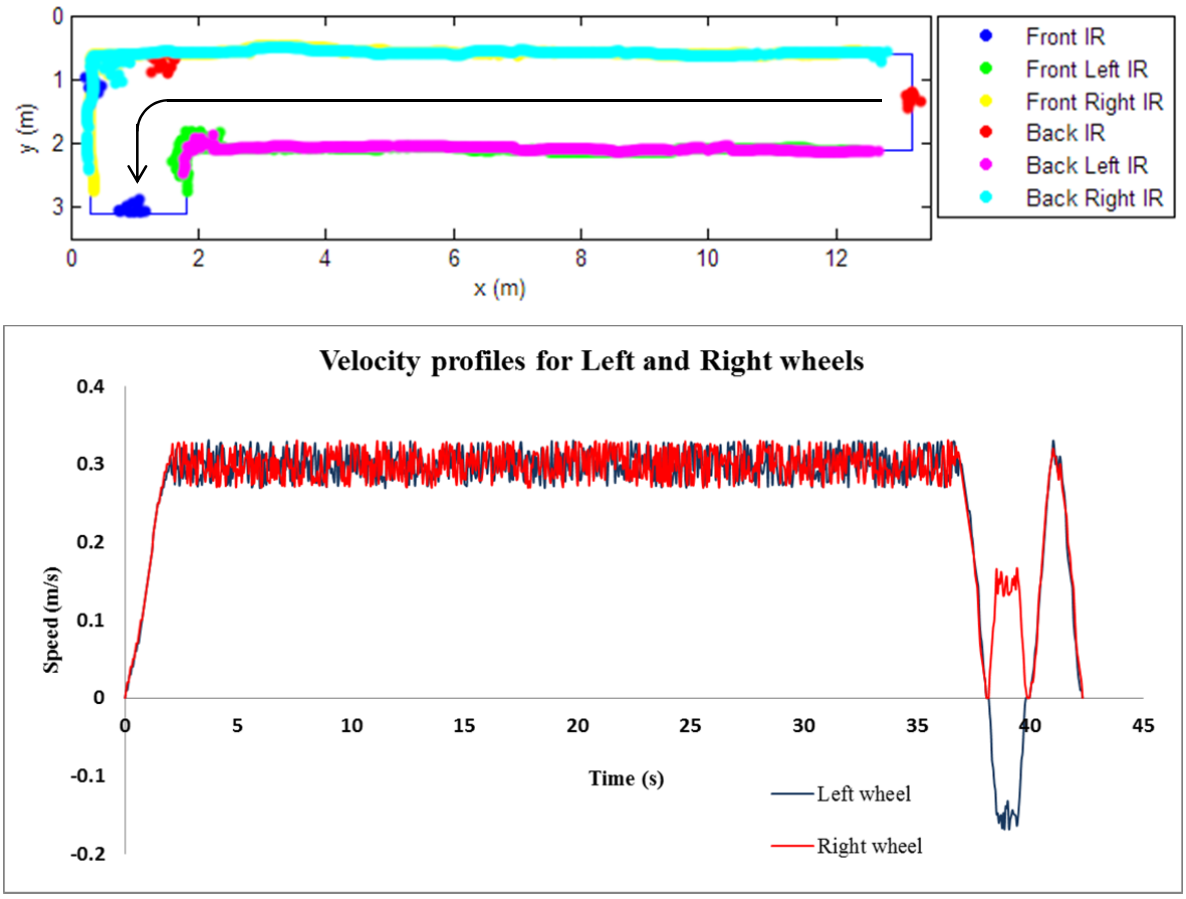

Figure 7.8 MARVIN turning to the postgraduate labs (LB313) at 0.3 m/s

It can be seen from the figures that MARVIN successfully completed the designated instructions, making a turn into the postgraduate labs and stopping at the given target position. The velocity profile of each of the two sample journeys shows that MARVIN was capable of maintaining the set velocity throughout the journey. The generated profile also shows three stages of the journey which were a straight travel down the corridor of 11.4 metres, a standing turn of about -45^{o} and another straight distance of 1.25 metres.

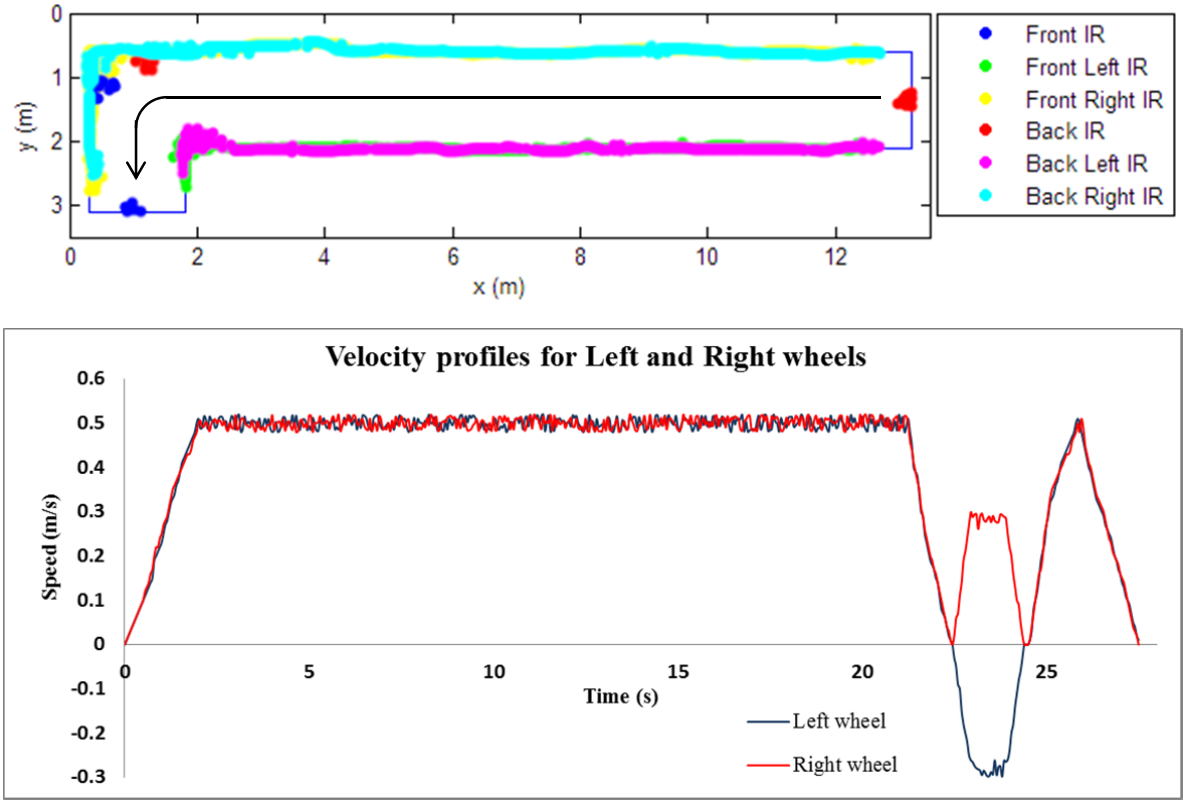

Figure 7.9 MARVIN turning to the postgraduate labs (LB313) at 0.5 m/s

MARVIN's trajectory in each case indicates that it kept to the middle of the corridor while traveling down the corridor and making the left turn. While making the left turn, the velocity profile taken at the speed of 0.5 m/s show that the control system ramps up the velocity of the right wheel more rapidly compared to the left wheel. Then it slows the right wheel to compensate for the "velocity surge".

Similar to the tests done in section 7.3.1, measurements of MARVIN's actual position and the target position were compared at the completion of all the journeys. These results are given in Figure 7.10.

Once again, it is apparent from the comparison that MARVIN responds well to low velocities. At a velocity of 0.5 metres per second, MARVIN has a slight tendency to be slightly right, and to overshoot or to fall short of the given target position. This can be attributed to a drift occurring during the left turn into the postgraduate lab. Drift can occur due to wheel slippage and minor differences between the rotational speeds of the wheels. The overshooting error can be corrected by travelling further after the standing turn, allowing the mechatron to gather more sensor readings thereby enabling it to reorient closer to any offset from the middle of the corridor.

The target position of this task was at (1.05, 2.62) metres with respect to the origin of the map. Generally, from the gathered measurements, the mechatron got very close to its target position after conducting a left turn and travelling a combined distance of 12.7 metres. The maximum deviation was -0.35 metres in the x direction at 0.5 metres per second. This position is within two grid spaces of the target point.

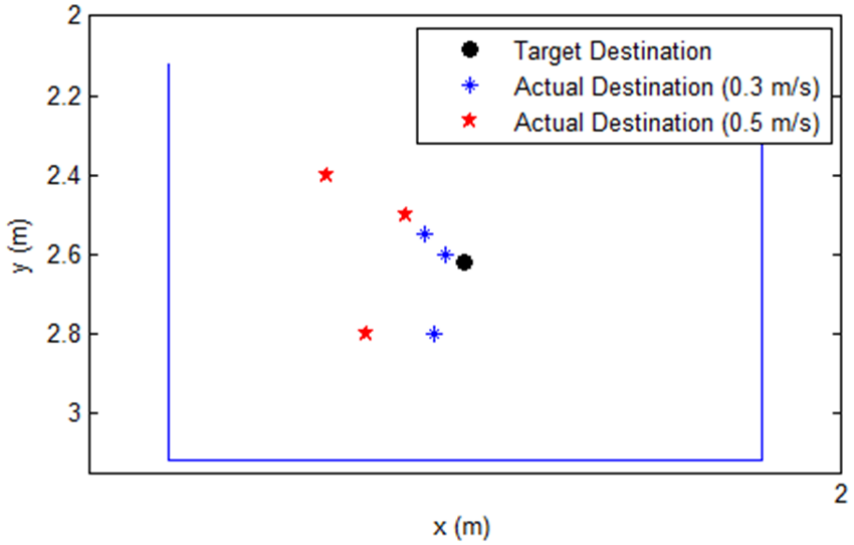

Figure 7.10 Comparison of the target and actual positions resulting from the turn into the postgraduate lab (LB313)

7.3.3 Obstacle avoidance

A third and final test was conducted to test MARVIN's obstacle avoidance capabilities. MARVIN was instructed to travel to a target position and detect any obstacles positioned on its path. If any obstacles were detected, it is expected that the mechatron would avoid them by executing appropriate manoeuvres and get back on course, stopping in the originally intended position. The control system adds a safety margin of 0.2 m to allow the robot to stop before colliding with obstacles. Stopping before an obstacle, the control system computes both the angular velocity and linear velocity to reach the goal. This causes the robot to make a turn and travel at the determined velocities. Once again, another obstacle is detected which in this instance is the wall. Reaching the obstacle, the control system repeats the process of computing a new angular velocity and a new linear velocity to reach the target the destination. The plots of these journeys and the generated velocity profiles are depicted in Figures 7.11 and 7.12.

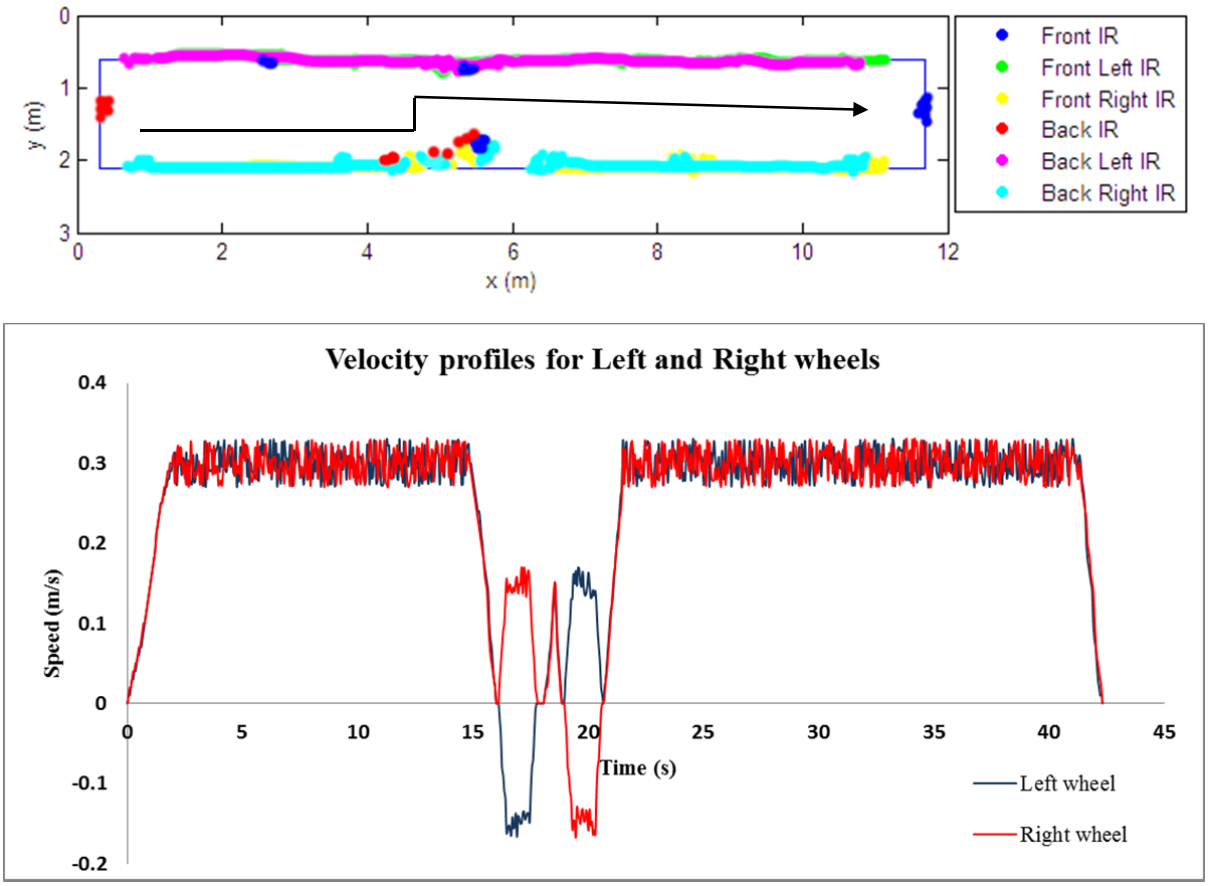

Figure 7.11 Avoid an obstacle at 0.3 m/s

Once again this test was sampled at the two velocities (0.3 and 0.5 metres per second). This test was not conducted at the top speed of 0.7 metres per second due to lack of space in the environment during any standing turns. MARVIN started the journey from a fixed point of (0.5, 1.5) metres with respect to the origin. For this test, a lab waste disposal bin was used as an obstacle and was placed at the point (5.5, 1.74) metres with respect to the origin.

The black arrow on these sample journeys indicates the trajectory of MARVIN as it avoids the obstacle on its path. At the point (5.5, 1.74) metres on the map, a collection of infrared sensor readings indicate the obstacle the mechatron has detected. The generated velocity profiles illustrate the number of instructions carried out to avoid this obstacle and to get MARVIN back on the correct course.

The first of the instructions indicate that it travelled straight down the corridor until the obstacle was detected in its path. As a result of the obstacle avoidance protocol mentioned earlier, it immediately made a standing left turn to avoid the obstacle. It then travelled a very short distance before making a standing right turn to stay on course.

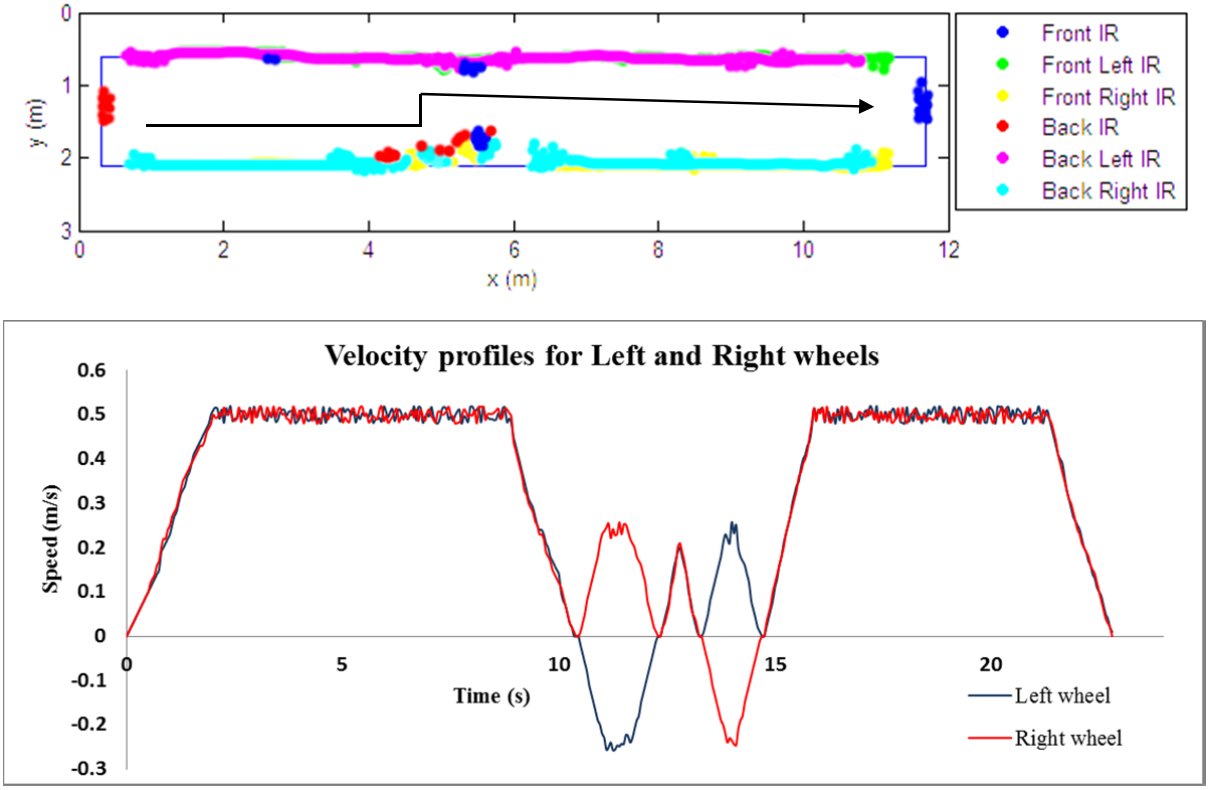

Figure 7.12 Avoid an obstacle at 0.5 m/s

After MARVIN avoided the obstacle and stopped, the intended position and the actual position are plotted for comparison as given in Figure 7.13.

Figure 7.13 shows that the robot has steered to the right of the target position of (11, 1.2) metres. It is apparent that neither of the two velocities used drove the mechatron closer to the target point. The worst drift was 0.3 metres away from the target point in the y direction at a velocity of 0.5 metres per second. With respect to the x direction, the worst result was -0.3 metres which occurred also at 0.5 metres per second. Similar to the other tests conducted previously, the mechatron performed well at the lowest speed. Although there was a drift, these results can be accepted given the number of turns it executed to avoid the obstacle and also lack of an accurate guidance system (combination of odometers and infrared sensors).

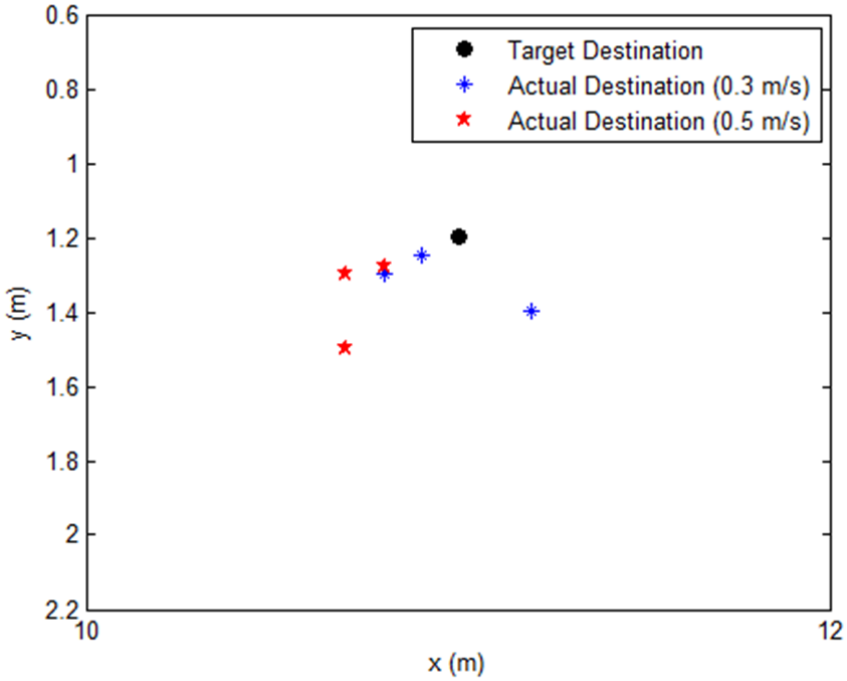

Figure 7.13 Comparison of the target and actual positions results from avoiding an obstacle

7.4 Summary

Prior to conducting any navigation tests on MARVIN, it was decided to calibrate the odometers. Initially, a odometer conversion factor of 12731 pulses per metre was calculated. Unfortunately, this factor did not reflect the actual conversion factor due to factors such as wheel radii changing with tyre pressure. To find an appropriate conversion factor, MARVIN was instructed to travel a distance of 2 metres at various speeds. This was compared against the expected distance it had travelled to plot the distance ratio. For velocities greater than 0.3 metres per second, the ratio values were accumulated and averaged to determine the average distance ratio. The average value was utilized to calculate the conversion factor of 13122 pulses per metre. This procedure was carried out on both sides of the mechatron to find two different conversion factors respectively.

The PID controller designed in section 3.5.5 was tuned using a trial and error scheme. To find a more stable system, the coefficients of the controller were manually tuned by plotting a graph. Using the graph, it was determined that it takes approximately 0.6 seconds to settle to a steady state. Although this is not tuned perfectly it should be sufficient for the purpose of MARVIN's navigation.

MARVIN was driven through a number of experiments to test the accuracy of navigation system. It was driven through the corridor of the third floor of Laby building at three different velocities ranging from 0.3 to 0.7 metres per second. The first experiment tested the accuracy of the navigation system when driven from a fixed starting point to a target point in a straight line. The second test was conducted to test the turn capabilities of the system. In this test, MARVIN successfully turned left into the postgraduate lab (LB313) and travelled towards the goal. The final experiment conducted tested the obstacle avoidance of MARVIN by instructing it to travel down the corridor avoiding a fixed obstacle. The second and third experiments were only carried out at speeds 0.3 and 0.5 metres per second due to the restrictions in the environment. Overall, the results obtained showed that MARVIN performed better at lower speeds. At higher speeds, it was not very consistent in terms of reaching the target destinations accurately mainly due to the drift generated from the turns.

Chapter 8 Conclusion and Future Work

This chapter concludes the work developed in this dissertation. First, the conclusions are presented by reviewing the work presented in the previous chapters. Next, the results from chapter 7 are discussed followed by detailed ideas for possible future improvements.

8.1 Project Review

This thesis deals with an architecture for controlling a mobile robot in accordance with the main objective in robotics, which is to develop an intelligent robot control system capable of suitable responses to dynamic environments. This control architecture has to process desirable features such as easy design, flexibility, real-time response, adaptability, coherent behaviour, and granularity.

The navigation system is designed and developed for a platform composed of a mobile robot called MARVIN, previously constructed at Victoria University. MARVIN is a two-wheeled robot which consists of a 360° infrared sensor network and two optical encoders to measure the wheel position and velocity. Chapter 3 gives an overview of each of these sensors including the details of the hardware in MARVIN. Initially, odometer calibration was carried out after noticeable discrepancies appeared between the distance the mechatron thought it had travelled and the actual distance it had travelled. Several tests were performed and results compared to determine the difference in these distances (section 7.1). To convert the odometry counts to a distance, a conversion factor of 12731 pulses per metre was determined. After taking the experimental results, the conversion factor was revised as 13122 pulses per metre. The only upgrades made to MARVIN were to bypass the use of a quadrature decoder to count the encoder pulses and to employ two independent PID controllers to maintain the desired speed. These PID controllers were tuned using a simple trial and error method. The coefficients that resulted in minimum oscillations were selected as the final values for driving the PID controllers (section 7.2).

A hybrid architecture, combining the benefits of both reactive and deliberative control, developed at Victoria University has been chosen as the basis for the underlying navigation system. As discussed in chapter 4, the deliberative control is developed using a modified version of the A * path planning algorithm and a rectangular occupancy grid map. The reactive component of the architecture combines a modified dynamic window approach and a direction sensor. In order to represent MARVIN on a common reference frame, the sensor readings from rangefinder sensors and encoders is fused together to form a common internal representation. This correlation is used to find

the position and orientation of the device. Sensor fusion is achieved using the dynamic weighted average algorithm allowing each of MARVIN's sensors to make a contribution.

Chapter 5 outlines the programming languages and tools used to implement MARVIN's control and navigation systems. The navigation system is developed using C# in combination with the libraries from the Microsoft Robotics Developer Studio. The developed application is compiled and debugged on Visual Studio. GNU C is used to develop the firmware of MARVIN's hardware.

The implementation details of the navigation system are described in chapter 6. The navigation system was developed following the Service-Oriented Architecture (SOA) and an event driven approach. At the heart of the software framework, the three layered architecture provides the drivers for the hardware, integrates the core navigation functionality, and presents a diagnostic tool for monitoring the system. The driver layer consists of several services to control the infrared sensors and the actuators, where each service is implemented as an abstraction interface to control the respective low-level hardware. Together with these services, the fundamental competences of the navigation algorithm is aggregated and integrated in the middle layer to form the orchestration service. All of these services are loosely coupled together. A graphical user interface is also developed as a service which can be executed on a remote computer to simplify the configuration of the navigation system properties.

To test the effectiveness of the navigation architecture, several experiments were conducted in the third level of the Laby building at Victoria University. Section 7.3 details the results from these experiments. The first test required MARVIN to travel a straight distance of 10.5 metres from a fixed point down the corridor stopping near the workshop room. This was conducted at velocities of 0.3, 0.5 and 0.7 metres per second. As expected, this test resulted in a stable movement as the device traversed down the corridor at the expected velocities. The mechatron reached close to the target points without much drift. The worst result was 0.3 metres in the x direction from the target destination at the maximum speed of 0.7 metres per second. The second test involved MARVIN travelling 11.4 metres along the corridor towards the postgraduate labs (LB313) and making a left turn and moving in the 1.25 metres in the y direction. This was conducted at velocities of 0.3 and 0.5 metres per second. The resulting actual destinations proved to be reasonably close to the target destination however there was a drift during the left turn causing the mechatron to stop to the right of the target. The worst deviation was -0.35 metres in the x direction at 0.5 metres per second. A further test was conducted involving the obstacle avoidance at velocities of 0.3 and 0.5 metres per second. The mechatron was instructed to travel the same distance as the first test. A waste bin was used as the obstacle to be avoided. MARVIN successfully avoided the obstacle at the given

velocities without resulting in any disordered behaviour. Once again there was a drift to the right of the target destination. The worst result was -0.3 metres in the x direction and 0.3 metres in the y direction at a velocity of 0.5 metres per second. In retrospect, there seems to be heading errors due to inaccuracies in speeds when the mechatron turns. It was more susceptible to these errors at speeds above 0.3 metres per second.

Overall, these results prove that the implemented navigation algorithm has met all the objectives, efficiently navigating MARVIN around a given environment and being able to avoid obstacles in its path.

8.2 Future Work

8.2.1 Odometer calibration improvements

The calibration discussed in section 7.1 is very specific to the environment the experiments were performed hence MARVIN's odometers must be calibrated to function in other environments with a minimum error displacement. To suit large indoor environments, MARVIN must be instructed to travel at least a distance of 10 metres to measure the displacement between the perceived distance and the actual distance it travels. This experiment should also be carried out on different floor surfaces to determine the effects of wheel slippage on the calibration. Thus, this will reflect a more suitable conversion factor allowing MARVIN to function in other indoor environments.

8.2.2 Improving the drift during turns

The results demonstrated that further improvement is required to correct the trajectory of the robot during turns. This could be due to inconsistencies in the speeds of the wheels. One way of solving this problem can be the use of integrated turns instead of standing turns which allows the robot to rotate about a point that lies along the left and right wheel axis. An angle can be sent to the control system to compute the necessary speed and distance each wheel should turn. In order to achieve an integrated turn the control system will be needed to compute the speeds of the wheels much earlier than standing turns.

8.2.3 Other enhancements

The graphical interface currently does not update the position and orientation of the mechatron giving a "live" view of where it is positioned in the map. This is due to a limitation in large dataset transfer rates. Currently, the position and orientation values are parts of a large dataset which also includes the rangefinder sensor data. This can be avoided by implementing a new dataset to include

the position and orientation information however a new interface will need to be defined to support the dataset.

Appendix A: CD Contents

The attached CD contains the following:

- **Soft copy of this thesis**
- **Photo Gallery**
- **Software Projects**

  GCC:

  - Rhino Drive Firmware

  C# Project (MRDS):

  - Generic Module Service
  - Rhino Drive Service
  - Sensor Network Service
  - Hybrid Navigation Service
  - Teleoperation Service

Glossary

A	Amp
Ah	Amp-Hour
AI	Artificial Intelligence
API	Application Programming Interface
CCR	Concurrency and Coordination Runtime
CORBA	Common Object Request Broker Architecture
DLE	Data Link Escape
DSS	Decentralized Software Services
DSSP	Decentralized Software Services Protocol
ETX	End of Text
FIFO	First-In-First-Out
GCC	GNU Compiler Collection
HTTP	Hypertext Transfer Protocol
IDE	Integrated Design Environment
MARVIN	Mobile Autonomous Robotic Vehicle for Indoor Navigation
MRDS	Microsoft Robotics Developer Studio
OOC	Object-Oriented Computing
PCB	Printed Circuit Board
PID	Proportional, Integral and Derivative
REST	Representational State Transfer
RMI	Remote Method Invocation
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computing
USB	Universal Serial Bus
VPL	Visual Programming Language
VSE	Visual Simulation Environment
WPF	Windows Presentation Foundation
XOR	Exclusive OR
XSLT	Extensible Stylesheet Language Transformation

References

Arkin, R. C. (1987). Towards cosmopolitan robots: Intelligent navigation in extended man-made environments. University of Massachusetts.

Arkin, R. C. (1998). Behavior-based robotics: The MIT Press.

Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D. P., & Slack, M. G. (1997). Experiences with an architecture for intelligent, reactive agents. Journal of Experimental & Theoretical Artificial Intelligence, 9(2-3), 237-256.

Botti, V., Carrascosa, C., Julián, V., & Soler, J. (1999). Modelling agents in hard real-time environments. Multi-Agent System Engineering, 63-76.

Brooks, R. (1986). A robust layered control system for a mobile robot. Robotics and Automation, IEEE Journal of, 2(1), 14-23.

Bruyninckx, H. (2001). Open robot control software: the OROCOS project.

Carnegie, D. A., Prakash, A., Chitty, C., & Guy, B. (2004). A human-like semi autonomous mobile security robot. Paper presented at the Proc. 2nd International Conference on Autonomous Robots and Agents, Palmerston North, NZ, .

Cepeda, J. S., Chaimowicz, L., & Soto, R. (2010). Exploring Microsoft Robotics Studio as a Mechanism for Service-Oriented Robotics.

Chand, P. (2011). Development of an artificial intelligence system for the instruction and control of cooperating mobile robots.

Chand, P., & Carnegie, D. A. (2011). Development of a navigation system for heterogeneous mobile robots. Int. J. Intell. Syst. Technol. Appl., 10(3), 250-278. doi: 10.1504/ijista.2011.040349

Chang, M., He, J., & Castro-Leon, E. (2006). Service-Orientation in the Computing Infrastructure. Paper presented at the Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering.

Chen, Y. (2006). Service-Oriented Computing in Recomposable Embedded Systems.

Chen, Y., & Bai, X. (2008). On robotics applications in service-oriented architecture.

Chen, Y., & Tsai, W. (2008). Distributed service-oriented software development.

Connell, J. H. (1992). SSS: A hybrid architecture applied to robot navigation.

dos Santos, A. G. N. C. (2008). Autonomous Mobile Robot Navigation using Smartphones Extended Abstract.

Firby, R. J. (1990). Adaptive execution in complex dynamic worlds. Citeseer.

Fitzpatrick, P. (1997). A novel behaviour-based robot architecture and its application to building an autonomous robot sentry. Citeseer.

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. Robotics & Automation Magazine, IEEE, 4(1), 23-33.

Gat, E. (1991). Integrating reaction and planning in a heterogeneous asynchronous architecture for mobile robot navigation. ACM SIGART Bulletin, 2(4), 70-74.

Gerkey, B., Vaughan, R. T., & Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems.

Gerkey, B., Vaughan, R. T., & Howard, A. (2005). Player/Stage, from http://playerstage.sourceforge.net/

Gerkey, B. P., Vaughan, R. T., Stoy, K., Howard, A., Sukhatme, G. S., & Mataric, M. J. (2001). Most valuable player: A robot device server for distributed control.

Guibas, L. J., Motwani, R., & Raghavan, P. (1997). The robot localization problem. SIAM J. Comput., 26(4), 1120-1138.

Innocenti, B. (2008). A multi-agent architecture with distributed coordination for an autonomous robot.

Jackson, J. (2007). Microsoft robotics studio: A technical introduction. IEEE Robotics & Automation Magazine, 14(4), 82-87. doi: 10.1109/M-RA.2007.905745

Johns, K., & Taylor, T. (2008). Professional Microsoft Robotics Developer Studio: Wrox Press Ltd.

Jones, M. T. (2008). Artificial Intelligence: A Systems Approach: Jones & Bartlett Learning.

Junior, V. G., Parikh, S. P., & Junior, J. O. (2006). Hybrid deliberative/reactive architecture for human-robot interaction.

Kamath, J.-F. A. (2009). Reflective communications framework: an approach to rapid deployment of communications architectures in distributed systems. University of Florida.

Kapach, K., Edan, Y., & Xiao, Y. (2007). Adaptive weighted average sensor fusion algorithms for mobile robots.

Kolp, M., Giorgini, P., & Mylopoulos, J. (2006). Multi-agent architectures as organizational structures. Autonomous Agents and Multi-Agent Systems, 13(1), 3-25.

Konolige, K., Myers, K., Ruspini, E., & Saffiotti, A. (1997). The Saphira architecture: A design for autonomy. Journal of Experimental & Theoretical Artificial Intelligence, 9(2-3), 215-235.

Kramer, J., & Scheutz, M. (2007). Development environments for autonomous mobile robots: A survey. Autonomous robots, 22(2), 101-132.

Kramer, J., & Scheutz, M. (2007). Development environments for autonomous mobile robots: A survey. Auton. Robots, 22(2), 101-132. doi: 10.1007/s10514-006-9013-8

Lee-Johnson, C. P., & Carnegie, D. A. (2006). Towards a Computational Model of Affect for the Modulation of Mobile Robot Control Parameters'.

Leonard, J., & Durrant-Whyte, H. (1991). Mobile robot localization by tracking geometric beacons (Vol. 7). New York, NY, ETATS-UNIS: Institute of Electrical and Electronics Engineers.

Loughnane, D. J. (2001). Design and Construction of an Autonomous Mobile Security Device. A thesis submitted of Master of Science in Physics and Electronic Engineering at the University of Waikato.

Macek, K., PetroviC, I., & Ivanjko, E. (2003). An approach to motion planning of indoor mobile robots.

MARVIN. (2011, December 07, 2011). Retrieved December 10, 2011, from http://ecs.victoria.ac.nz/Groups/Mechatronics/MARVIN

Mataric, M. J. (2007). The Robotics Primer (Intelligent Robotics and Autonomous Agents): The MIT Press.

Mataric, M. J. (2009). The Basics of Robot Control.

McClymont, J. (2011). MARVIN User Manual. Victoria University of Wellington.

McClymont, J., & Carnegie, D. A. (2008). A Hardware Design Philosophy for an Autonomous Robotic System. Paper presented at the Paper presented at ENZCon 2008, Auckland, NZ, .

Miro Manual. (2009). (pp. 128). Retrieved from http://developer.berlios.de/docman/display_doc.php?docid=1921&group_id=10725

Murphy, R. (2000). Introduction to AI robotics: The MIT Press.

Muscettola, N., Dorais, G., Fry, C., Levinson, R., & Plaunt, C. (2002). Idea: Planning at the core of autonomous reactive agents.

Nakhaeinia, D., Tang, S., Mohd Noor, S., & Motlagh, O. (2011). A review of control architectures for autonomous navigation of mobile robots. Int. J. Phys. Sci, 6(2), 169-174.

Orebäck, A., & Christensen, H. I. (2003). Evaluation of architectures for mobile robotics. Autonomous robots, 14(1), 33-49.

Pirjanian, P. (1997). An overview of system architecture for action selection in mobile robotics. Laboratory of Image Analysis, Aalborg University, Aalborg, Denmark.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., . . . Ng, A. Y. (2009). ROS: an open-source Robot Operating System.

Qureshi, F., Terzopoulos, D., & Gillett, R. (2004). The cognitive controller: a hybrid, deliberative/reactive control architecture for autonomous robots. Innovations in Applied Artificial Intelligence, 1102-1111.

Rosenblatt, J. K. (1997). DAMN: A distributed architecture for mobile navigation. Journal of Experimental & Theoretical Artificial Intelligence, 9(2-3), 339-360.

Simmons, R. G. (1994). Structured control for autonomous robots. Robotics and Automation, IEEE Transactions on, 10(1), 34-43.

Singh, S. P. N., & Thayer, S. M. (2001). ARMS: Autonomous Robots for Military Systems: a Survey of Collaborative Robotics Core Technologies and Their Military Applications: Carnegie Mellon University, The Robotics Institute.

University of Ulm Robotics Group. (2005). Miro - middleware for robots, from http://miro-middleware.berlios.de/

Utz, H., Sablatnog, S., Enderle, S., & Kraetzschmar, G. (2002). Miro-middleware for mobile robot applications. Robotics and Automation, IEEE Transactions on, 18(4), 493-497.

Victorino, A. C., Rives, P., & Borrelly, J. J. (2000). Localization and map building using a sensor-based control strategy.

Willow Garage. (2012). ROS/Introduction - ROS Wiki, 2012, from http://www.ros.org/wiki/ROS/Introduction