Development of a Real-Time Aided Inertial Navigation System For an Underwater Robotic Vehicle

Mohammad Dzulkifli Bin Mohyi Hapipi

School of Mechanical & Aerospace Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2005

ABSTRACT

Underwater robotic vehicles (URV) have a practical significance in the offshore engineering field, especially in underwater environment surveying, man-made structure inspection, and pipe and cable laying. Most URVs are designed to be autonomous as in autonomous underwater vehicle (AUV) or remotely operated vehicles (ROV). Modern URVs uses onboard navigation system to provide the necessary information about its environment. It is one of the primary components in all types of URV and it is used to define the states of the vehicle such as acceleration, velocity, position and angular rate and position. It consists of the physical hardware such as sensors and the computer system and the software layer such as the data acquisition, data management and the navigational algorithms.

This thesis explores the development on an Aided Inertial Navigation System (AINS) for a compact, low cost twin-barrel URV developed by the Mechanical and Production Engineering (MPE) department of Nanyang Technological University (NTU), Singapore. This system uses low cost sensors, which produce large drifts and bias errors. Low cost sensors have errors up to 1000 times larger than high cost precision sensors. The URV utilises the IMU600CA-200 inertial sensor from Crossbow and is aided by non-inertial sensors such as Argonaut MD Velocity Doppler from Sontek, KVH compass, Tritech Altimeter, Crossbow Tilt sensor, Sonavision scanning sonar and Falmouth pressure sensor. Relying of the fast update rate of the IMU, the additional sensors aids the IMU by correct the vehicle states at periodic interval. All the data processing had to be done by an onboard Octagon microcomputer running at 133 MHz. For optimal performance, QNX 4.25 operating system was used to enable real time processing for the sensors. For the navigation system to function as required, the sensors, computer hardware and operating system need to be optimally configured.

Data acquisition and data management architecture were designed for the system. One of the primary factors that affect the system performance is in obtaining data from the sensors. A polling only configuration requires the process to wait for the data after the poll thus results in time wastage. On the other hand, interrupt only configuration requires the computer to service the interrupt, thus affects the running process. It was observed that using the interrupt/polling combination provides the best performance to the system since the process releases the CPU for other processes under idle states. A shared memory objects was used to manage the data, as it requires minimum synchronisation and allow other processes to extract it when required. It also complements the network data transfer via both message passing and message queue via Ethernet link between the URV control and navigation pod.

Navigation algorithms such as frame transformations, kalman and low pass filter, software integration and gravity and steady state compensation were designed and tested for the system. These algorithm, affects the performance of the designed architecture which is fundamental in maintaining a real time acquisition and processing. Therefore, the algorithms were tested based on the navigational accuracy and processing speed. Lab and navigation experiments were carried out to test the robustness of the algorithm which is represented by the accuracy of the navigational performance. A stable, robust real-time architecture is fundamental in the development of the navigation system.

ACKNOWLEDGEMENTS

All praise be to god who gave the author the vigilance and health to complete the research work. The author would like to thank the following people who have helped in one way or another in carrying out the project; Associate Professor Eicher Low, the author's project supervisor for his guidance and timely advice. Associate Professor Gerald Seet for his valuable advice and guidance; The staff of NTU RRC, Mr. Lim Eng Cheng, Mr. You Kim San and Ms Agnes Tan for their kind assistance; The staff of NTU SRC swimming pool; Not forgetting fellow colleagues from Robotics Research Center, with special mention given to Ong Kai Wei and Tan Ching Seong who helped with the suggestions and experiments. The author would also like to thank his family and friend who gave all the moral and financial support for the author to complete this level of education.

CONTENTS PAGE

CONTENTS PAGE

LIST OF FIGURES PAGE

LIST OF FIGURES

LIST OF TABLES

ABBR.	DEFINITIONS
ϕ, θ, ψ	Roll, Pitch And Yaw Angles Based On The Fixed Frame
p, q, r	Angular Rates About The Vehicle Frame's
a_x, a_y, a_z	Acceleration in the fixed frame X, Y, Z Axis
a_{xl}, a_{yl}, a_{zl}	Acceleration in the vehicle frame X, Y, Z Axis
a_x, v_x, s_x	V: velocity, S: position. Subscripts same as in acceleration
C, S (before angle)	Cosine And Sine Respectively
ATRV	All terrain robotics vehicle
AINS, INS	(Aided) inertial navigation system
ARV	Autonomous Robotics Vehicle
AUV	Autonomous Underwater Vehicle
BCD	Binary Coded Decimal
C.G.	Center Of Gravity
CPU	Central Processing Units
CURV	Cable Controlled URV
CORBA	Common Object Request Broker Architecture
ECEF	Earth Centred Earth Fixed (Frame)
EM wave	Electro Magnetic Wave
FDD	Floppy Disk Drive
FIFO, LIFO	First In First Out, Last In First Out
FOG	Fibre optic gyroscope
GPS	Global Positioning System
GUI	Graphical User Interface
G-vector, G	Gravity Vector
HDD	Hard Disk Drive
IIR filter	Infinite Impulse Response filter
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
IPC	Inter Process Communication
IRQ	Interrupt Request
ISA	Industry Standard Architecture
KF, EKF, UKF	Kalman filter, Extended KF, Unscented KF
LPF	Low Pass Filter
MHZ	Megahertz
OS	Operating System
PC	Personal Computer
PCI	Peripheral Connect Interface
QNX	A Real Time Operating System
RAM	Random Access Memory
ROM	Read Only Memory
LCROV, ROV	(Low Cost) Remotely Operated Vehicles
RRC	Robotics Research Centre
TCP/IP	Transmit Control Protocol / Internet Protocol
URV	Underwater Robotics Vehicle
VGA	Video Graphic Array

CHAPTER ONE

INTRODUCTION

The development of underwater robotic vehicle (URV) has grown rapidly in the last three decades [1]. Since then, the development of URV had separated into two paths namely manned and unmanned URVs. While early URV requires humans to be physically onboard the URV to control and navigate the system [2], rapid improvement in the navigation sensor technology prove the popularity of the unmanned system as a human replacement in the hazardous underwater environment. Research into unmanned systems such as the autonomous underwater vehicle (AUVs) and remotely operated vehicle (ROVs) are being done in many institutions around the world [3]. In these systems, problems arise in navigating the URV in the underwater environment. Without the line of sight, the states of the URV can only be predicted based on the onboard sensors. In building a navigation system for a low cost URV, this problem becomes more prominent as low cost sensors tends to have higher drifts and errors, thus provides a good opportunity for further research.

1.1 Background

The ocean has always been an important source of food and drinking water for humans since the dawn of mankind. As the human population grows, the large economy of scale turns these necessities into commercial viability. Furthermore, with other richness such as pearls and energy source and with the beauty of coral reefs, the ocean is a never-ending source for human commercial activities. However, the ocean turned into a "gold mine" in the last three-decade [1], with the discovery of oil and natural gas underneath the seabed. This fuelled research into URVs, initially designed mainly for military and research purposes, as a tool to replace humans in doing repair works where hazardous conditions are expected.

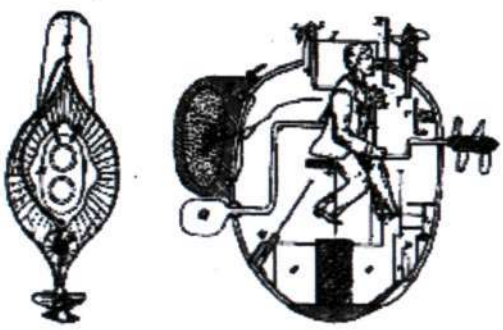
1.2 History and development of URV

The development of underwater vehicle can be traced back to 1623 when Cornelius Drebbel designed the first ever, working submarine. From then to 1776, submarines were designed just to test the capability of underwater transportation without having the test for its possible speed, range, depth, or features. Even when the first ever submarine, the 'Turtle' [2], figure 1.1(a), was designed for military purpose by David Bushnell, it is just an underwater vehicle with a specific purpose. However, it provides a blueprint for future underwater robotic vehicle with an actuator, a steering mechanism and an arm in the form of a drill with the human manning the system actuates the vehicle and the arm, navigate the vehicle to the required destination and making all the necessary decisions.

It took nearly two decades before the first URV was developed. Dimitri Rebikoff developed the first tethered ROV, named POODLE in 1953 [4]. However, at the beginning, ROV remained only for research works. ROVs only gained its fame when U.S Navy developed a recovery ROV, the Cable Controlled Underwater Recovery Vehicle (CURV) that managed to recover an atomic bomb lost off Palomares Spain in an aircraft accident in 1966 [5]. Early ROVs before 1970s were limited only to research and military use. Only in the late 1970s, commercial firms operating in offshore oil operations identified the commercial viability of ROVs.

It was only in the early 1980s that the advancement in the technology reinforced the development efforts in the ROVs. With the development of small, low powered computers and the advances in software systems, complex guidance and control algorithms together with complicated sensor and vision system can be implemented in a ROV [6]. This changed the ROV from a simple data acquisition and inspection vehicle into a work class ROV, capable of replacing humans in numerous underwater tasks. Furthermore, with this advancement, the Autonomous Underwater Vehicle (AUV) became a viable option for underwater operations.

Since then, the pace of URV development has greatly increased. The development further expanded into areas of cabling with the boom in information technology. These URVs were designed to crawl over the seabed, digging trenches, laying down, and reburying cables [7]. Cable inspection URVs were also designed to periodically do inspections on the cables. Today, URVs can be found mapping the seabed, inspecting ship wreckage, doing work in hazardous radioactive environment, inspecting man-made structures such as dams and bridges, studying underwater life form, even descending as deep as the deepest point in the ocean, the challenger deep, with depth of 10,911.8m (35,800ft) [8]. It is estimated that by this year, 2003, there are over 400 operational URV being used for all the different purpose [9]. URVs have reached a milestone from being equipment for research and military purposes into standard equipment for most underwater research and industries.



(a) 'Turtle' [2] the first underwater war vehicle

(b) 'Kaiko' [8] a URV capable of reaching the deepest depth of 10,911.8m

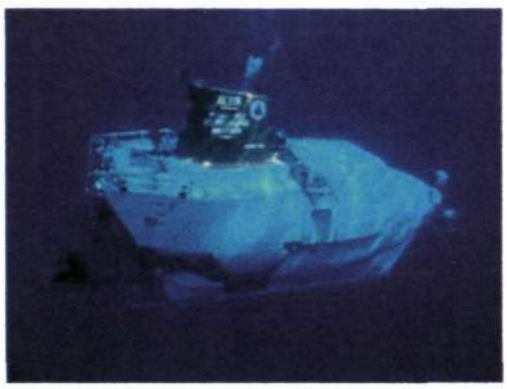
Figure 1.1 Difference between early and current underwater vehicle

1.3 Classification of commercial URVs

Ever since the growth in the development of URVs, a wide range of URVs were designed for different purposes. Most URVs can be classified as manned and unmanned vehicles. Manned vehicle can be classified by its passenger capacity, while unmanned vehicles can be classified by its type, mainly AUV, ROV, autonomous robotics vehicle (ARV) and crawler URVs.

1.3.1 Manned URVs

In the 1960s and 70s, manned URVs were popular among scientist and marine life researches as it provides close contact with the marine environment. Most of these URVs were designed with large pressure hull commonly used to accommodate one to four persons such as Alvin, figure 1.2(a), [10] and hemispherical glass dome giving the operator a clear view of the environment. Unfortunately, these URV did not find its popularity due to its size and cost. Even though the numbers starts to dwindle from an estimated 160 during its peak to 40 now [11], manned URV such as Mergo, figure 1.2(b), have found new uses in other areas such as tourism and recreation. Tourism URVs were designed to be able to take large number of passengers to view the underwater marine life and environment. On the other hand, recreational URVs are a cross between diving gear and mini URVs, giving a single person operator the capability of achieving the depth and speed of a URV and yet retain the flexibility and control of a diving apparatus.



(a) Alvin [10], a three man URV

(b) Mergo [11], a ten man tourist URV

Figure 1.2 Research and tourist manned underwater vehicles

1.3.2 Unmanned URVs

Remotely Operated Vehicles (ROVs)

ROVs are unmanned URVs designed with an operator, operating from a remote location. Most ROVs are designed with a tether commonly used to provide power to the URVs, transfer data between the surface and onboard computer and provide video signal link from the onboard camera to a monitor on the surface.

ROV can be classified into two classes, namely the work-class ROVs such as Triton XL, figure 1.3(a) [12] and inspection-class ROVs. Work-class ROVs are designed to be large and stable underwater to carry out underwater works ranging from cable and pipe lying to repair works. Being big, the ROVs can be fitted with hydraulic manipulators in order to carry out the repair works with heavy equipments such as grinders and cutters. Inspection-class ROVs on the other hand are designed to be small and nimble in order to inspect areas where large URVs cannot fit. One of the main uses of this class of ROV is to study underwater wreckage conditions in order to find the most feasible means of lifting ship wreckage.

Autonomous Underwater Vehicle (AUVs)

AUVs as in figure 1.3(a) are untethered, unmanned vehicles designed to conduct underwater surveys in predefined courses. Being untethered, this class of URV needs to be self sufficient in power, control and navigation capabilities thus giving the AUV the freedom to moving around in its environment. These capabilities led to AUVs deigned mainly for survey purposes such as mapping of the seabed and testing of the water condition within a particular region.

Autonomous Robotics Vehicle (ARVs)

Autonomous robotics vehicles are a hybrid between AUVs and ROVs. During the current time, ARV are conceptualised to be an autonomous work-class URV. Unfortunately, current available technology is insufficient in realising this class of vehicle as a fully functional URV.

Crawler URVs

The demand for specific uses of URVs led to the design of this class of URVs. Applications such as cable lying and inspection do not need a URV capable of ascending and descending in the water. Therefore, these URVs such as the gator, figure 1.3(d) are designed to particularly crawl on the seabed, guided by the cables for navigation.

(a) Triton XL, a work class ROV

(b) LCROV, an inspection class ROV

(c) OKPO, a surveying AUV

(d) Gator, a bottom crawler URV

Figure 1.3 Different types of unmanned underwater vehicle

1.4 Other application of URVs

Ever since the initial research into military URVs, the use of URV have diverted into a few different areas such as commercial and research use. The area into which the URV evolved is affected by the market conditions, thus affecting its design and configuration. While the ROVs as a work and inspection tools are the most common type of URVs available in the market, other URVs are also being designed with the main area of divergence is in the area of military and research.

Military URVs

URVs used military purposes are mainly in the class of AUVs. The URVs designed for military purposes are mainly large, capable of reaching the size of a small submarine. These URVs are designed to be big in order to carry the

necessary power supply for long-range operations. Most of the URVs for military use are designed for mine hunting whereby multiple URVs are used to sweep an area for any underwater mines. This is to pave the way for any manned ship or submarine to reach an area. Some of the URVs are designed to accommodate variable payloads since the equipped torpedoes used to destroy mines or possible enemy vessels causes would a change in the vessel payload. Unfortunately, most of the advanced military URVs are still under research stage, incapable of carrying the advanced missions.

Research URVs

Research URVs are usually designed to be small and compact and are designed to be either in the class of ROVs or AUV. Being small and compact allows these URVs to be easily deployed when required. These URVs are also designed with configurable hardware to allow different set of sensors to be fitted to the URV as and when the sensors are required.

1.5 Navigational sensors for URVs

In any URV, the navigational system is a major component of the URV system. Even in a manned URV, navigational sensors provide information to the pilot to enable him to navigate the URV to his destination. In most URVs, the navigation system is responsible in tracking the motion of the URV, station keeping and obstacle avoidance. In all instances, the sensors used can be broken down into static state and dynamic state sensors where static state sensors outputs absolute values such as depth and angular displacement, while dynamic state sensors outputs dynamic values such as velocity and acceleration.

1.5.1 Dynamic state sensors

Dynamic state sensors are mainly used to determine a vehicle dynamic state. In a linear axis, dynamic sensors such as accelerometers and velocity Doppler are used to measure acceleration and velocity respectively. On the other hand, in a rotary axis, dynamic sensors such as gyroscopes are used to measure the rotation rates.

Accelerometers

Accelerometers, figure 1.4(a) are sensors that are capable of measuring a vehicle state of acceleration. Basic accelerometers consist of a mass attached with springs at the center of a rigid body. When a body accelerates, the inertia force of the mass resists the acceleration while the spring system induces force to the mass to move [13]. Measured over a calibrated range, the deflection can be used to determine the vehicle state of acceleration.

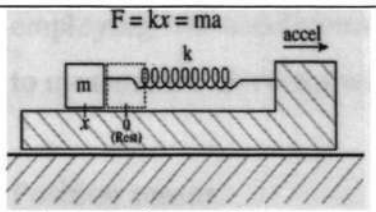
Accelerometers measure absolute acceleration in a particular axis. One drawback of the accelerometer is that the accelerometer measures the gravity (G) vector, a vector pointing towards the center of the earth with a value of $9.81 m/s^2$. Therefore, the gravity vector needs to be resolved and subtracted before it can be used to determine a body's acceleration. However, the inherent measurement of the G vector can be used in the measurement of a vehicle tilt condition. If the sensor is flat, the G vector can only be read in the Z-axis towards the earth. However, any tilting (roll or pitch) will result in partial resolving of the G vector in the X and Y-axis. Based on these resolved values, the tilt condition of the vehicle can be obtained.

Velocity Doppler

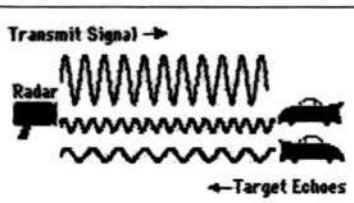
Velocity Doppler, figure 1.4(b), use the effect of Doppler shift in determining the velocity of a moving body. When a wave of specific frequency reflects from a stationary body, the reflected wave will be the same as the source wave. When the body is moving, the reflected wave will be of lower frequency if the body is moving away from the source and higher when it is moving towards the source. Based on this change of frequency the velocity of the system can be determined. A basic velocity Doppler is made of a transducer capable of transmitting and receiving a wave source. The difference between a land-based and water-based velocity Doppler is in the source frequency generated. Due to the difference in the fluid in which the waves travel, land based velocity Doppler uses frequency in the electro-magnetic (EM) wave or laser based Doppler, while water based velocity Doppler uses frequency in the ultrasonic range.

Fiber optic Gyroscope

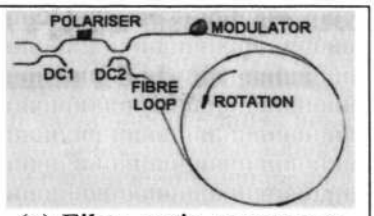
Gyroscope, figure 1.4(c) uses the precession effect, whereby a rotating disk will tend to have its axle point to a specific initial direction to determine the angular rate of a body. Fiber optic gyroscope is based upon a phenomenon, observed by Sagnac in 1913. Using a polariser and modulator, two coherent beams of light are sent around a ring in opposite directions. Since they travel the same length, they should be in phase when they return. When a body rotates, the gyroscope causes interference to the light pulses travelling in the fibre optic cable [14]. It shortens the path of one beam and lengthens the path of the other. The phase shift between the two light sources can be used to determine the rotation of the body. The resolved integrated rotation rate can be used to obtain the body's angular position.



(a) A Basic accelerometer principle using mass spring system



(b) Basic Velocity Doppler principle



(c) Fibre optic gyroscope principle

Figure 1.4 Principles of dynamic state sensors

1.5.2 Static state sensors

Static sensors are sensors designed to obtain the absolute location or position of a body. Unfortunately, most applications of sensors are applied in dynamic motion; the static sensors may have an inherent error that may result while in motion. However, static sensors are important in initialising a system and in station keeping conditions.

Ranging sensor

Ranging sensors commonly comes in three primary types, namely laser ranging, ultrasonic ranging, figure 1.5(a) and infrared ranging sensors. While ranging sensors are designed as static sensors, most application of ranging sensors is while they are in dynamic motion. One of the common applications of the ranging sensor is in mapping an environment with an autonomous or semi-autonomous vehicle.

Depth sensor

Depth sensor, figure 1.5(b) uses pressure difference in measuring the depth of a URV from the water surface. By knowing the pressure difference between the surface and the pressure at a specific depth, the depth of the URV can be converted from the pressure difference.

Angular sensor

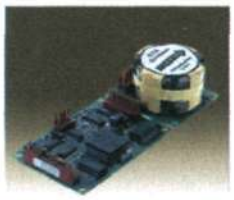
Angular sensors, figure 1.5(c) use the natural magnetic field along the earth surface in determining the angular states. Magnetic sensors aligned in the three primary axes can be used to determine a body's heading. Unfortunately, due to the earth magnetic fluctuations, magnetic sensors are not very accurate. Therefore, dynamic sensors are required to complement the angular sensors. Furthermore, a tilt sensor employing two accelerometer aligned perpendicularly in a planar manner are used to measure the G-vector whose angle can be used to determine a body tilt status.

Position sensor

The widely used position sensor is the Global Positioning System (GPS). The GPS comprise of 24 satellites orbiting around the earth at fixed orbits, constantly emitting positional signals. With a GPS receiver capable of intercepting the signals from three to four satellites, a body location on the earth in terms of longitude, latitude, and altitude can be triangulated.

(a) A ranging altimeter

(b) Depth sensor



(c) A magnetic compass

(d) GPS receiver

Fig 1.5 various types of static state sensors

1.5.3 Inertial measurement unit (IMU)

While the IMU is not a category of sensor, it is a black box worthy to be described. IMU is the primary sensor of an aided inertial navigation system (AINS) whereby non-inertial sensors described above are used to aid the navigation system. IMUs comprise of dynamic sensors or a combination of static and dynamic sensors.

IMU comprising of tri-axis accelerometers and gyroscopes

Common IMU comprise of three accelerometers and three gyroscopes to measure a body's dynamics in the six degree of freedom. The three accelerometers are aligned to the sensor's X-Y-Z axis to measure the acceleration and the three gyroscopes are aligned to measure the rotation rate corresponding to the three axis. Furthermore, some of these types of IMU combine the measurement of the G vector, obtained from the accelerometers, together with the high update rate of the gyroscope in order to obtain the tilt angles.

IMU comprising of six-axis accelerometers

IMU comprising of six accelerometers are also known as gyro-free IMU [15]. In this type of IMU, three of the accelerometers are aligned in the three primary axes while the other three are aligned midway through the primary axis. By resolving the acceleration of the all the accelerometers, the IMU is capable of measuring the acceleration and rotation rate of a body. Furthermore, by resolving the G-vector from all the accelerometers, the IMU is capable of generating the rotation angles of the body with respect to a fixed frame.

IMU comprising of tri-axis accelerometers, gyroscopes and magnetometer

The difference between these types of IMU and the first type of IMU is the addition of a tri-axis magnetometer. The tri-axis magnetometer measures the earth magnetic field in the three primary axes. Combined with the dynamic state of the gyroscopes and the G-vector obtained from the accelerometers, the IMU is capable of providing the three angular rotation of a body together with it dynamic states. While these types of IMU require less processing by the end-user, it tends to cost more than an accelerometer/gyroscope IMU due to its sensors and algorithms.

IMU does not only differ in its configuration but also in its operating range and inherent error. This is to accommodate usage in different conditions such as air, land and water based vehicle. While error is represented as a percentage of the operational range, the selection of an IMU for each vehicle should take into account this factor.

1.5.4 Low-cost High performance navigation sensors

In the interest of lowering the cost involved in a URV, the sensors used, especially the IMU, have a large impact on the cost of the system. In developing an AINS for a low cost URV, low cost sensors need to be used. However, the performance of the system cannot be compromise at the cost of the sensors.

IMU used in military applications require high precision and reliability. These IMU can cost more than US$100,000 as discussed by P. Luethi and T. Moser [61]. The cost of the IMU itself would cost more than a URV used for commercial applications. Furthermore, additional sensors are still required to aid the IMU. In the interest to keep cost low, Luethi P.[61] described the development of a low cost INS consisting of three gyroscopes, three accelerometers, a temperature sensor and a GPS costing less than US$4000. The developed INS achieved precision is sufficient for measuring the actual spatial representation. However, a minor misalignment in the sensors mounting may result in large deviation in the absolute position as described by E. Nebot and H. F. Durrant-Whyte[59].

Using commercial IMU reduce this problem since sensors are aligned by the manufacturers with great precision. While IMU can be classified by cost, a better way of classifying an IMU is based on its bias error since cost rise exponentially with higher precision. PHINS[32], a high precision INS initially designed for military purposes, have a gyroscope bias of 0.01 deg/hr and accelerometer bias of 0.5mG. Good low cost IMU can be classified with gyroscope bias of 10 deg/hr and accelerometer bias of 15 mG [59]. Therefore in implementing a INS system, the sensors needs to be aligned and calibrated properly and initialised before use and proper algorithms needs to be implemented to reduce possible errors.

1.6 Motivation

Underwater navigation plays an important part in a URV. While most underwater navigation centers on INS, GPS and visual aided system, there are many of the same sensors available in the market with different characteristics and performance. While selecting the best sensors available might give a reasonable navigation system for a URV, the cost imposed on the system might not be a viable option for its intended purpose. Presently, the immediate issue is to improve underwater intervention capabilities in terms of autonomy, cost-effectiveness, and performance [16].

For a URV to fit into cost effectiveness and performance, the URV needs to be designed based on its intended use. While large URVs are capable of accommodating larger sized personal computers (PCs) and a wide array of sensors, compact URVs requires the use of customised micro PCs with limited number of sensors. With the reduction in manufacturing cost for compact URVs, there is a need to compromise on the type and speed of the PCs and sensors that may drive the overall cost used. Therefore, the level of compromise should not get to a point that the commercial viability is lost due to the increased cost and reduced functionality.

As with any URV, there is a need to have a self-sufficient navigation system. Being compact do not mean that it should function any less than its intended designs. Therefore, even with a compact URV with specialised equipments, the URV still needs to function as a unit. The navigation system still needs to process the sensors data and make it available to other subsystem. The rest of the subsystem together with the navigational unit still needs to function as a complete URV. This provides a challenge in configuring the sensors and providing the algorithms to enable the navigation system to function as a unit and yet not too complex that it will affect the performance of the system.

1.7 Objectives

The objective of this project is to integrate a navigation system based on the Aided inertial navigation System (AINS) for a low-cost, compact twin-barrel URV designed by the Robotics Research Centre (RRC) in Nanyang Technological University (NTU).

1.8 Scope

The scope of the project will include:

- Designing a suitable software architecture for data acquisition and management in the URV onboard computer system and sensors
- Evaluate pre-processing algorithms suitable for the sensors
- Evaluate navigational algorithms suitable for the URV
- Testing of the navigational subsystem as a unit

1.9 Thesis organisation

Chapter 1 introduces the event that fuelled the development of URVs and look at the trends and development of the URV and its navigational system. Chapter 2 looks into related works in integrating a navigation system together in terms of sensors, computer system, operating system and software architectures in unmanned vehicles and how it compares with the URV.

Chapter 3 looks at the twin-barrel URV computer subsystem including the hardware, operating system (OS) and the software architecture designed to communicate with the sensors and organise the data for other subsystem. Chapter 4 looks into the navigational algorithms commonly used in underwater navigation system.

Chapter 5 and 6 looks at the results obtained from tests conducted in the software architectural design and navigational algorithms. Chapter 7 discusses the problems experienced in considering and selecting the suitable software architecture and algorithms for the system followed by the conclusion and future works.

CHAPTER TWO

LITERATURE REVIEW

Modern navigation uses a wide range of modern technological devices and complementary navigational techniques in determining the vehicle states. Most of the technology used depends on the navigational techniques, range, accuracy and environment in which the vehicle navigation is taking place. Early navigation dates back to ship captains sailing close to the shoreline using landmarks in determining the ship's location [17]. When the compass was discovered, sailors navigated using dead reckoning by mapping the ship's travelled distance and heading on a map at periodic intervals [18]. These two areas of navigation can be linked to the early form of absolute and referenced based navigation that are commonly used even today. However, as technology improves, the mundane task of navigation shifts away from humans to sensors processed by computers and microprocessors. Using the early navigation techniques, various sensors sets are designed for absolute referencing and dead reckoning. In processing the sensors data, various sensors with its complementary software processing techniques are used for data acquisition and processing.

2.1 Background

The subject for review can be broken down into three parts, namely navigation of an underwater robotics vehicle (URV), inertial navigation system as a tool for navigation (INS) and the computer architecture that will bind the two areas. In doing the project, these areas needs to be deeply explored in determining the techniques commonly used in the three areas and all its strength and weaknesses needs to be identified. This is necessary in proceeding with the project in order to implement the most suitable system for the URV.

2.2 Review of Navigation

Navigation involves the localisation of a vehicle or robot and determining its states, namely acceleration, velocity, position, angular rate and angular position in the three primary X-Y-Z axes. In determining, the states required in navigation, there are primarily two techniques, namely the absolute reference frame navigation and relative reference frame navigation.

2.2.1 Absolute reference frame navigation

Absolute reference frame navigation involves navigating in a known, mapped area. In these kinds of navigation, navigating techniques such as the global positioning system, terrain aided navigation, beacon aided navigation and pipe and cable tracking are developed.

Global positioning system (GPS)

The global positioning system uses the earth centered earth fixed frame navigation whereby the coordinates are fixed at the center of the earth with the Z-axis points to the north pole, X-axis intersects the plane is defined by the intersection of the prime meridian and the equatorial plane and the Y-axis is completed by the right hand orthogonal system [19]. In this system, twenty-four satellites orbiting the earth in six orbital planes, transmits two frequencies of wavelength 0.19m and 0.24m. A vehicle with a GPS receiver with a direct line of sight with at least four satellites is capable of determining its position on the surface of the earth in three-dimensional space.

While GPS are commonly used as a primary navigation sensor in land and air based vehicle, the application of GPS in underwater environment is limited in which, a URV GPS receiver will only have a direct line of sight with the satellites when it surfaces. Even with this limitation, underwater vehicles with GPS have been successfully implemented for long-range surveying and military missions URVs. These underwater vehicles such as the Autosub discussed by Griffiths et. Al. [20] uses the GPS to update the URV position at predetermined intervals. In order to overcome the problem of wavy waters, the antenna is fixed at the URV tail fin, a point slightly higher than the vehicle itself. Once it surfaces and updates its position, the URV will submerge utilising other navigational sensors such as the velocity Doppler and IMU.

The Seaglider, discussed by Charles et. Al. [21], extended the surfacing technique by periodically submerging and surfacing at 45° angle. This allows the vehicle to predict its probable surfacing position even before it surfaces. The Dorado [22] uses extendable antenna fixed on buoys to minimise the need to totally surface. This allows the URV to function under ice sheets where limited room may be available for the URV to surface. Wernli [23] presented another technique in whereby buoys fixed with a GPS data acquisition system is towed on the surface by a URV traveling underwater. The URV communicates with the GPS system using acoustics means in order to utilise the GPS resources.

Terrain aided navigation

Terrain aided navigation uses fixed key points in the terrain in assisting the navigation of a vehicle. With the coordinates of the fixed key points know, the position of the vehicle relative to the fixed key point can be approximated using ranging sensors such as ranging and scanning sonar [24] and stereo vision system [25]. One problem of the underwater terrain aided navigation is the lack of natural landmark to be used as referenced key points. Griffiths et. Al. [26] discussed on using rough sea terrain maps as navigational key points. This allows the vehicle to use predefined seabed maps as a comparison with the on going navigation.

Somajyoti et. Al. [27] explores the use of Map Building and Localisation for Underwater Navigation. The technique discussed the fusion of data from the vehicle vision system and scanning sonar in order to map the underwater terrain. Mapping the coloured, vision data over the scanning sonar map allows a more detailed profiling of the seabed and thus a more detailed key point system.

Pipe and cable tracking

One of the common means of navigation is the tracking of fixed man-made objects such as pipes and cables. Since the position of the cables and pipes in the underwater environments is known, a URV only needs to track the long line of pipes and cables in order to navigate to the desired destination. Marks [28] uses object tracking vision system as a tool for underwater navigation. The visual feedback from the URV allows the remote operator to navigate the URV over large pipes and cables. An alternative pipe tracking system presented by Amat [29] uses a combination of thermal imaging and a scanning sonar system. A thermal sensor detects the friction-induced heat in pipes while the scanning sonar maps the external profile of the pipe. The lack of a vision system allows the system to be built based on a low cost URV and yet perform the function of a vision-based pipe tracking system.

2.2.2 Relative reference frame navigation

Relative reference frame navigation typically uses dead reckoning in determining a vehicle position. In using dead reckoning, the inertial navigation system (INS) and velocity aided navigation is used in determining a vehicle position.

Dead reckoning

In dead reckoning [18], navigational frame is fixed on the surface of the earth. In underwater navigation, the frame is aligned in such a way that the Z-axis points towards the center of the earth, the X-axis is defined by pointing the axis towards the north pole and the Y-axis completes a right handed orthogonal system by pointing to the east. A localised fixed-frame is fixed at the point of the vehicle departure, with another frame fixed on the vehicle, known as the vehicle frame.

Since the vehicle is in constant motion, the vehicle frame constantly moves with the vehicle. In dead reckoning, onboard sensors periodically update the vehicle states. Updated vehicle states are referenced back to the previous states until the starting point, which is the local fixed frame. This is needed to resolve the positional states of the vehicle.

McPhail [30] discussed on building a simple, low-cost navigation system for an autonomous underwater vehicle (AUV). The primary systems used are the GPS and an IMU. The AUV is used for long-range scientific mission, thus the GPS is used to periodically correct the AUV position. However, once the AUV submerges, the INS is used to navigate the vehicle. Due to the large drift in the accelerometers, a low cost velocity Doppler and depth pressure sensors are used. In experiments conducted with the AUV, surfacing is only required to be done every fifty kilometres to minimise the predicted error over the AUV location.

Ferguson [31] presented the Theseus AUV, a long-range cable laying AUV capable of performing missions longer than five hundred kilometres. In one of its mission in 1996, it is required to navigate autonomously under ice over a range greater than 450KM. It uses a Honeywell 726 ring-laser-gyro INU and an EDO 3050 Doppler sonar. The INU provides heading and attitude data, while the Doppler measures forward and lateral ground speeds, as well as altitude. The sensor combination allows the AUV to navigate with an error of less than 1% of distance traveled.

Napolitano et. Al. [32] looked at the designing of a high performance IMU named PHINS, photonic inertial navigation system. The INS was the first to utilise the fiber optic gyroscope (FOG). By using high performances FOG with drift of less than 0.1°/hr and accelerometers with bias of less than 0.0005G and white noise of 0.0001G, the IMU is capable of unaided navigation with positional drift of less than three meters in a hundred seconds. When coupled with a velocity Doppler, it is capable of achieving drifts of less than three meter per hour. The only disadvantage of the INS is its cost and relatively large size and weight of 4096cm^3 and 3.8kg respectively.

2.3 Review of Inertial Navigation System (INS)

The inertial navigation system (INS) uses the IMU as the primary navigation sensor. As described in chapter one, the common IMU consist of tri-axial accelerometer and gyroscope and as described earlier, before the positional and orientation states can be obtained from the IMU, processing such as removal of errors and the resolving of navigational frame needs to done on the sensors data. In designing the PHINS, Napolitano et. Al. [32] discussed the model shown in figure 2.1 in processing the data to obtain the positional and orientation states.

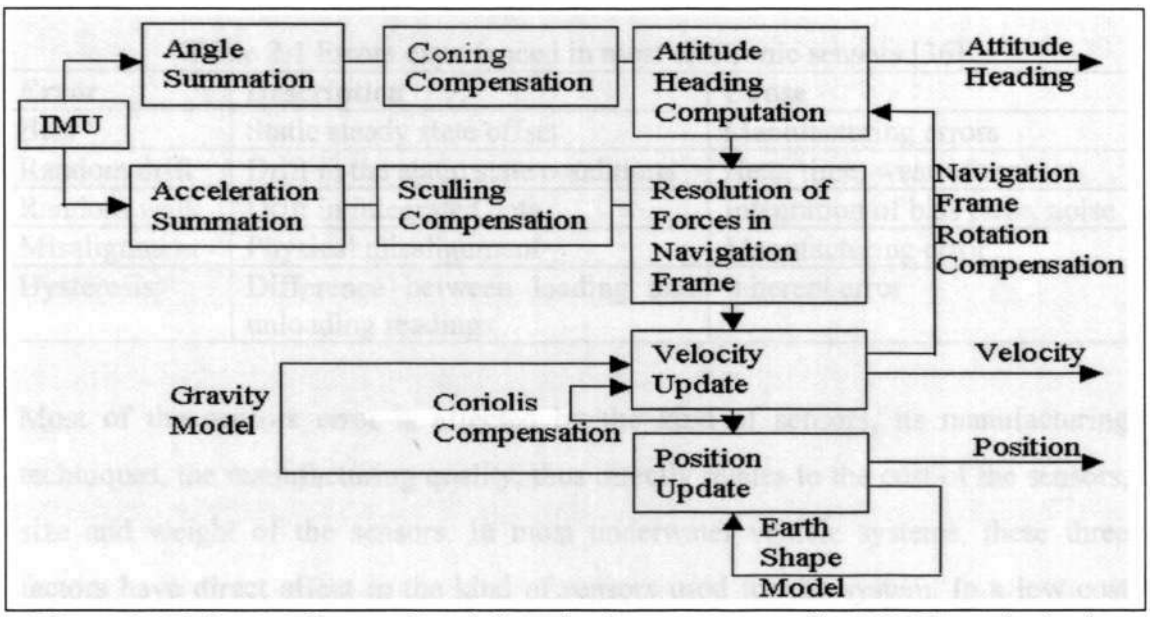


Figure 2.1 Scheme of a pure inertial navigation system used in PHINS to obtain the various states of a vehicle [32].

The model shows an IMU integrated to obtain its angular position and velocity. The result of direct cosine matrix transformation results in the coring and sculling errors, which is described in [33]. Therefore, these errors need to be compensated in order to obtain a more accurate orientation and velocity. Based on the obtained orientation, the velocity is resolved to the navigational frame before further processing is done. The gravity model discussed in [34] is used to compensate the difference in the actual gravity and accelerations resulting from deflections in the sensors. The earth shape model maps the navigation frame to the shape of the earth. Depending on the range of operation, simpler earth shape model can be used on shorter-range operations.

In INS, the IMU is only capable of determining the vehicle dynamic states. Therefore, there is a need to initialise the sensors. Static state sensors such as tilt sensors, compass, pressure sensor and the GPS are used to initialise the vehicle before deployment and during stationary conditions. Besides being used for initialisation, Giaffe [35] discusses the use of these sensors as a redundant array. Since multiple sensors are used to determine certain states, should the primary sensor fails, the redundant sensors takes over the primary sensor's role. Even so, one of the primary uses of redundancy is in reducing errors. Table 2.1 shows the various errors experienced by electronic sensors [36].

Table 2.1 Errors experienced in most electronic sensors [36]

Error	**Description**	**Cause**
Bias	Static steady state offset	Manufacturing errors
Random drift	Drift in the static state conditions	Heat, time, wear of sensors
Random walk	Drift in integrated data	Integration of bias error, noise
Misalignment	Physical misalignment	Manufacturing error
Hysteresis	Difference between loading and unloading readings	Inherent error

Most of the sensors error is affected by the kind of sensors, its manufacturing techniques, the manufacturing quality, thus directly relates to the cost of the sensors, size and weight of the sensors. In most underwater vehicle systems, these three factors have direct affect in the kind of sensors used for the system. In a low cost navigation system, to fit the budget constraint, the sensors used tend to have higher error range. Therefore, data from different sensors needs to be fused in order to reduce the overall error.

In IMU, due to the need of integrating the dynamic states values, minor drifts in the dynamic states will result in large accumulative drifts in the static states. Therefore, data from velocity Doppler and static state sensors are commonly used to be fused with data from the IMU. Napolitano et. Al. [32] shows the primary technique in fusing data from various sensors with the IMU. This is shown in figure 2.2 below.

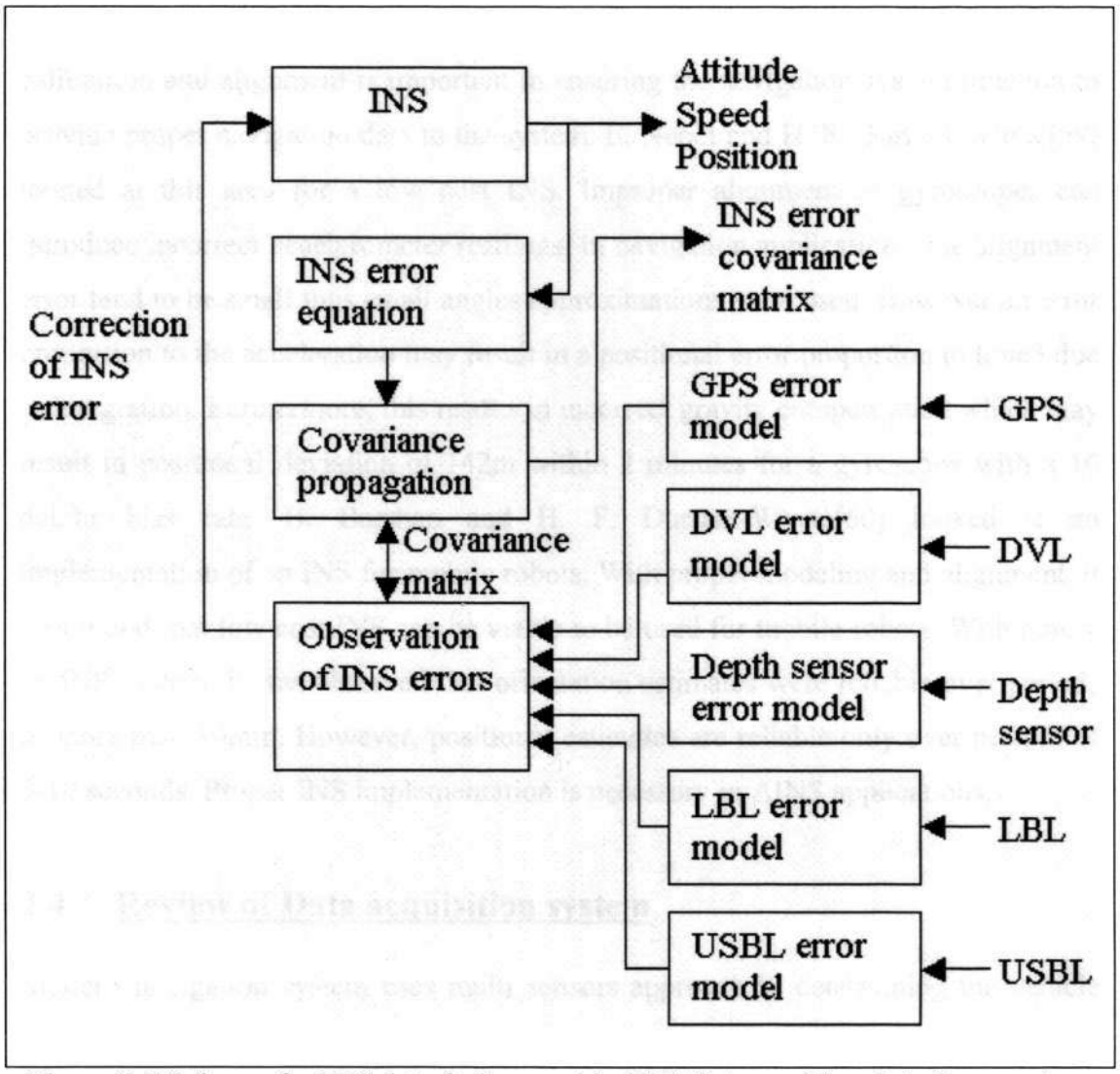


Figure 2.2 Scheme for INS data fusion used in PHINS to combine data from various sensors with that of IMU to improve tracking accuracy [32]

In the scheme shown in figure 2.2, non-inertial sensors are used to aid the INS forming an AINS. The INS error model is derived from experimentation and presented as the INS error equations. Data from various sensors are obtained to be used in the data fusion. Using the sensors error model, the error covariance can be deduced from experimentation. Based on the values obtained from each sensor together with the experimentally obtained error covariance, the data can be filtered in order to obtain a cleaner data with smaller overall error covariance. The corrected states are then used to correct the INS sensor model in order to determine the vehicle states with higher precision.

With errors being a fundamental part of an inertial navigation system, proper calibration and alignment is important in ensuring the navigation system function to provide proper navigation data to the system. E. Nebot and H. F. Durrant-Whyte[59] looked at this area for a low cost INS. Improper alignment of gyroscopes can introduce incorrect accelerometer readings. In navigation applications, the alignment error tend to be small thus small angles approximations were used. However an error proportion to the acceleration may result in a positional error proportion to time3 due to integration. Furthermore, this results in incorrect gravity compensation which may result in positional deviation of 142m within 2 minutes for a gyroscope with a 10 deg/hr bias rate. B. Barshan and H. F. Durrant-Whyte[60] looked at an implementation of an INS for mobile robots. With proper modeling and alignment, it was found that low cost INS can be viable to be used for mobile robots. With purely an INS system, it was observed that orientation estimates were reliable over periods of more than 10min. However, positional estimates are reliable only over periods of 5-10 seconds. Proper INS implementation is necessary in AINS applications.

2.4 Review of Data acquisition system

Modern navigation system uses multi sensors approach in determining the vehicle states. In most systems, one or two computer systems are used in servicing the data from the sensors. In such systems, the computer system needs to do timely data acquisition of the onboard sensors and perform other processing requirements of the system. A real time system is required in servicing the sensors in order to meet the timely needs of the system.

Real time systems

Kopertz [37] explores the design principles for real time systems. Most real time systems can be separated into two categories, the soft-real time system and the hard-real time system. In all real time systems, the timely acquisition of data is a critical factor. However, in hard real time systems, a missed or delayed deadline for data may lead to catastrophic consequences in the system. In soft real time system, a missed or delayed deadline will lead to significant loss to the system but the system

is still able to function as required. Table 2.2 shows the common difference between a hard and soft real time system.

Table 2.2 Differences in hard and soft real time system [38]

Characteristic	**Hard real time**	**Soft real time**
Response time	Hard required	Soft desired
Peak load performance	Predictable	Degraded
Control of pace	Environment	Computer
Safety	Often Critical	Non-critical
Size of data files	Small/Medium	Large
Redundancy type	Active	Checkpoint-recovery
Data integrity	Short-term	Long-term
Error detection	Autonomous	User assisted

The table shows that in hard real time systems, sensors control the data output to the computer, which have to meet the sensors requirements even in peak load performance in order to perform predictably. In order to meet these needs, the data size is often small with only short-term data integrity. Any data corrections are done without the user intervention. The requirement in meeting the timing needs often affects the system in a critical manner.

Hard real time underwater vehicle system

In the area of underwater vehicle, there are a few systems designed to meet the hard real time constraints for the onboard computer. Biharu, McGhee, Luqi and Yee [39] looked at the rapid prototyping of a real time operating system for an underwater vehicle. The operating system is designed to meet hard real time constraint for the military NPS AUV. The hard real time system is designed for the embedded system onboard the AUV. The AUV is designed to perform mine hunting and counter measure during war and peacetime missions. Due to the mission risk involved, the AUV needs to service the sensors at real time, perform motion and higher-level task planning without any possibility of system degradation.

Vestgard [40] discusses in his research on the implementation of hard real time system for the Hugin 3000 AUV, an AUV designed for Deep-water surveying. Hugin was designed for high payload, sensor data quality, high positional accuracy

high survey speed and quick heading changes. Its primary sensors for navigation are the GPS, INS and velocity Doppler. In achieving the hard real time constraint for the system, three processors are used in the system to handle Control, Navigation and Payload. The processor systems are networked via fast Ethernet for inter-PC communication. In the designed architecture, the navigational processor is designed to process only the data from the IMU while the control processor is tasked to handle the supporting sensors such as the velocity Doppler and the GPS and also perform the high-level control tasks.

As a survey AUV, Hugin needs to perform seabed mapping and log the high quality data on its onboard data logging system. The payload processor is tasked to perform these surveying needs. The onboard surveying sensors such as the multi-beam echo sounder, side scanning sonar, sub-bottom profiler, fishery research echo sounder and others are controlled by the payload processor to minimise the effect on the control and navigation system. The payload processor is also tasked to perform the logging of all the data obtained from the surveying sensors to the onboard drive. It also performs filtering of the data and plots the digital terrain model of the seabed.

Soft real time underwater vehicle system

In the use of real time system for an underwater vehicle, most vehicles are designed to only meet the soft real time needs of the system. In most soft real time systems, commercial operating systems are usually used, as when combined with a suitable hardware, it is capable of meeting the soft real time constraints. Microsoft windows operating system (OS), a common operating system for personal computers is also used with onboard URV computer system. Rosenblatt [41] presented in his research, the use of Windows NT operating system in performing soft real time operations for the Oberon underwater vehicle. It uses sonar and vision based navigation system to follow an intended course, maintaining a specific height above the seabed and avoiding obstacles. The main design purpose of the URV is for geographical surveying and surveillance. With a goal driven architecture, the computer system with the Windows NT operating system is sufficient for the system to response in real time to the environment.

While Windows NT operating system may not be the most suitable operating system to perform soft real time tasks, other operating systems are designed with specific constraints just to perform real time tasks. VXworks, a real time operating system is used in the Kambara described by Silpa-Anan [42]. Running on a single PowerPC 233 MHz processor, the Kambara is capable of processing the navigational system based on the INS together with the higher-level control tasks and thrusters control. The Kambara is a remotely operated system, designed to perform inspections with the onboard video signals are directly feedback to the operator on a remote computer.

Underwater vehicle system and data sharing architecture

In all underwater vehicle systems, data from individual sensors needs to be shared in order to perform post processing on the data. In certain systems such as the Hugins AUV presented by Vestgard [40] uses multiple processors to specifically handle specific tasks. For such systems to function properly, data may need to be shared among the processors in order to perform a more accurate control, navigation or surveying. Various techniques are developed and used just to enable these requirements for data processing onboard the URV.

The onboard inter-processor communication proposed by Vestgard [24] uses TCP/IP Ethernet communication for Hugins three processor system. It uses a commercial Common Object Request Broker Architecture (CORBA) protocol for its inter-processor communication. Its object-oriented design uses a flexible publish/subscribe communication pattern with event and error reporting. This system is critical in meeting its hard real-time and data logging requirements for its autonomous survey missions.

Williams [43] describes single-processor architecture for the Oberon ROV using the Windows NT operating system, shown in figure 2.3. In the proposed architecture, all the sensors are connected via serial RS232 or via the PCI bus using an analogue to digital converter. The tethered connection uses Ethernet for remote communication and video signals are directly feedback to the remote computer via a coaxial cable. It

communicates between computers via TCP/IP using, using Microsoft message passing protocols. A central communication hub is used to route all the sensor messages to the mission planner level where all the data are processed.

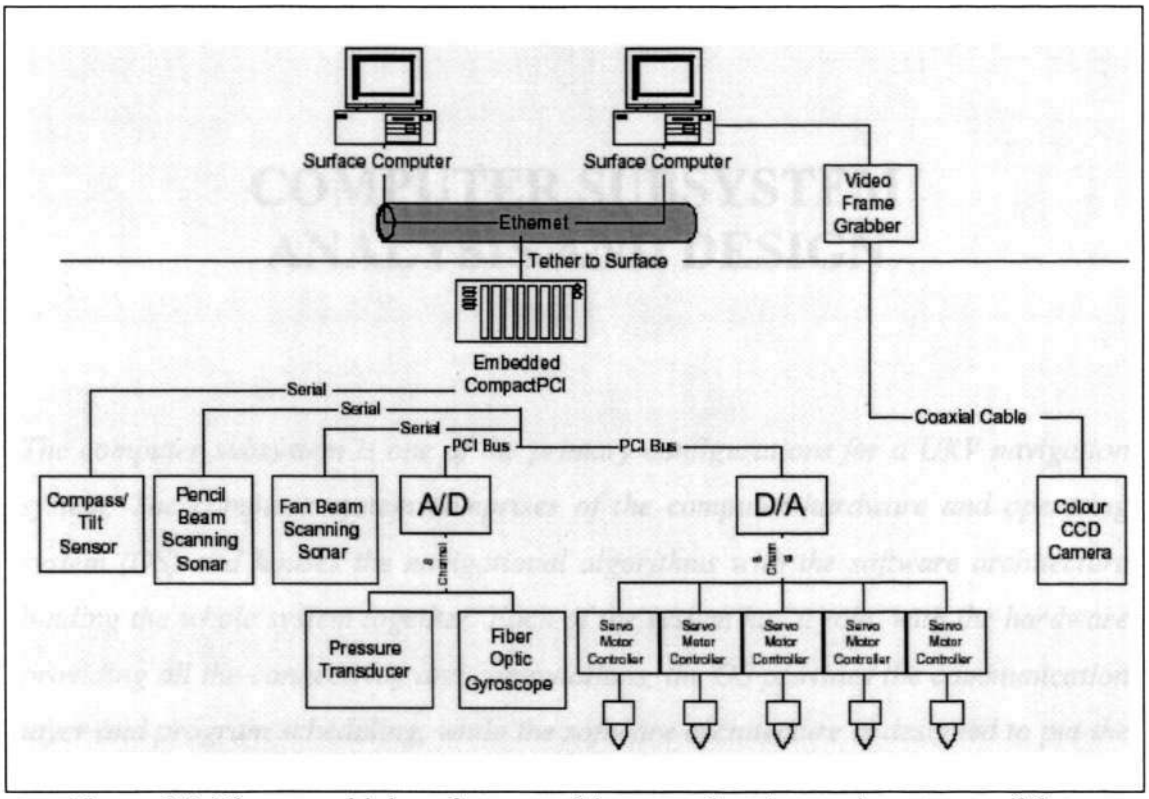


Figure 2.3 Oberon vehicle software architecture showing various types of data acquisition hardware used on a single system [43]

2.5 Chapter summary

The design of a vehicle navigation system had taken many turns through out the history. Due to the approximate spherical shape of the earth, different earth shape model have been developed for different range and accuracy of mapping the navigational environment. The suitability of each earth shape model is highly dependent on the vehicle design and its intended use. A vehicle based on the aided inertial navigation system (INS) as its primary navigation system may require a wide range of static state sensors in order to initialise and update the vehicle states forming an aided inertial navigation system (AINS). Furthermore, advanced data acquisition and processing techniques are required in servicing the sensors and navigation system in a timely manner.

CHAPTER THREE

COMPUTER SUBSYSTEM ANALYSIS AND DESIGN

The computer subsystem is one of the primary configurations for a URV navigation system. The computer system comprises of the computer hardware and operating system (OS) and houses the navigational algorithms with the software architecture binding the whole system together. Each of the system has a role, with the hardware providing all the connectivity and computations, the OS provides the communication layer and program scheduling, while the software architecture is designed to put the system as a nice complete package[44]. Therefore, the design of the computer architecture is critical since it needs to work seamlessly with the sensors and other subsystems and needs to be optimised for the system. The architecture should function as a black box with only the input and the output will matter to the rest of the system[45]. In doing so, the architecture needs to meet the timely needs of the sensors and other subsystem while doing all the necessary navigational processing. Furthermore, since system failure is not a possible option, the architecture needs to be robust and yet implement fail-safe mechanism to ensure redundancy in determining the system states.

3.1 Background

In modern navigation systems, sensors are designed to process all analogue data from the sensors and output it in digital form. While most commercial sensors are designed to communicate using the RS232 serial protocol, the baud rate, data size, data type may differ according to the sensor designer specifications [46]. Therefore, the computer subsystem is responsible in reading and managing these data sets.

3.2 Hardware architecture design

With a small hull, a micro personal computer (PC) is used to service the navigation system. However, with a slower processing speed, QNX 4.25, a real time operating system was selected to schedule and coordinate the programs and processes.

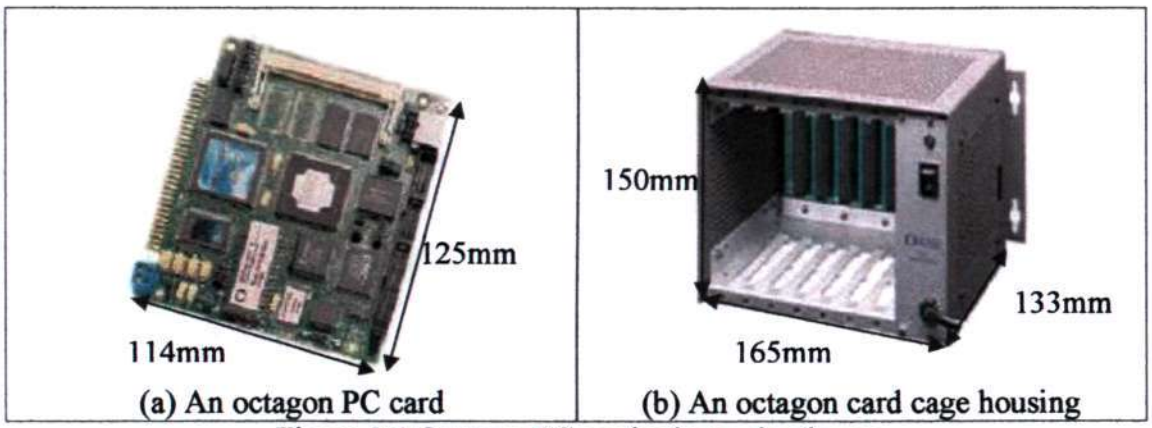
3.2.1 Computer hardware

The PC used for the twin-barrel URV navigation system is a computer system from Octagon System Corporations based on an AMD 5x86 architecture with a 133 MHz processor. It is designed based on PC cards measuring 125mm in width and 114mm in height, connected via an ISA-bus backplane housed in a card cage measuring 163mm in length by 150mm in width by 133mm in height. It is powered either from a power supply card or via a power supply connecter with a ±5V power supply [47].

Table 3.1 Octagon micro PC cards used in the twin-barrel URV system

PC card	Description
3065 backplane card	An ISA bus backplane card consist of six expansion slots used to connect the micro PC cards together into a computer system. Since it is based on the ISA-bus architecture, it supports only eight-interrupt request (IRQ) thus requires IRQ mapping between the 5066 PC card, which supports internal PCI architecture with 16 IRQs.
5066 CPU card	A CPU card embedded with a 133Mhz processor. The card comes with 2MB RAM, 2Mb Flash ROM for OS installation and storage of customised software. For expansion, it comes with a ram expansion slot, two serial RS232/RS485 ports, a single parallel port, and a PS2 keyboard connector.
5558 8-port serial card	An eight port serial card used to expand the system serial connectivity. It supports both RS232 and RS485 serial protocol with speeds up to 115Kb/s. Due to the IRQ limitations, the 5558 card utilise two interrupts to be shared with all eight serial ports.

PC card	Description
5500 network card	A network card supporting both the RJ45 and the coaxial Ethernet connection. The card supports the Ethernet specifications of 10Mbps baud rate with cables capable of reaching 100m in length. Supporting advanced programming techniques, the 5500 card is a more suitable for inter-PC communication compared to the serial connection.
5800A external storage card	An external storage card capable of supporting a floppy disk drive (FDD) and hard disk drive (HDD) supporting capacity up to 512MB. While the 5066 flash rom is capable of supporting embedded QNX OS, an external HDD is capable of supporting a more complete OS with complier for software design purposes. Furthermore, an external storage is required for onboard data logging for experiments.
Unigen IFM-441 flash drive	A 96MB flash module used in conjunction with the 5800A storage card. Unlike normal HDD, the flash module does not have any mechanical parts thus suitable to be used in an environment such as the URV. Even so, the flash drive have a shorter life span with only about 500 thousand times rewriting before the flash drive fails.
5420 VGA card	A display card for displaying a display console. Even though the micro PC is capable of operating without any user intervention, the display card is necessary for use in directly observing the operation of the software running onboard the micro PC during software testing.



(a) An octagon PC card (b) An octagon card cage housing

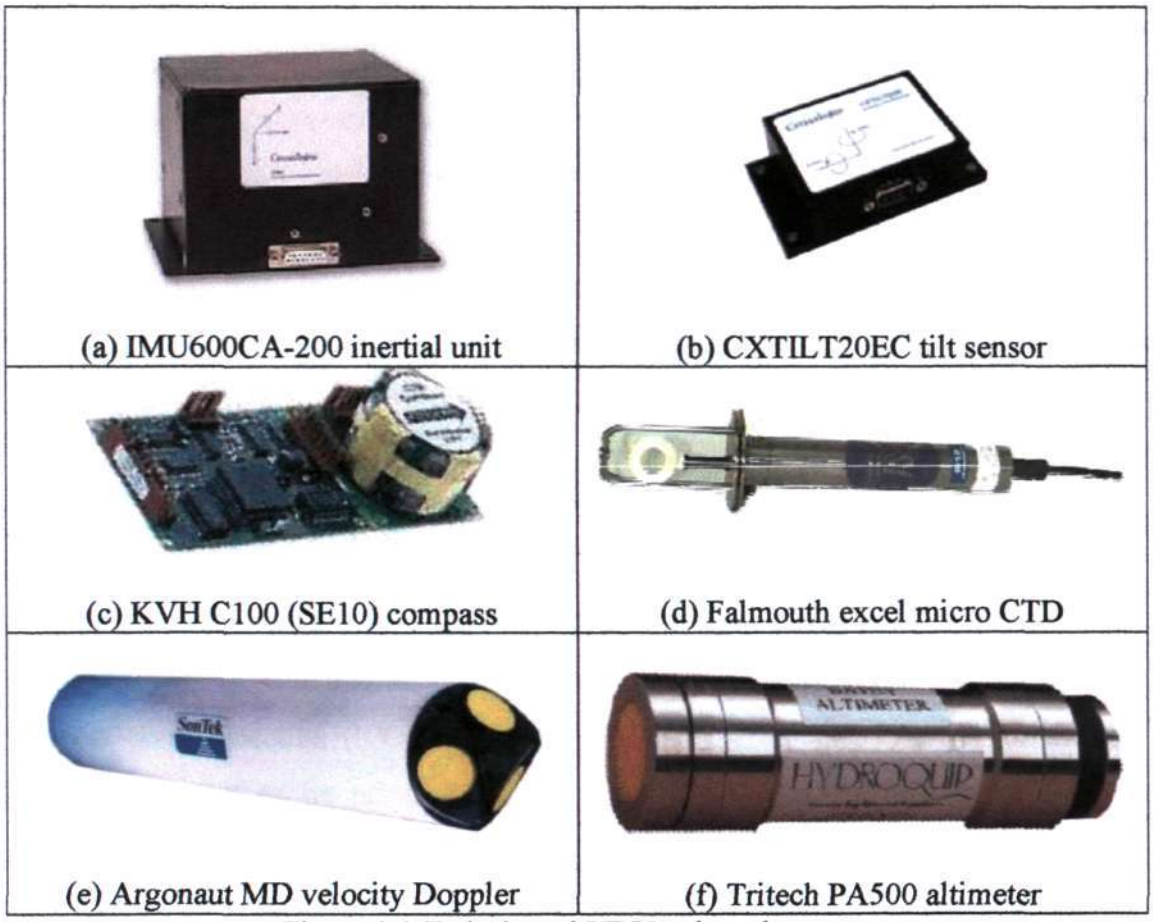
Figure 3.1 Octagon PC card primary hardware

3.2.2 Sensor hardware

The twin-barrel URV sensor system is based on an aided inertial navigation system (AINS). The onboard sensors are connected to the computer system via RS232 serial interface. Since each sensor outputs different types of data at different baud rate, each of the sensor specifications needs to be explored before the system architecture can be designed for the system. Primary output specifications such as the output mode such as continuous stream or polling, the output baud rate; the output format, the data size and the update rate need to be explored for the architecture design.

Table 3.2 Summary of the twin-barrel URV sensors data specifications (Appendix B)

Sensor	Output mode	Baud rate (bps)	Output format	Data size (Bytes)	Update rate (Hz)
IMU600CA-200 IMU	Polling	38400	BCD	22	>80
CXTILT02EC tilt sensor	Polling	9600	ASCII	8	1
KVH C100 (SE10) compass	Polling	9600	ASCII	16	1
Falmouth excel micro CTD pressure sensor	Polling	9600	ASCII	27	6
Argonaut MD velocity Doppler	Polling	57600	ASCII	54	3
Tritech PA500 altimeter	Continuous	9600	ASCII	9	10



(a) IMU600CA-200 inertial unit (b) CXTILT20EC tilt sensor

(c) KVH C100 (SE10) compass (d) Falmouth excel micro CTD

(e) Argonaut MD velocity Doppler (f) Tritech PA500 altimeter

Figure 3.2 Twin-barrel URV onboard sensors

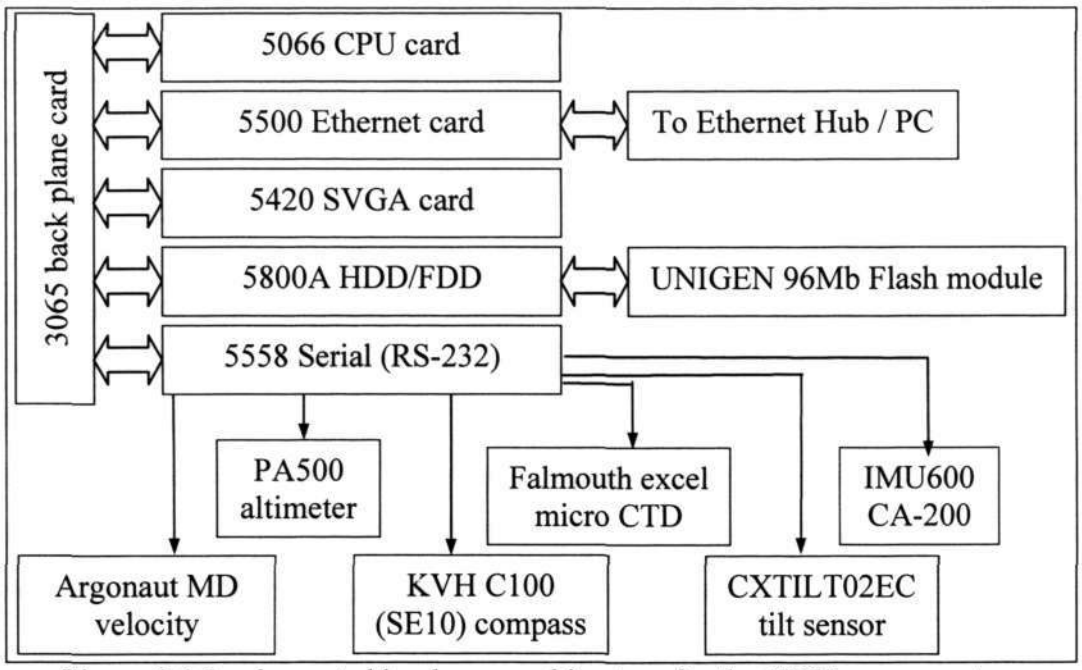


Figure 3.3 Implemented hardware architecture for the URV sensor system

3.2.3 Computer operating system

The main advantage of using a standard PC hardware is the availability of commercial operating systems, thus reducing the need to perform low-level tasks required in coding commonly performed algorithms. Even so, a majority of OS, such as Microsoft Windows is not designed for real time usage. However, as studied in the literature review, OS such as QNX, real time Linux and VXworks are capable of processing programs in real time. In the twin-barrel URV system, a QNX system was used as an interface layer between the hardware and the software [48].

QNX 4.25 Operating system (OS)

QNX 4.25 was selected, as it is a real time operating system designed to manage processes requiring stringent timings and scheduling requirements. It is designed to be small and customisable. It is capable of functioning only with a microkernel less than 8KB in size, handling all the essential tasks of program scheduling and inter-process communication (IPC) in a multi process environment. QNX incorporates message passing as its primary IPC. This requires a process to send, while a receiving process receive it and reply to the sender process.

The OS can be customised to add additional functionality to the system. The process manager, network manager, device manager and file system manager files and drivers can be added and removed when required. This allow the OS to be customised according to the space constrain of the system it resides in.

QNX supports multiple mode of scheduling, namely the first in first out (FIFO), round robin and adaptive scheduling. In a multi process environment, the round robin scheduling manages the processes by allocating fixed time before another process takes over. FIFO scheduling allows a process to enter a wait state before another process takes over. The adaptive scheduling functions as the round robin scheduling except that a process priority is reduced once it enter a wait state.

QNX network (Qnet)

QNX supports the standard TCP/IP network protocol for connecting to a large computer network or Qnet for QNX native network. While TCP/IP is a commonly used network protocol capable of supporting large network, it's inter-network connectivity needs multiple headers to be attached to the data stream to bridge the differences in network and hardware and thus affects the transmission rate. Qnet on the other hand only functions between QNX systems therefore the overhead caused by data headers is minimised, thus increase the throughput of the network.

Watcom C/C++ compiler

QNX supports the C/C++ language as its primary programming language. It supports ANSI and POSIX real time extension for real time applications. As a commonly used programming language, less time will be required to write the necessary software.

Twin-barrel URV bootup architecture design

Since the system can undergo boot up from either the flash drive, the flash ROM or enter the system BIOS, the system was designed to be usable under the different mode of bootup. Furthermore, since the system could function with or without the display card, the boot up architecture as shown in figure 3.3 takes all these into account for different kind of system boot up.

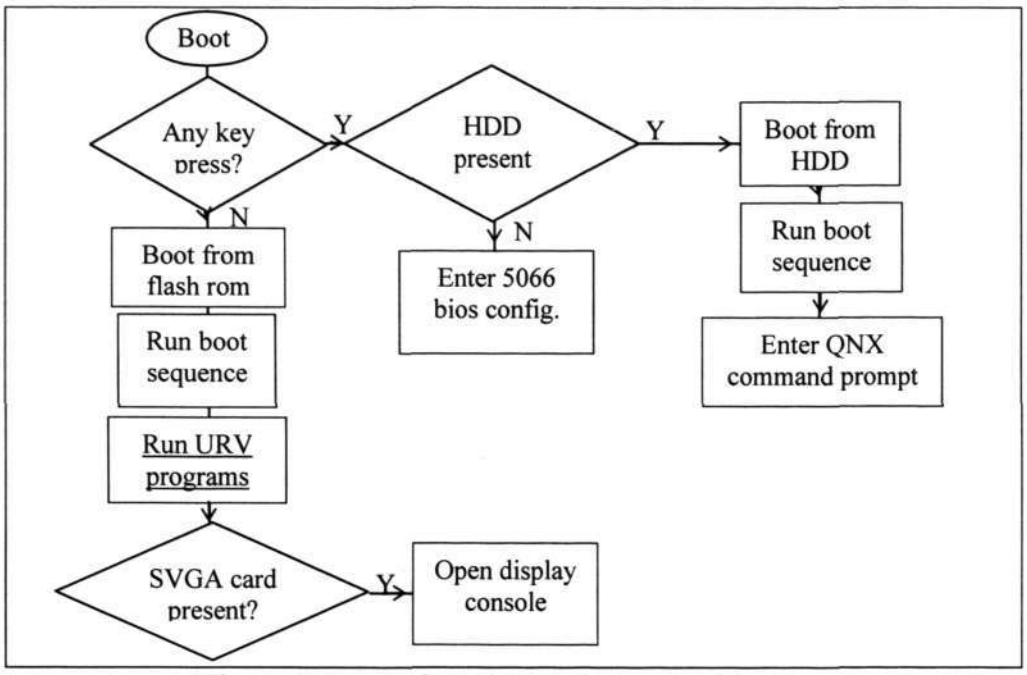


Figure 3.4 Twin-barrel URV bootup architecture

3.3 Primary (internal) software architecture design

The primary software architecture was designed to handle all the internal processing required within the navigational pod. Based on the sensors characteristics and the computer hardware capabilities, the primary architecture was designed to function optimally in the system. Serial server architecture was designed to periodically service all the sensors and the data is placed in shared memory architecture for sharing of the sensors data. A data logging architecture was designed to log essential experiments data to the flash module for future analysis.

3.3.1 Serial server architecture design

The serial server architecture written in C language as in Appendix A is a data acquisition program designed to obtain data periodically from the sensors. Using the multi processes capabilities of QNX 4.25, the serial server program spawn individual sensor program that will handle data acquisition for each of the sensors.

Since all the sensors are based on the RS232 serial specifications, the data acquisition routine for each of the sensors are designed to be modular. Modularity improves the easiness of the software design and future editing. The data acquisition

software for each of the sensor differs only in the baud rate, data length, data formatting and the acquisition rate. Using a multi-process design allow individual software to activate according to the output rate of the sensors and thus maximise the processing capabilities of the computer system.

During the interrogation process with the sensors, the CPU may have to wait for the sensor to reply. Utilising a combination polling and interrupt routine, the computer will poll the sensors, enter the interrupt mode and change to a wait state. During this time, the CPU is released to handle other tasks while waiting for the sensors data. Once the complete data is available, it interrupts the CPU to handle extract the data and continue with the process. This further improves the optimality of the software.

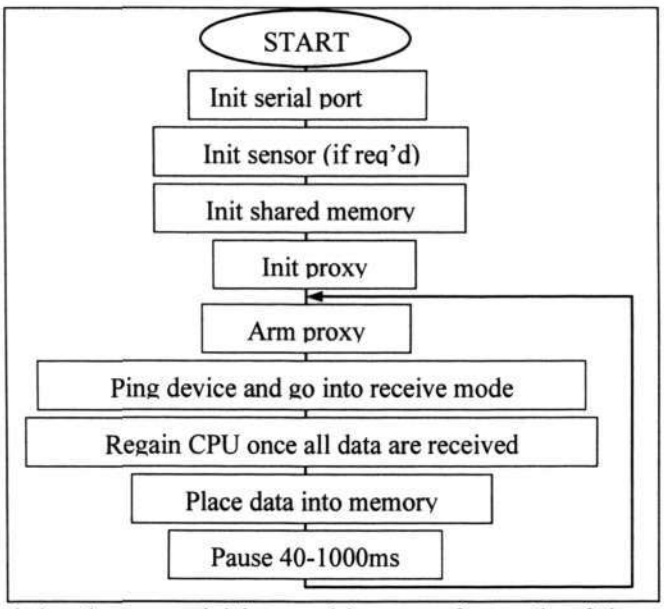


Figure 3.5 Modular data acquisition architecture for each of the sensor software

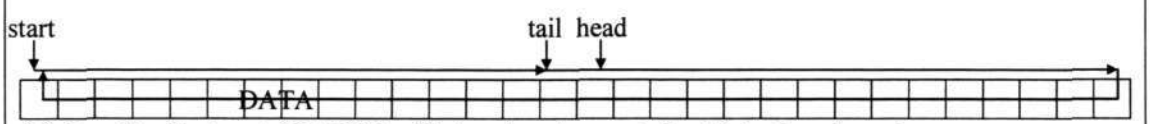
3.3.2 Shared memory architecture design

The shared memory architecture is an architecture designed to manage data retrieved from the sensors. In this architecture, the serial server reserves a memory region within the RAM, which is then partitioned to be accessed by different sensors. The serial server has the access to all the partitions while the rest of the serial software is allowed to read and write only in the individual partitions. Other processes that require access to the data obtain the permission from the serial server process.

To ensure the data conforms to the same data type, all the data are converted to the float data type before being placed in the serial server. This allows further processing of the data to use the standardised data type and thus reduce error that may result in handling different data type. A buffering algorithm was also implemented within the shared memory architecture that would allow the retention of the last few sets of data. Using last in first out buffering (LIFO), it ensures any access to the data will always retrieve the latest data. The buffering algorithm is required by filtering and data analysis processes.

Configuration flags	Unsigned short data type, each data set allocated with 10 flags
DMU data	Float data type, twenty-two data per set
Altimeter data	Float data type, one data per set
Pressure sensor data	Float data type, one data per set
Compass data	Float data type, one data per set
Scanning Sonar data	Float data type, variable data per set
Tilt sensor	Float data type, two data per set
Velocity Doppler	Float data type, eight data per set
Combined data	Ax, Ay, Az, Vx, Vy, Vz, Px, Py, Pz, Rr, Pr, Yr, R, P, Y, Time stamp (float)

(a) Shared memory architecture for the twin-barrel URV



(b) Last in first out (LIFO) buffering implemented within the shared memory

Figure 3.6 Shared memory data management architecture

3.3.3 Data logging architecture design

The data logging architecture is used when the sensors data is required to be logged during experiments for future analysis. In the architecture, files between a system reset are increased by a hundred while files within a system reset are incremented by one every time a logging signal is sent to the process. The signal for starting and ending a data log is sent from the surface PC to the control pod and then through the Ethernet connection to the navigational pod. Such a file numbering design allows the differentiation in the files when multiple experiments are conducted within a short period of time.

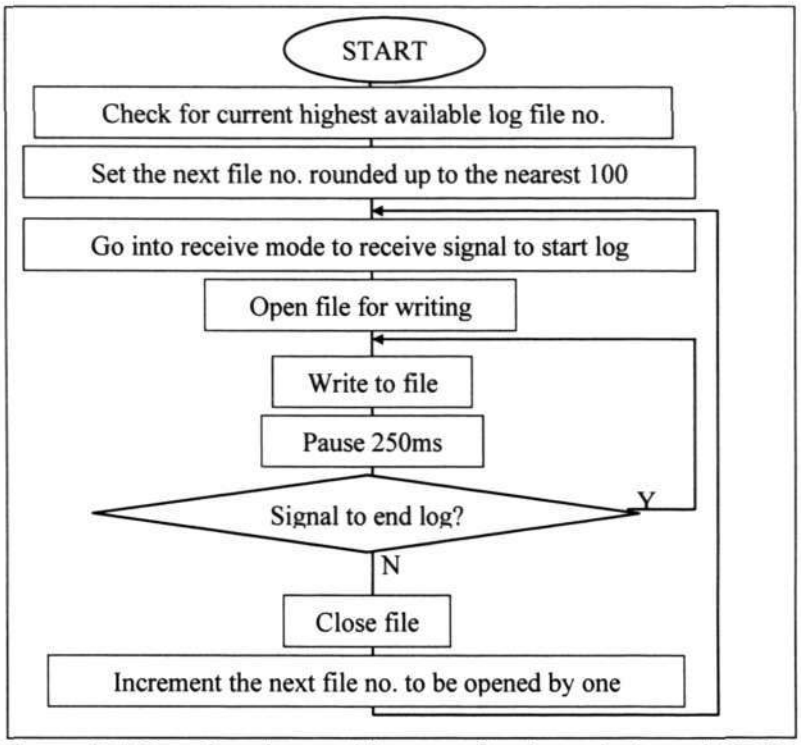


Figure 3.7 Data logging architecture for the twin-barrel URV

3.4 Secondary (external) software architecture design

The primary software architecture handles all the sensor data acquisition and data management. However, the twin-barrel URV requires data to be transferred from one PC node to another. Furthermore, under different modes of testing, the URV connection may need to change, such as during software testing where multiple accesses is required but only a single tether can be used during pool experiments. Therefore, the secondary architecture was designed to overcome these issues.

3.4.1 Inter-PC networking

The secondary software architecture centres on the QNX network, Qnet. The inter PC communication that takes place in the system all uses the QNX. As stated in section 3.2.3, QNX support its native network Qnet over the Ethernet protocol. Through the 5500 Ethernet card, the QNX network can be established for the URV system. Furthermore, since the Ethernet network supports point-to-point access and connection through a hub, the secondary software architecture can be established.

Shared resource as a network feature

One problem with having multiple PC is accessing hardware and software between PCs. Furthermore, uneven load distribution may lead to poor performance of the system. Therefore, the system was designed to allow shared resources between multiple PC. This would allow testing of hardware and software from remote PCs, logging of data on a single PC and synchronisation of the system to allow it to function as a single unit. The shared resources allocate specific task to specific PC thus reducing the need to perform similar operations for all the PCs.

Passing of critical data using message passing

In every operation, certain data such as command instructions and signals needs to be given higher priority and may require handshaking to ensure data are correctly transmitted form one PC to another. In QNX, the message passing feature allows such an operation to be carried out. In this architecture, a virtual circuit layer is established between the two PCs. When a signal or command is sent to the navigation PC, the system will send back a validation signal to validate the data sent. In case of miscommunication, the data will be resent back before the system will proceed with its normal tasks. However, such a data transfer may not be suitable for rapid data transfer. Therefore, another means are required to transfer sensor data between PCs.

Passing bulk data using message queue

The message queue is capable of passing messages through the Qnet via a queue object. The queue object needs to be configured for the length of data and the queue depth before it can be used. The queue depth is buffered using last in first out (LIFO) system to ensure the latest data are always received from the receiver node. Unlike message passing, message queue utilise one-way communication. One advantage is that the message passing can be done asynchronously. Furthermore, there is no time lost in waiting for a reply message, which is required in message passing. However, without any form of handshaking, the message queue may not be suitable for sending critical data.

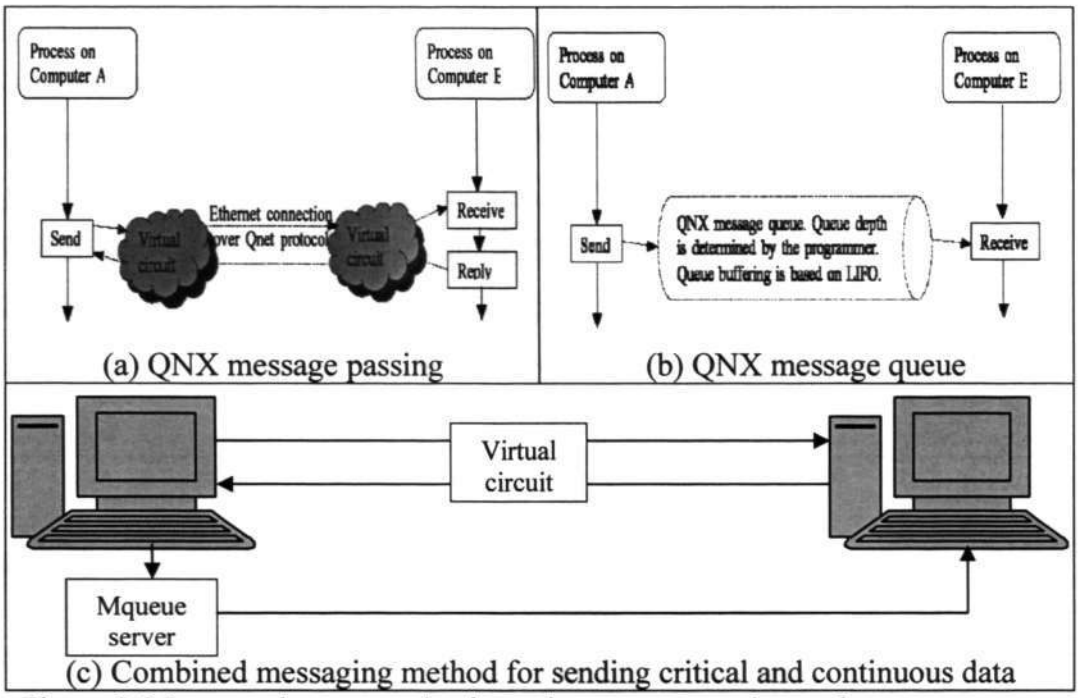


Figure 3.8 Inter-node communications via message passing and message queue

3.4.2 Applied system architecture

In conducting experiments and testing program codes and algorithms, different types of connectivity is required for different test conditions. Furthermore, the primary software architecture may need to work differently under the different conditions. Therefore, modifications were made to the primary architecture to support the secondary software architecture.

Primary tethered configuration

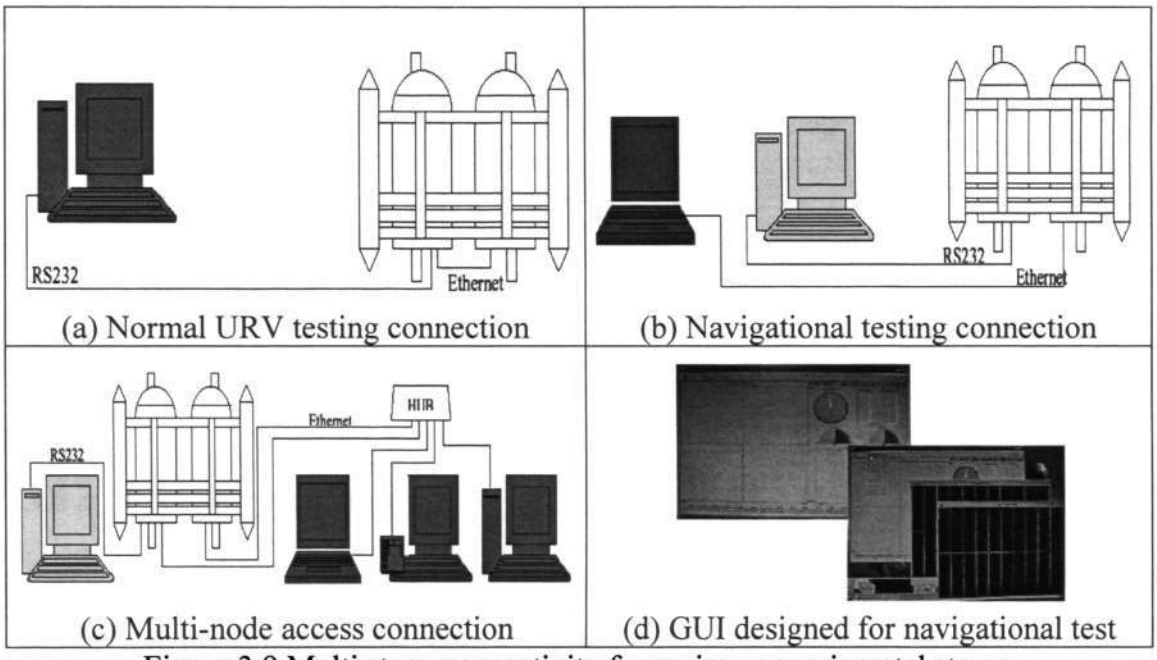
Under the configuration shown in figure 3.9(a), the message queue is used to constantly output data from the sensors to the control pod via the Ethernet connection. When a control command is sent to the sensors, message passing is used to transmit the command to the navigational pod.

Navigational test configuration

In the configuration shown in figure 3.9(b), the data from message queue is used to output to a computer with the graphical user interface (GUI) shown in figure 3.9(d). The data obtain is used to show the sensors trend and navigational plots and also can be logged to a file for future analysis.

Multi-node access configuration

This configuration shown in figure 3.9(c) is used in laboratory conditions where multiple accesses are required from different users testing the software and algorithms to be implemented on the system. Under this configuration, when the individual sensors program is ran, it stops all the other programs and run at its fastest update rate and logs it to a file. This configuration can be used to test individual sensors functionality should any abnormality occurs in the sensor readings.



(a) Normal URV testing connection (b) Navigational testing connection

(c) Multi-node access connection (d) GUI designed for navigational test

Figure 3.9 Multi stage connectivity for various experimental stages

3.5 Overall software architecture

The overall primary software architecture as in figure 3.8 using broadcast communication diagram, that shows how each of the processes function within the URV navigational pod. The diagram shows from the point where the URV starts, initiating the sensors programs and periodically updated according to the given time.

The diagram also shows how data are continuously sent to the message queue and data is logged and stopped when a signal is sent. In a multi access mode, when individual sensor program are ran, it stops the rest of the program and constantly log the data to file and display it on screen.

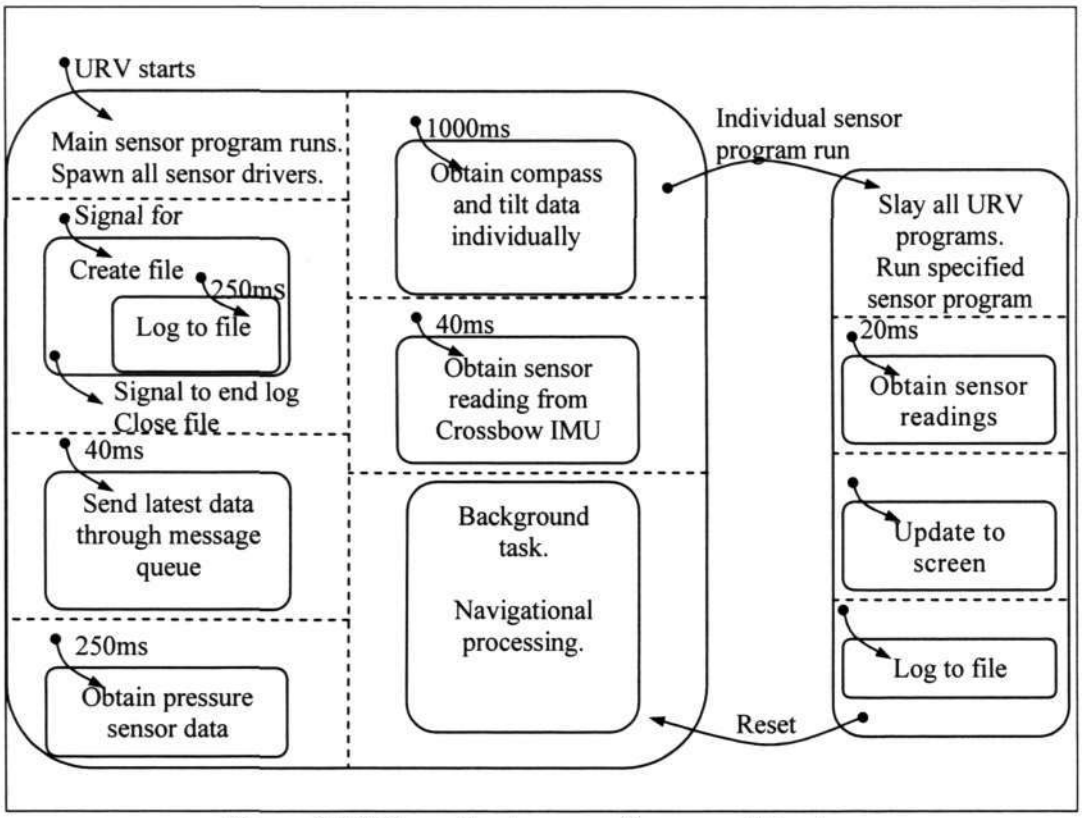


Figure 3.10 Overall primary software architecture

3.6 Chapter summary

The computer subsystem contains the sensors, the computer hardware, the operating system and the software architecture. Since the software is affected by the hardware and operating system, this chapter looked at all the sensors specifications, the computer hardware in which the software are to run in, and the operating system capabilities. Based on the specifications and capabilities of the system, the software architecture was designed to extract data periodically from the sensors and manage all the data in a shared memory architecture where multiple processes can place and extract data from the system. The secondary software architecture is used to expand on the primary architecture capabilities by allowing different mode of connectivity for different purpose such as system or software testing.

CHAPTER FOUR

NAVIGATIONAL SUBSYSTEM ANALYSIS AND DESIGN

The navigational subsystem is the subsystem responsible for processing the data obtained from the sensors and makes it usable for the system. Since data from sensors differs, the primary pre-processing function is to convert the data into standard form based on SI units. Further pre-processing in removing noise and random errors are done through filtering before the data used to predict the vehicle states. Since the onboard sensors cannot determine all the vehicle states, software data integration is applied to specific data where necessary. Once data for all the vehicle states are obtained, the data is processed in terms of frame transformation between the sensor frame and the body frame, and between the body frame and the local fixed frame. Furthermore, since all rotation rates obtained from the sensors are with respect to the body frame, the rotation rate needs to be constantly transformed to the local fixed frame angular notation of roll, pitch and yaw. This needs to be done before all the frame transformations can be applied for all the vehicle states.

4.1 Background

Navigation involves the guiding of a vessel from a point to another. In a manned vehicle system, the human uses the onboard sensors and his personal judgement to perform this navigation task. In an unmanned vehicle system, the operator, operating from a remote location needs to perceive the vehicle from a fixed remote location point of view. The sensors onboard the vehicle needs to be transformed from the sensor frame to the vehicle frame and the vehicle frame needs to be transformed to the local fixed frame for the operator to perceive the vehicle states from the remote location. All these frames need to be established before any of the transformation can be carried out properly.

4.2 Frame establishment

In any vehicle system, there are three primary frames of reference [50], see figure 4.1, namely the local fixed frame, the vehicle frame and the sensor frame. Each frame can be defined by the linear acceleration, velocity and displacement and angular velocity and displacement. These frames need to be properly established for the vehicle system before the navigational algorithms can be applied to the system.

Local (inertial navigational) fixed frame

In the world of navigation, a wide array of reference frame is established to assist the navigation process, with the common ones being Earth-Centered Earth-Fixed frame (ECEF) [51] and geodetic frame navigation system [51]. In ECEF the frame is fixed with the earth with the Z-axis points towards the north pole, the X-axis is defined by the intersection of the plane define by the prime meridian and the equatorial plane and the Y-axis completes a right handed orthogonal system by a plane 90 degrees east of the X-axis and its intersection with the equator. In geodetic frame navigation system, the reference frame is fixed on the earth surface. The frame is established with the Z-axis pointing towards the center of the earth or in the direction of the gravity vector; the X-axis is defined by pointing the axis towards the north pole and the Y-axis completes a right handed orthogonal system by a plane 90 degrees east of the X-axis.

These two frame of reference takes into account navigation around the earth. If a vehicle were to travel around the equator, the vehicle will be travelling in a circular instead of a linear manner. In a tethered URV configuration, the tether constrains the travel boundary usually not more than 1000m. This is only 0.00784% compared to the earth diameter of 12756km. Therefore, any linear motion on the earth surface can be approximated to be linear instead of a curved motion. Based on this, the geodetic navigation frame can be reduced to a basic Cartesian frame transformation sequence as used in [52]. This would allow a simplified model to be implemented into the system.

Body (vehicle) frame

The body frame coincides with the center of gravity (CG) of the vehicle. In establishing the frame the X-axis points towards the front of the vehicle, the Z-axis points downwards and the Y-axis completes a right handed orthogonal system by a plane 90 degrees east of the X-axis [53]. Unlike the local fixed frame, the body frame moves with the vehicle. The control input of the system is based on this frame of reference. However, for it to be useful in navigation, the body frame needs to be transformed to the local fixed frame. The vehicle resolved states can be obtained from the operator point of view.

Sensor frame

For the body frame to local fixed frame transformation to be valid, all the sensors needs to be mounted on the vehicle CG. Unfortunately, it is physically impossible to mount all the sensors about the vehicle CG. Therefore, the sensor frame needs to be transformed to the vehicle frame for the data to be useful. The sensor frame coincides with the sensor CG, which is established by the manufacturer. In most vehicle systems, the sensor frame is mounted to coincide with the vehicle frame of reference [53]. This establishes consistency in the establishment of the frame of reference and thus reduces the complexity of the transformation.

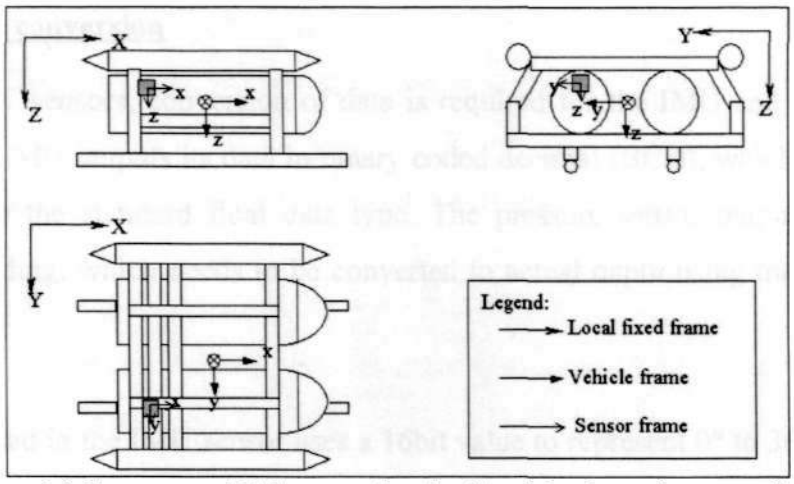


Figure 4.1 Frame establishments for the fixed, body and sensor frames

In defining the navigation space, (a, v, s) are used to define the acceleration, velocity and displacement. Subscripts (x, y, z), (x_1, y_1, z_1) are used to define the states in the local fixed and vehicle frame respectively. $\dot{\theta}, \dot{\phi}, \dot{\psi}$ and θ, ϕ, ψ are the angular velocity and displacement in the fixed frame and p, q, r are the angular velocity in the vehicle frame.

4.3 Pre-processing algorithms analysis

Pre-processing of the data obtained from the sensors involves removing any possible noise and biased error obtained from the sensors. It also involves numerical integration of data to obtain certain vehicle states that cannot be obtained directly from a sensor. These are shown in figure 4.2. Pre-processing of the data is necessary in improving the result obtained thus reducing the error in predicting the vehicle states.

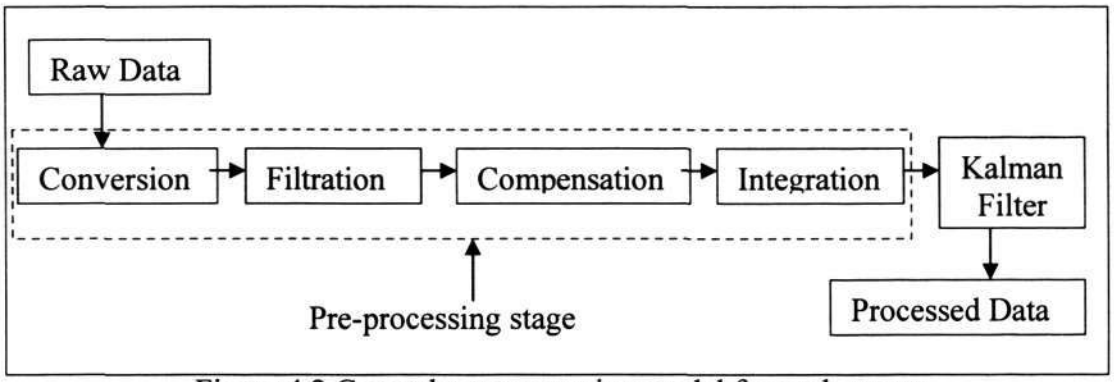


Figure 4.2 General pre-processing model for each sensor

4.3.1 Data conversion

For the URV sensors, conversion of data is required for the IMU and the pressure sensor. The IMU outputs its data in binary coded decimal (BCD), which needs to be converted to the standard float data type. The pressure sensor outputs data as a pressure reading, which needs to be converted to actual depth using the Bernoulli's equation.

The BCD used in the IMU sensor uses a 16bit value to represent 0° to 360° for angle and 2G for acceleration. Doing so allow floating point number to be represented by a real number and maximise the allocated memory. It can be converted by using equation 4.1:

$$angle, acc = \frac{data}{65536} * full\ range(\ 2 for\ acc, 360 for\ angle) \qquad \text{-- 4.1}$$

The pressure sensor on the other hand outputs hydrostatic water pressure. On the other hand, the actual depth of the vehicle is required from the sensor. Therefore, the water pressure can be obtained from the sensor data by using equation 4.2:

$$p = \rho * g * d \qquad \text{-- 4.2(a)}$$

$$\because d = {}^{p}\!/_{\rho g} \qquad \text{-- 4.2(b)}$$

Where p is the water pressure, d is the water depth in meter,

ρ is the water density taken to be 1000kg/m^3, g is the gravitational acceleration taken to be 9.81m/s^2.

4.3.2 Gravity compensation

The IMU accelerometer measures the absolute acceleration experienced by the sensor. The effect of the gravity (G) vector will cause readings in the accelerometer even though the IMU is stationary. This vector needs to be cancelled out by the following equations:

$$g_x = -g(\sin\theta) \qquad \text{-- 4.3(a)}$$

$$g_y = g(\cos\theta \sin\phi) \qquad \text{-- 4.3(b)}$$

$$g_z = g(\cos\theta \cos\phi) \qquad \text{-- 4.3(c)}$$

Where g_x is the G-vector resolved to the body's x coordinate, g_y is the G-vector resolved to the body's y coordinate, g_z is the G-vector resolved to the body's z coordinate, G is the gravity constant taken to be 9.81 m/s^2, θ is the pitch angle and ϕ is the roll angle. For this algorithm to be usable, bias error needs to be removed through calibration and the system need to be initialised by an external sensor. This can be done by using a tilt sensor, whereby the initial values are used obtained while the vehicle is placed on a stable platform under static conditions. Once the system is initialised, the system is updated at periodic interval by the navigation algorithm in section 4.4 and 4.5.

4.3.3 Numerical data integration

Numerical data integration involves integrating a data in a discrete time system. Numerical integration involves approximating states between the discrete times with a straight line. The two mode of approximation is the rectangular approximation and the trapezoidal approximation. Rectangular approximation involves multiplying the states with the time interval thus resulting in a square block. In trapezoidal approximation, the past and the current states are used with the time interval in integrating the data. While the trapezoidal approximation provides a better approximation, lower time interval may improve the accuracy of the rectangular approximation. Equation 4.4(a) shows the numerical integration using block method and 4.4(b) uses the trapezoidal method.

$$\int v\,\Delta t = v_n * \Delta t \qquad \text{-- 4.4(a)}$$

$$\int v\,\Delta t = (v_n + v_{n-1})/2 * \Delta t \qquad \text{-- 4.4(b)}$$

Where v_n and v_{n-1} is the current and previous data and is the Δt time difference.

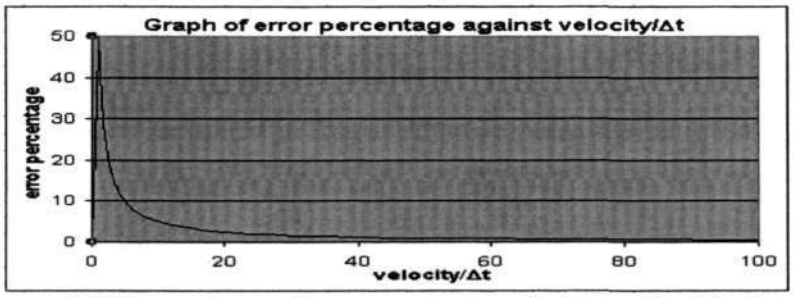


Figure 4.3 Comparison of error percentage between the block and trapezoidal integration against the ratio of velocity and time slice.

Figure 4.3 shows the comparison between the maximum expected error percentage between the two integration methods and the velocity/Δt ratio. From the graph, it can be seen that the maximum error is more than 10% when the ratio is between 0.2 and 5.0. If the block integration was to be used with a sampling time of 0.05s, the velocity needs to maintain below 0.1m/s or above 0.25m/s to minimise the error.

This sampling rate was arrived at after weighing in the factor of processing time for the algorithms and the requirement for correct data to be obtained at stringent timings. A fast sampling is required to reconstruct the signals, especially the noise signal, since insufficient sampling will cause the data to be positively or negatively biased at a specific period thus result in false movements even when the vehicle is stationary. This is when continuous points are sampled to be positive or negative when in actual condition, the noise signal is fluctuating about the X-axis. However, allowing the acquisition rate to be set higher would deprive the other algorithms of their processing time and thus the output may be even lower than the current sampling/output rate of 25Hz. Furthermore, system locks and data corruption may result from the behaviour. The chosen sampling rate allow a synchronised sampling and output rate therefore compliments the timeliness required for the system. Furthermore, the sampling rate together with the implemented pre-processing algorithms should be suitable for the intended URV, which have slow rate of motion.

4.3.4 Low pass filter analysis and design

Low pass filtering involves observing the sensors data in the frequency domain, comparing it with the vehicle dynamics and truncating the frequency range that is deemed to be useless. Since all the data are digital, hardware low-pass filter requires the signal to be converted to analogue before filtering and later converted back to the digital domain. A simpler means is to implement the filter in the digital domain. This requires the filter to be designed in the analogue domain first and transformed to the digital domain by implementing the z-transform [54]. Once the filter equation is obtained, the output from the sensor is input to the filter and the output can be used for further processing in the system. Equation 4.5 shows the Infinite impulse response (IIR) low pass filter equations to be used in filtering the raw digital signal.

$$y[n] = \sum c[j] * x[n-j] + \sum d[j] * y[n-j] \qquad \text{-- 4.5(a)}$$

$$H[f] = \frac{\sum c[j] * \exp(-2\Pi j(f\Delta))}{1 - \sum d[j] * \exp(-2\Pi j(f\Delta))} \qquad \text{-- 4.5(b)}$$

Where y is the filtered data and x is the unfiltered data.

The transfer function H[f] is obtained by transforming the continuous time domain into the discrete time domain and is used to determine c and d, the weightage applicable on the past filtered (y) and unfiltered (x) data in equation 4.5. n is the number of stages for the IIR filter. The block diagram of the algorithm is shown in figure 4.4 below.

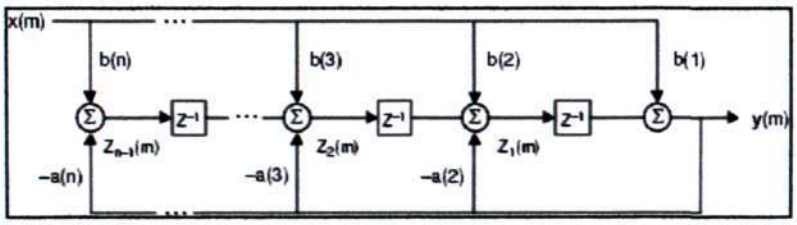


Figure 4.4 Transfer function of the infinite impulse response filter

Infinite impulse response (IIR) low pass filter design

In designing the filter, the dynamics of the system needs to be analysed. The result obtained from the IMU while the URV is in motion is fed to a Fast Fourier Transform (FFT) program. The graph in frequency domain is shown below in figure 4.5. From the graph, it can be seen that the motion of the URV occurs below 2Hz. Higher frequencies can be considered to be random noise and will be filtered out. Therefore, any frequency below 2Hz is allowed to pass while frequency above 7Hz needs to be attenuated by about 40dB to ensure that only the useful data are extracted from the sensors.

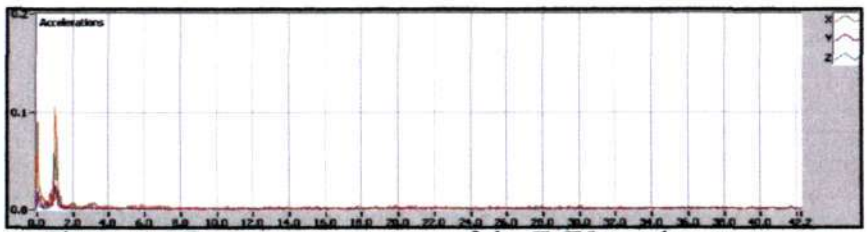


Figure 4.5 Frequency response of the IMU accelerometers

In the digital form, there are four general IIR filter response namely Butterworth, Chevychev I and II and the elliptic filter [55]. The general frequency response is shown in figure 4.6 below. While the Butterworth response closely resembles the analogue low pass filter, the other forms trade off linear response in exchange for lesser filtering stage and sharper attenuation. To ensure data integrity in the pass band, only the Butterworth and Chebychev II form was considered since the pass band is not affected by the ripple.

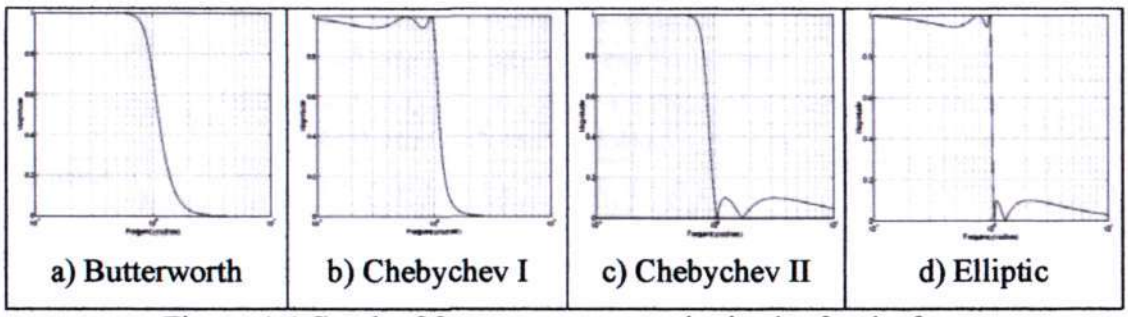


Figure 4.6 Graph of frequency vs. magnitude plot for the four different types of infinite impulse response filter response.

Based on the filter specifications, the Butterworth and the Chebychev II form was designed using matlab. Figure 4.7 and 4.8 shows the magnitude and phase response of Butterworth and Chebychev II form respectively. The graph shows the Butterworth form have a linear magnitude and phase response while in the Chebychev II, ripples are seen on the magnitude response and steps in the phase response. To ensure a stable system, the Butterworth form was selected as for the system low pass filter. Based on the Butterworth form, the IIR low pass filter was design for all the sensors and the transfer function is shown in table 4.1.

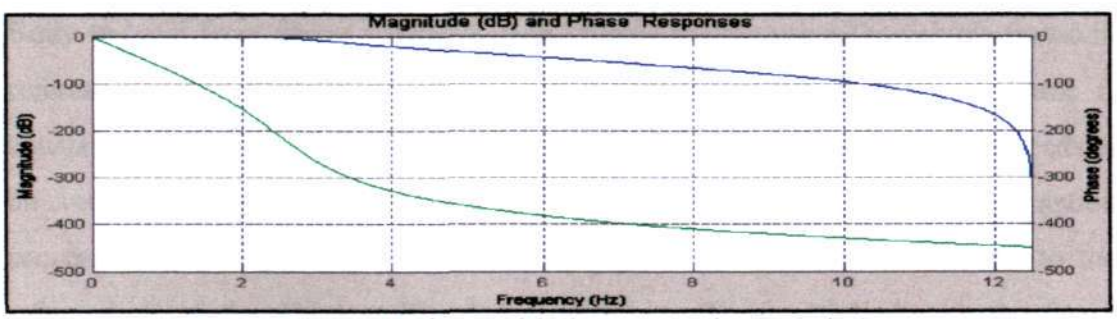


Figure 4.7(a) Butterworth form with linear magnitude and phase response.

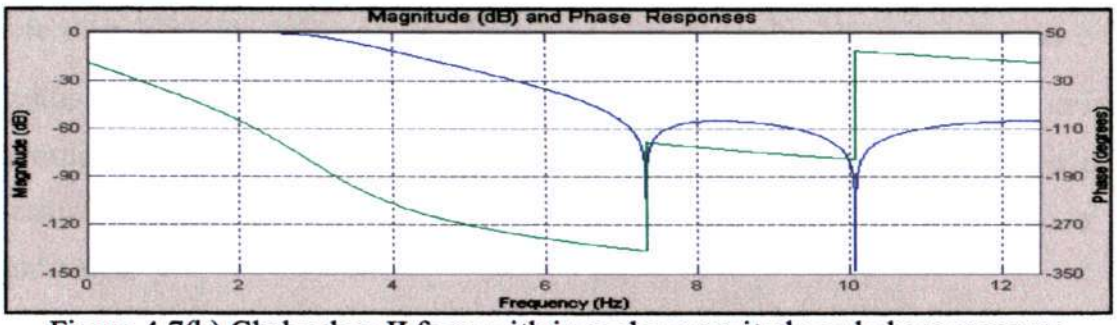


Figure 4.7(b) Chebychev II form with irregular magnitude and phase response.

Sensor	Filter equations	
IMU	$H(z) = \frac{0.0214 + 0.0442z^{-1} + 0.059z^{-2} + 0.0442z^{-3} + 0.0214z^{-4}}{1 + 2.20z^{-1} - 2.48z^{-2} + 1.46z^{-3} - 0.389z^{-4}}$	-- 4.6(a)
Tilt	$H(z) = \frac{0.0214 + 0.0442z^{-1} + 0.059z^{-2} + 0.0442z^{-3} + 0.0214z^{-4}}{1 + 2.20z^{-1} - 2.48z^{-2} + 1.46z^{-3} - 0.389z^{-4}}$	-- 4.6(b)
Compass	$H(z) = \frac{0.957 + 1.91z^{-1} + 0.957z^{-2}}{1 + 1.91z^{-1} + 0.915z^{-2}}$	-- 4.6(c)
Depth	$H(z) = \frac{0.00624 + 0.0125z^{-1} + 0.00624z^{-2}}{1 - 1.176z^{-1} + 0.789z^{-2}}$	-- 4.6(d)
Doppler	$H(z) = \frac{0.802 + 1.60z^{-1} + 0.802z^{-2}}{1 + 1.56z^{-1} - 0.644z^{-2}}$	-- 4.6(e)

Table 4.1 Transfer function for the designed low pass filter

4.4 Frame transformation algorithms

Frame transformation algorithms involve transforming the pre-processed data into data that are useful to the operator. It involves the inclusion of algorithms to convert one frame of motion to another. This is necessary since the vehicle frame of motion and the reference local fixed frame do not coincide while the vehicle is in motion.

4.4.1 Body to local fixed frame transformation

Body to local fixed frame transformation involve transforming the vehicle frame of reference to the local fixed frame of the operator. Since the vehicle frame is constantly changing with respect to the local fixed frame, the transformation needs to be constantly updated in order to reduce any loss of data in the linear approximation process. This transformation involves rotating the vehicle frame of motion to the reference local fixed frame. The linear states are then multiplied with the frame transformation matrix to obtain the linear states of the vehicle with respect to the

reference local fixed frame. The directional cosine matrix, C_b^n is represented by multiplying the roll (ϕ) transformation by pitch (θ) and yaw (ψ) transformation as in equation 4.7(a) [56]. Since the order of rotation is not commutative, the order of rotation needs to follow in sequence in the order of roll, pitch and yaw about the x, y and z axis.

$$C_b^n = \begin{pmatrix} C\psi & -S\psi & 0 \\ S\psi & C\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} C\theta & 0 & S\theta \\ 0 & 1 & 0 \\ -S\theta & 0 & C\theta \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & C\phi & -S\phi \\ 0 & S\phi & C\phi \end{pmatrix} \quad \text{-- 4.7(a)}$$

$$C_b^n = \begin{pmatrix} C\theta C\psi & -C\phi S\psi + S\phi S\theta C\psi & S\phi S\psi + C\phi S\theta C\psi \\ C\theta S\psi & C\phi C\psi + S\phi S\theta S\psi & -S\phi C\psi + C\phi S\theta S\psi \\ -S\theta & S\phi C\theta & C\phi C\theta \end{pmatrix} \text{-- 4.7(b)}$$

ψ, θ and ϕ represents the yaw, pitch and roll angle respectively, C=cos, S=sine.

By multiplying the vehicle frame acceleration or velocity, the vehicle acceleration with respect to the local fixed frame can be obtained from this transformation.

4.4.2 Sensor to body frame transformation

For the body to local fixed frame transformation to be valid, the data obtained from the onboard sensors needs to coincide with the vehicle CG. Since all the sensors are scattered all over the vehicle, the sensor to body frame transformation needs to be implemented to remove any residual motion due to the off-centred motion. Unlike the body to local fixed frame transformation, the sensor to body frame transformation is dependent on the vehicle states. A sensor based on position, velocity and acceleration have its reference set of transformation.

Equation 4.7(b) represents a general frame transformation matrix for a vehicle. Since the alignment of the sensors onboard the vehicle coincides with the vehicle body frame, the three angles can be substituted with a zero in equation 4.7(b). The new matrix represents the transformation for any static state sensors such as depth,

angular and range [56]. Multiplying the matrix with the offset in the X, Y and Z-axis will compensate for the difference in the distance resulting in equation 4.8(a).

$$\begin{pmatrix} s_{x_1} \\ s_{y_1} \\ s_{z_1} \end{pmatrix} = \begin{pmatrix} s_{x_0} \\ s_{y_0} \\ s_{z_0} \end{pmatrix} + \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_{offset} \\ y_{offset} \\ x_{offset} \end{pmatrix} \quad \text{-- 4.8(a)}$$

In obtaining the transformation for velocity, the position of the sensor is resolved into the individual plane of X, Y and Z since a rotation about a particular axis do not include the offset for the particular axis. When a rotation occurs about that axis, a tangential velocity is obtained by differentiating the equation, shown in figure 4.8. Since a rotation can be represented by a combination of roll, pitch and yaw, the matrices are added together resulting in equation 4.8(e). Likewise, differentiating the equations again results in residual centripetal acceleration as shown in equation 4.8(f). This can also be obtained by performing differentiation on equation 4.7(b) and substituting zeros for the angles.

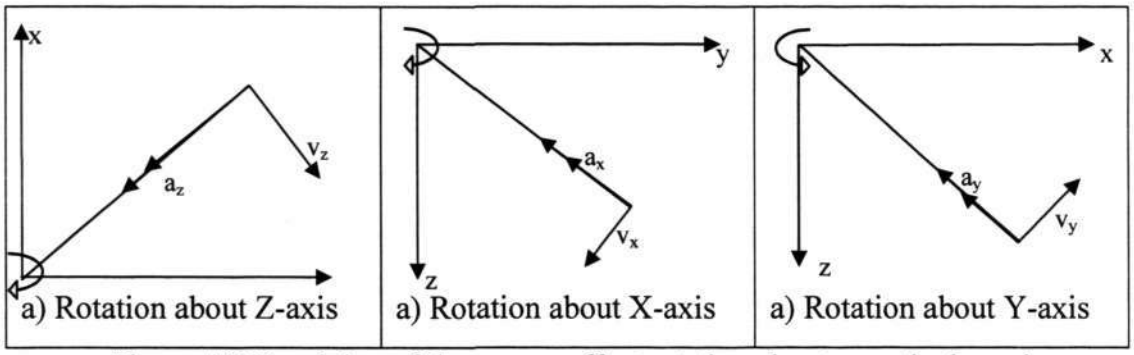


Figure 4.8 Resolving of the sensor offset rotation about a particular axis

Taking the rotation about the z-axis, let the angle be α, therefore

$$s = x_{offset}\underset{\sim}{i} + y_{offset}\underset{\sim}{j} = p'\cos\alpha\underset{\sim}{i} + p'\sin\alpha\underset{\sim}{j} \quad \text{-- 4.8(b)}$$

$$\dot{s} = -\dot{\alpha}p'\sin\alpha\underset{\sim}{i} + \dot{\alpha}p'\cos\alpha\underset{\sim}{j} \quad \text{-- 4.8(c)}$$

Substituting back $x_{offset} = p'\cos\alpha, y_{offset} = p'\sin\alpha$, equation 4.8 (d) is obtained.

$$\dot{s} = v = \begin{bmatrix} 0 & -\dot{\alpha} \\ \dot{\alpha} & 0 \end{bmatrix} \begin{bmatrix} x_{offset} \\ y_{offset} \end{bmatrix} \quad \text{-- 4.8(d)}$$

In the equation $\dot{\alpha}$ represents the angular velocity about the Z-axis. Repeating the step for the X and Y plane and combining the equations, equation 4.8(e) is obtained.

$$\begin{pmatrix} v_{x1} \\ v_{y1} \\ v_{z1} \end{pmatrix} = \begin{pmatrix} v_{x0} \\ v_{y0} \\ v_{z0} \end{pmatrix} + \begin{pmatrix} 0 & -\dot{\psi} & \dot{\theta} \\ \dot{\psi} & 0 & -\dot{\phi} \\ -\dot{\theta} & \dot{\phi} & 0 \end{pmatrix} \begin{pmatrix} x_{offset} \\ y_{offset} \\ z_{offset} \end{pmatrix} \qquad \text{-- 4.8(e)}$$

$$\begin{pmatrix} a_{x1} \\ a_{y1} \\ a_{z1} \end{pmatrix} = \begin{pmatrix} a_{x0} \\ a_{y0} \\ a_{z0} \end{pmatrix} + \begin{pmatrix} -\left(\dot{\theta}^2 + \dot{\psi}^2\right) & 0 & 0 \\ 0 & -\left(\dot{\phi}^2 + \dot{\psi}^2\right) & 0 \\ 0 & 0 & -\left(\dot{\phi}^2 + \dot{\theta}^2\right) \end{pmatrix} \begin{pmatrix} x_{offset} \\ y_{offset} \\ z_{offset} \end{pmatrix} \qquad \text{-- 4.8(f)}$$

4.4.3 Angular rate transformation

The angular rate obtained from the fibre optic gyroscope is attached to the vehicle frame in terms of p, q and r. Before the angular rate can be integrated to obtain the angular displacement of the vehicle, the angular rate of the vehicle needs to be transformed to the navigation frame in order to obtain the correct angles.

Euler transformation

Euler transformation involves direct transformation of the angular rates based on the vehicle frame to the angular rate based on the navigation frame as shown in equation 4.9 [57]. Even though the Euler transformation is sufficient in most cases, the lack of normalising procedure for correcting any rounding and numerical integration errors will cause a substantial drift over time. While the quaternion angles discussed in appendix C resolves this problem, additional computation affects the performance of the system.

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & S\phi T\theta & C\phi T\theta \\ 0 & C\phi & -S\phi \\ 0 & S\phi / C\theta & C\phi / C\theta \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \qquad \text{--eq. 4.9}$$

4.5 Kalman filter algorithm design and analysis

The primary idea of kalman filter is to merge data from multiple sensors to improve the accuracy in its application [58]. A simple form of merging data is by performing weighted average, in which pre-determined weight are given to each data set. Since some states can be obtained directly and indirectly via data integration, it may not result in the same reading. Therefore, a higher weightage based on experiments is given to the more reliable sensor thus fusing the two data together.

$$\bar{x} = w_1 x_1 + w_2 x_2 \qquad \text{-- 4.10}$$

Where $\bar{x}$ is the fused data, w_1 and w_2 are the weightage given to each sensor and x_1 and x_2 are the data obtained from the sensors. However, sensor performance varies with time; this is where advanced data fusion techniques such as the Kalman filter is required to fuse data from multiple sensors. Kalman filter dynamically fuse the data based on the sensor performance.

4.5.1 Kalman filter analysis

The Kalman filter is a predictor-corrector estimation algorithm that uses a dynamic system model to predict state values and a measurement model to correct this prediction [58]. Based on the probabilistic properties of a time series of measurements, the Kalman filter update the estimate of the system state and utilises a dynamic model to propagate the state estimate between measurements. The Kalman filter first developed by R.E. Kalman [58] in 1960 addresses a discrete linear model. However, advancement in digital computing extends the Kalman filter to process non-linear model by linearising about the current mean in the extended Kalman filter (EKF) or by a deterministic sampling approach through weighted sample points in the Unscented Kalman filter (UKF) [59].

The discrete-time Kalman filter addresses systems with dynamic models based on a linear, vector difference equation as shown in equation 4.11(a). Based on the available sensors, the measurement equation is formed as in equation 4.11(b).

$$\dot{x}_k = F x_{k-1} + w_{k-1} \qquad \text{-- 4.11(a)}$$

$$z_k = H_k x_k + v_k \qquad \text{-- 4.11(b)}$$

Where x_k and x_{k-1} are the predicted and current system states, related by the matrix A. z_k is the measurement matrix, related to the predicted states by the matrix H. The random variables w_k and v_k represent the process and measurement noise (respectively). They are assumed independent of each other, white, and with normal probability distributions.

Based on the definition, the process-noise covariance matrix Q and measurement-noise covariance matrix R can be formed from w_k and v_k respectively, resulting in a diagonal matrix where the diagonal components are the error covariance. With the predictor model and the measurement model together with the covariance matrix Q and R, the Kalman filter algorithm can be formulated. However, prior to the filter formulation, the Q and R matrix needs to be formulated whereby the measurement covariance R can be obtained from experiments on the sensor characteristics and Q, the process noise can be estimated based on the operational environment.

The Kalman filter algorithms can be divided into two sets of equations, namely the Time update equations, which is used to predict the states and the Measurement update equations, which is used to correct the states [58]. This set of equations follows an ongoing cycle as shown in figure 4.9 and the corresponding equations are shown in equation 4.12 (a-e).

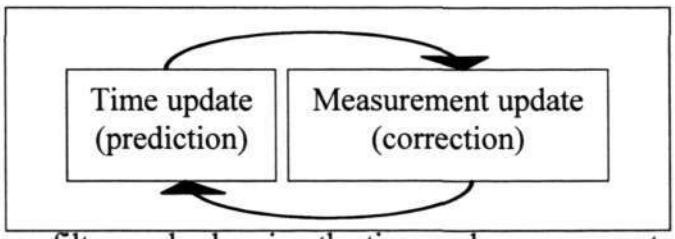


Figure 4.9 The Kalman filter cycle showing the time and measurement update stages

Time update equations:

$$\hat{x}_k^- = A_{k-1}\hat{x}_{k-1} \qquad \text{-- 4.12(a)}$$

$$P_k^- = A_{k-1}P_{k-1}A_{k-1}^T + Q_{k-1} \qquad \text{-- 4.12(b)}$$

Measurements update equations:

$$K_k = P_k^{'} H_k^T [H_k P_k^{'} H_k^T + R_k]^{-1} \quad \text{-- 4.12(c)}$$

$$\hat{x}_k = \hat{x}_{k-1} + K_k (z_k - H_k \hat{x}_{k-1}) \quad \text{-- 4.12(d)}$$

$$P_k = (I - K_k H_k) P_k^{'} \quad \text{-- 4.12(e)}$$

Where the common terms used are the same as those in equation 4.11, K denotes the kalman gain, I is the identity matrix and the P defines system error covariance.

The filter starts with the input of the initial estimates of $\hat{x}_k^-$ and $P_k^{'}$. Based on these estimates, the Kalman gain can be obtained from equation 4.8(c). Measured readings are placed into z_k and the corrected states are obtained. The new error covariance can then be obtained. The corrected states and error covariance are then used to predict the new estimated states and error covariance and the cycle continues.

4.5.2 Kalman filter design

The main purpose of the twin-barrel URV kalman filter is to improve on the sensors output by fusing the data. For this purpose, the kalman filter is designed based on the sensor kinematics model. Before the filter can be designed, the states in the navigational space and the sensors update states needs to be analysed. The URV, from the remote operator point of view can be described by its linear and angular states for the each of the X-Y-Z axes. The linear states can be defined by the body's acceleration, velocity, and position while the angular states can be defined by angular velocity and position. States in the navigation space: $[a_x, v_x, s_x, a_y, v_y, s_y, a_z, v_z, s_z, \dot{\theta}, \theta, \dot{\phi}, \phi, \dot{\psi}, \psi]$

The onboard sensors as discussed in section 3.2.2 would only provide certain states, namely $[s_z, \theta, \phi, \psi]$, in the navigational space. The acceleration $[a_{x1}, a_{y1}, a_{z1}]$, velocity $[v_{x1}, v_{y1}, v_{z1}]$ and angular rate $[p, q, r]$ are described in the vehicle space. These states need to be mapped back to the navigational space using the algorithms in section 4.4. The update process is further discussed below in the measurement update equations.

Furthermore, the positional states $[s_x, s_y]$ cannot be obtained directly from the sensors.

Besides the states, the filter needs to fit certain criterion. The main criterion is to ensure that after the introduction of the filter, the system is still capable of performing in a soft real time, whereby the filter does not affect the data acquisition performance. Since the computation of the filter, involve matrices and matrix inversion in equation 4.13(c), the processing time increase exponentially with the increment of each stage. Therefore, there is a need to redefine the filter into smaller filters with lesser stages in order to reduce the computation time, but with minimal impact to the prediction of the stages.

The other criterion for the system is that since the sensor system is also used as a feedback to the control system, the states in the vehicle space needs to be retained. The control system will require the input in the vehicle state. Therefore, the output states and the measurement states need to be identical.

With the criterion, the approach in the designing of the kalman filter for the twin-barrel URV is based on fusing all the angular states in a single kalman filter. However, the angular rates defined by the sensors are based on the body frame. There is a need to convert the body attached angular rates, p, q, r to the local fixed frame angular rate $\dot{\theta}$, $\dot{\phi}$, $\dot{\psi}$, which is required to predict the fixed frame angles θ, ϕ, ψ. This is discussed in section 4.4.3 of this chapter. The angles can then be used for the transformation between the vehicle states and the navigational states. Euler angular rate transformation is used and merged into the filter algorithm.

In merging the acceleration and velocity, the two states are filtered based on the individual vehicle frame X-Y-Z axes. This is then resolved into the navigational states and numerical integration can be performed to obtain the positional states of the vehicle. From the filter design, since the filter still output the vehicle frame velocity and angular velocity, the filtered data can still be used as a feedback to the control system. On the other hand, the filtered velocity can be resolved to the

navigation frame, which can be used to resolve the URV position in the navigation frame.

Angular state kalman filter algorithm design (p, q, r, θ, ϕ, ψ)

The states for the kalman filter are as follows:

x_1: body attached angular rate p x_4: fixed frame angle θ

x_2: body attached angular rate q x_5: fixed frame angle ϕ

x_3: body attached angular rate r x_6: fixed frame angle ψ

Therefore $x = [\, p \;\; q \;\; r \;\; \theta \;\; \phi \;\; \psi \,]^T$

Based on the states and Euler transformation, the state model ($\dot{x} = fx$) is as follows:

$$\dot{p} = 0 \quad \text{-- 4.13(a)}$$

$$\dot{q} = 0 \quad \text{-- 4.13(b)}$$

$$\dot{r} = 0 \quad \text{-- 4.13(c)}$$

$$\dot{\phi} = p + (\sin\phi \tan\theta)q + (\cos\phi \tan\theta)r \quad \text{-- 4.13(d)}$$

$$\dot{\theta} = (\cos\theta)q - (\sin\theta)r \quad \text{-- 4.13(e)}$$

$$\dot{\psi} = (\sin\phi / \cos\theta)q + (\cos\phi / \cos\theta)r \quad \text{-- 4.13(f)}$$

This result in the following matrix (f) form:

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & \sin\phi\tan\theta & \cos\phi\tan\theta & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 & 0 & 0 \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \\ \phi \\ \theta \\ \psi \end{bmatrix} \quad \text{-- 4.13(g)}$$

From equation 4.13(a-f), the system Jacobian ($F = \partial f / \partial x$) is:

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & \sin\phi\tan\theta & \cos\phi\tan\theta & (\cos\phi\tan\theta)q - (\sin\phi\tan\theta)r & (\sin\phi/\cos^2\theta)q + (\cos\phi/\cos^2\theta)r & 0 \\ 0 & \cos\theta & -\sin\theta & 0 & -(\cos\theta)r & 0 \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta & (\cos\phi/\cos\theta)q - (\sin\phi/\cos\theta)r & -(\sin\phi/\sin\theta)q - (\cos\phi/\sin\theta)r & 0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \\ \phi \\ \theta \\ \psi \end{bmatrix}$$

-- 4.13(h)

From equation 4.14(g), the transition matrix ($A_k = x + \dot{x}\Delta t$) is

$$\begin{bmatrix} p \\ q \\ r \\ \phi \\ \theta \\ \psi \end{bmatrix}_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \Delta t & \sin\phi\tan\theta\Delta t & \cos\phi\tan\theta\Delta t & 1 & 0 & 0 \\ 0 & \cos\theta\Delta t & -\sin\theta\Delta t & 0 & 1 & 0 \\ 0 & (\sin\phi/\cos\theta)\Delta t & (\cos\phi/\cos\theta)\Delta t & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \\ \phi \\ \theta \\ \psi \end{bmatrix}_{k-1} \quad \text{-- 4.13(i)}$$

For the measurement update equation,

z_1: body attached angular rate p (IMU)

z_2: body attached angular rate q (IMU)

z_3: body attached angular rate r (IMU)

z_4: fixed frame angle θ (Tilt)

z_5: fixed frame angle ϕ (Tilt)

z_6: fixed frame angle ψ (Compass)

Therefore $z = [\, p \;\; q \;\; r \;\; \theta \;\; \phi \;\; \psi \,]^T$

In mapping the measurement update states, since x and z are identical, a linear measurement equation result in the identity matrix as follows:

$$\begin{bmatrix} p \\ q \\ r \\ \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \\ \phi \\ \theta \\ \psi \end{bmatrix} + v(k)\{noise\ vector\} \quad \text{-- 4.13(j)}$$

Equation 4.14 (g), (j) and discretised form of equation 4.14 (h) based on $A_k = I + F\Delta t$, combined with equation 4.11 and 4.12, forms the kalman filter. Q and R diagonal matrix can be formulated from experiments and x_{k-1} and P_{k-1} are initialised as zeros.

General Linear state kalman filter algorithm design (v, a)

The states for the kalman filter are as follows:

x_1: body attached velocity v x_2: body attached acceleration a

Therefore $x = [v\ a]^T$

Based on the states and Euler transformation, the state model ($\dot{x} = fx$) is as follows:

$$\dot{v} = a \quad \text{-- 4.14(a)}$$

$$\dot{a} = 0 \quad \text{-- 4.14(b)}$$

This results in the following matrix form:

$$\begin{bmatrix} \dot{v} \\ \dot{a} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ a \end{bmatrix} \quad \text{-- 4.14(c)}$$

The transition matrix becomes:

$$\begin{bmatrix} v \\ a \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ a \end{bmatrix} \quad \text{-- 4.14(d)}$$

For the measurement update equation,

z_1: body attached angular rate v (Doppler) z_2: body attached angular rate a (IMU)

Therefore $z = [v\ a]^T$

Since z states are identical to the x states, an identity matrix is formed.

$$\begin{bmatrix} v \\ a \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ a \end{bmatrix} + v(k)\{noise\ vector\} \quad \text{-- 4.14(e)}$$

Equation 4.15 (d), (e) combined with equation 4.12 and 4.13, forms the kalman filter. Q and R diagonal matrix based on the process-noise while the sensor noise can be formulated from experiments, discussed in the next chapter and x_{k-1} and P_{k-1} are initialised with zeros.

4.6 Chapter summary

The navigation process involves inclusion of algorithms to pre and post process the data to make it useful for the operator. While there are quite a number of algorithms that can be used in the processing, certain algorithms have a higher precedence over another. However, in doing so, errors in approximation may result especially in data integration. Furthermore, in constructing the Kalman filter, assumptions, namely splitting the angular states from the linear states into separate filters. While this reduces the effectiveness of the filter, it reduces the computation required by the system. As a non-linear system, the Kalman filter designed is uncontrollable and unobservable whereby the initial states cannot be obtained from the output at a later time. Therefore, the Kalman filter updates only display the prediction for certain unobservable states. Since, certain algorithms may be highly computer intensive, including the algorithms in the system will result in a slower processing. Slower processing may result to function worst of as compared to a system without the algorithms. Therefore, the performance of the algorithms needs to be established and where possible, a reduced forms of the algorithms to be used to process the data.

CHAPTER FIVE

SYSTEM PERFORMANCE

In implementing the navigation system, various decisions were required to be made. Experiments are required to calibrate the sensors, validate the algorithms and to obtain the best possible solution for each problem. Since all sensors that are produced may have slight variance in performance, calibration is done to ensure the sensors function as required. Furthermore, since certain situations such as the angular rate transformation, the most suitable algorithm needs to be singled out for use with the system. Furthermore, experiments are required to validate the design filters are suitable for the system. Experiments needs to be designed to ensure the required parameter is tested while keeping the rest of the variables constant. This is to ensure the most appropriate combination of navigational architecture and algorithms are implemented on the system.

5.1 Background

Experiments are required to test the performance of the system in terms of timing and errors in determining the states. In performing the experiments, an identical computer system was configured to simulate actual system performance. The timing performances of the algorithms were tested on the system to ensure the combined algorithms are capable of running in soft real time constraints.

The sensors are added to the system for calibration and testing purpose. Certain sensors such as depth and altimeter require water as the operational medium, while others work in the air. This allows certain test to be conducted under controlled lab/land condition before it is merged with the water-based sensors for tests in water condition. From the sensors available, the classification of the sensors can be done as follows:

Table 5.1 Classification of the navigation sensors used on the URV

Sensors	Water medium	Air medium
Static state	Depth sensor, Altimeter	Inclinometer, KVH compass
Dynamic state	Velocity Doppler	IMU

For calibration and algorithms design and testing, a Kawasaki robotic arm was selected as it provide controlled lab-based condition. Testing was done on the air-based sensors using offline processing, whereby data are collected from the sensors before passing through the algorithms. This allows comparison of the algorithms using identical dataset. Once the algorithms are selected, the sensor system is tested for its ability to track predefined paths. For this purpose, a trolley and all terrain robotics vehicle (ATRV) was used. Doing so allows the fine-tuning of the kalman filter initial parameters.

The water-based sensors are then calibrated in the diving pool. Without advanced equipment, testing rigs were designed to minimise variable parameters. Once the sensors are calibrated, the whole sensor system is assembled together for water-based path tracking test using a floating platform. The results conclude the testing system of the navigation sensor system.

5.2 Experiment: Computer subsystem processing performance

One requirement of the system is to ensure it function within soft real time constraints. An experiment was setup to test the algorithms performance with the URV computer system, using a separate identical system as discussed in chapter three and shown in figure 5.1. Comparison was done between various data acquisition, integration, angular rate and frame transformation methods, number of low pass filter stages and general sequences such as reading and writing to memory and files. The results would be used with the other experiments to determine the most suitable algorithms for the system.

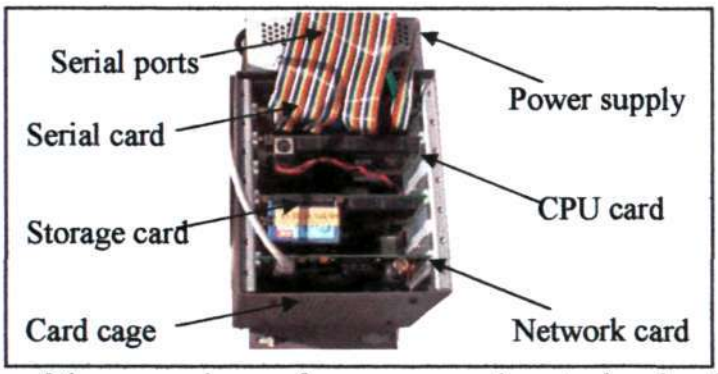


Figure 5.1 Setup of the processing performance experiment whereby algorithms are looped in the system and the time taken was recorded.

Timings for serial data acquisition

Data from the serial port can be access using several programming techniques as discussed in section 3.31. These techniques affect the availability of the CPU resources to other processes and thus affect the system performance. The techniques used are polling, polling with interrupt return after the first byte, polling with interrupt return after the last byte. The IMU was used for this experiment running at 38400bps whereby timings for the three different techniques are shown in figure 5.2.

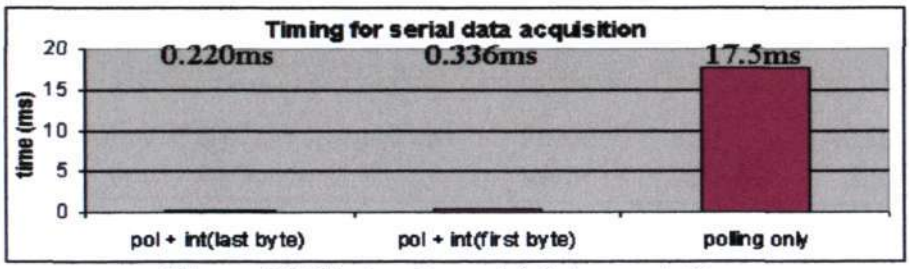


Figure 5.2 Timing for serial data acquisition

The timings show that using only the polling technique results in 0.0175s of wasted time in waiting for the sensor to obtain the signal, process the data and respond to the computer. However, when an interrupt was activated for the serial port and the CPU was released to service other processes, the timing drops drastically. It slightly drops further when the interrupt was set to activate only after the last byte was received. The large difference in timing greatly affects the computer ability of real-time handling the sensors since there are seven sensors onboard the URV. The later technique is suitable to be used in the twin-barrel URV as the interface between the computer and the sensors.

Timings for specific algorithms

Certain algorithms are required to be included in obtaining the states of the vehicle. Since some of the algorithms take longer time than others, there is a need to minimise the use of the algorithms where possible. Figure 5.3 shows timings for specific algorithms, which were looped for a million cycles except for writing to the flash module, which was looped for 1000 cycles. The algorithms are; empty 'for' loop, IMU data conversion from BCD to float, offsetting the centre of gravity for the IMU accelerometer, body to local fixed frame transformation, handling sine and cosine routine for all the three axis and writing a 16 byte data to the flash module. All these algorithms were discussed in chapter four under section 4.3, except for the data logging algorithm, which was discussed in chapter three under section 3.3.3.

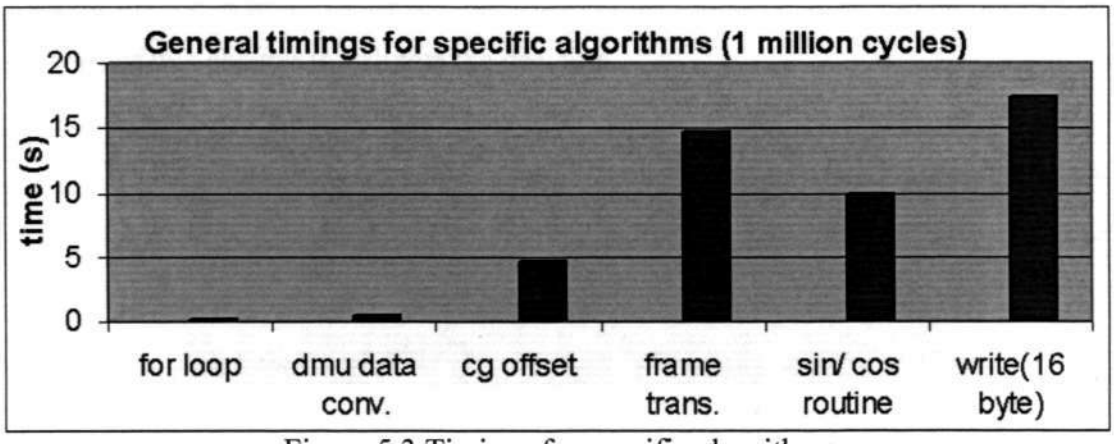


Figure 5.3 Timings for specific algorithms

The result shows a wide range in timings. The timing for the algorithms gives and idea on how the system will perform. The algorithms, which include sine and cosine functions, namely CG offset, frame transformation and sine/cosine conversion routine consumes quite a significant amount of time. However, these algorithms are necessary for the system to function. The 'for' loop hardly affect the computer time but writing to the flash module greatly affects the computer time even though it was only ran a thousand cycles. While writing to the flash module necessary to log the data for future analysis, the logging rate needs to be set to a level, which is required for each experiment. Logging cannot be activated at a high rate, since it will greatly affect the computer performance, which would affect the system soft-real time requirement.

Timings for the sensor to body frame transform

Figure 5.5 shows the timing for the sensor to body frame transformation, which was discussed in section 4.4.2. The sensor to body frame transformation affects dynamic sensors such as velocity and acceleration based sensors. From the algorithms, the accumulated error is resulted from the multiplication of the sensor to CG offset distance multiplied by the angular rate. Unlike air or land vehicles where the sensor to CG distance and the angular rate is significant, the IMU and the Velocity Doppler for the twin-barrel URV are mounted within 0.8m from the CG and the maximum turning rate is about 2.5rad/s. With a significant processing time and a minimal expected improvement based on the specifications, the algorithms may not be suitable for the system. However, the tracking performance is still explored in later sections of this chapter to validate the decision.

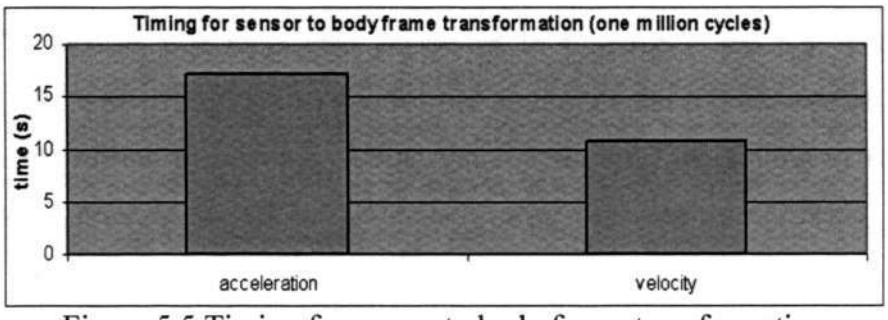


Figure 5.5 Timing for sensor to body frame transformation

Low pass (IIR) filter design and analysis

In exploring the impact by the low pass filter algorithm on the system, timing was taken for a single, double, triple and four stage low pass filter algorithms. From the results the higher the stages in the LPF, the more time it needs to compute the algorithms. From the results, the increase in computation time is directly proportional to the number of low pass filter stages. Unfortunately, with lesser LPF stages, the drop in the gain will be gentler and there will be lesser attenuation. Therefore, in order to obtain the conditions set on the system, the designed low pass filter cannot be changed, as the designed low pass filter needs to be match to the sensor output and the expected operational condition.

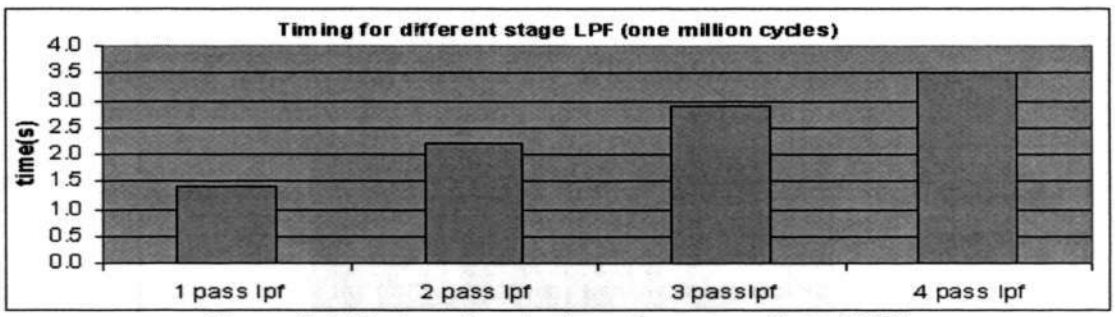


Figure 5.6 Timings for single to four stage digital LPF

5.3 Experiment: Sensor calibration and algorithms performance

This stage requires the sensors to be calibrated and the algorithms tested before it can be used in the navigation system. The sensors are connected to the computer system used in 5.2 using the architecture designed in chapter three. For this purpose, the system needs to be tested in a controlled lab environment. The Kawasaki robotics arm was chosen, as it is an industrial equipment, which allows precise control in 3D space. The system was connected as shown in figure 5.7 and 5.8. The control software was written in QNX to control the motion of the robotic arm and log the raw data for offline processing. Using the robotic arm, the following tests were conducted:

1. Static calibration
2. Dynamic calibration
3. Performance of the Euler transformation in converting from the body frame p, q, r form to the navigation frame ϕ, θ, ψ form.
4. Performance of the Sensor frame to the Body frame transformation
5. Testing of the designed Kalman filter to merge the angular and angular rate data.

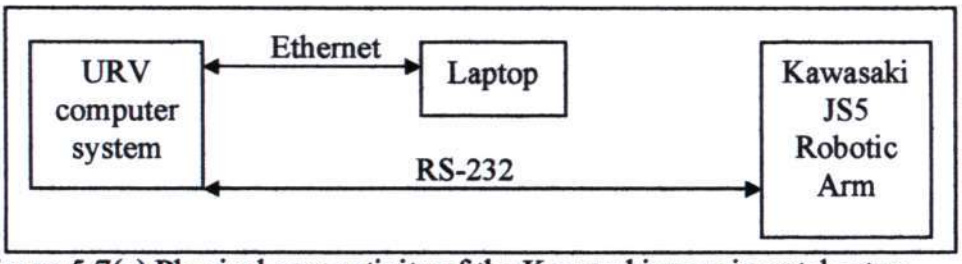


Figure 5.7(a) Physical connectivity of the Kawasaki experimental setup

Figure 5.7(b) Kawasaki experimental setup initialisation using a spirit level to test for initial flatness of the end-effector.

Sensor calibration

Calibration is required before the sensors can be used with the system. The first step of this experiment is to calibrate the sensors in both static and dynamic condition. The calibrations of the sensors are shown in appendix B, where procedures are outlined and the results are plotted. Table 5.2 shows the results of the sensor zero error and standard deviation of the sensors under static condition. The zero error is required to be offset from the sensors to ensure the sensors output the correct values while the standard deviation shows the spread of the data distribution. From the results, the rate gyro used in the IMU has a large variance while the tilt sensor and the compass have a much smaller variance. Since the rate gyros are used to

compliment the Tilt sensor and Compass, which when applied in the kalman filter, the result should further improve. The small variance of the accelerometers is important since a slight chance in the acceleration results in a large change in position.

Table 5.2 Sensors zero error and standard deviation

Sensors	Data	Zero error	Std deviation
IMU	Rate gyro P	0.0130°/s	0.153
	Rate gyro Q	-0.0328°/s	0.139
	Rate gyro R	-0.0269°/s	0.144
	Ax_1	-0.0103m/s^2	0.0287
	Ay_1	0.00281m/s^2	0.0221
	Az_1	-0.00304m/s^2	0.0244
Tilt sensor	Roll	-1.27°	0.00904
	Pitch	-0.0320°	0.00731
Compass	Heading	N.A.	0.0926
V. Doppler	Vx_1	0.00347m/s	0.00477
	Vy_1	0.00174m/s	0.00387
	Vz_1	-0.00278m/s	0.00379
Depth	Pz	0.0314m	0.0267

Angular rate transformation test

In section 4.4.3, the algorithm to convert the body attached angular rate to the fixed frame angular rate was discussed. For the URV, the Euler transformation should be sufficient since it is extremely unlikely that the URV will pitch to 90°, which results in a singularity. This experiment was conducted to test the Euler angle conversion of the angular rate from the body frame to the navigation frame. The robotic arm was used to rotate the sensors clockwise from 0° to 360° and counter clockwise back to 0° about a tilting angle of 45° above the horizon. Figure 5.8 illustrates this experiment. Figure 5.9 shows the roll and pitch motion of the robotic arm and the results obtained from the IMU, while figure 5.10 shows the Euler angular rate transformation results.

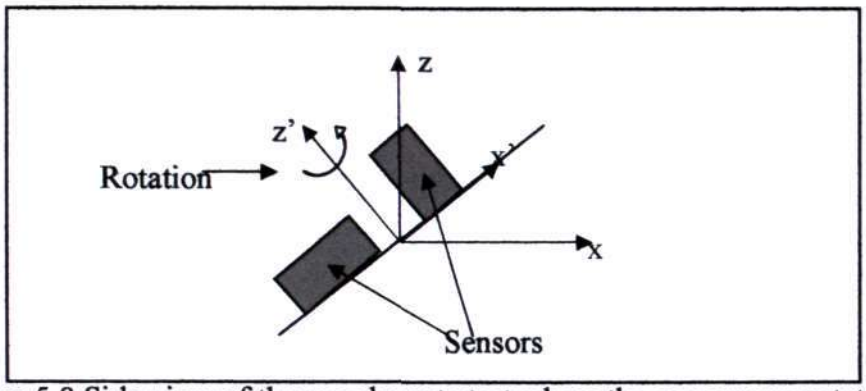
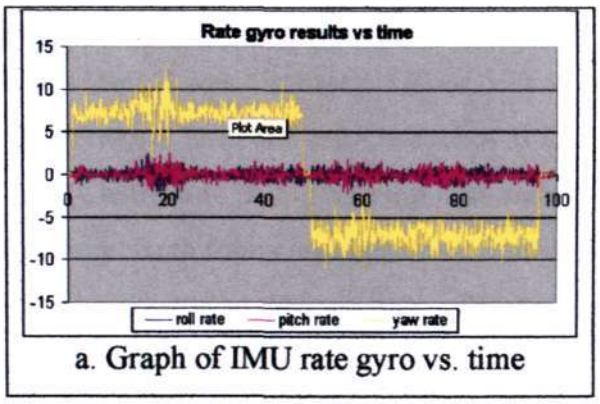
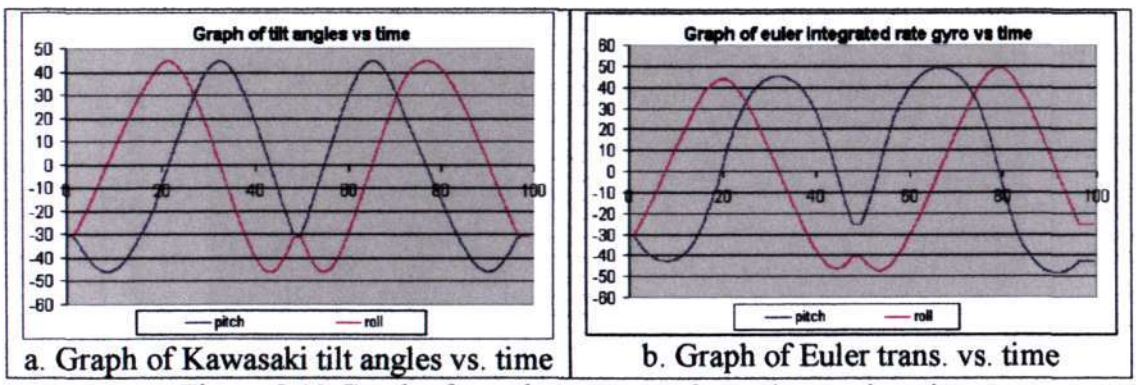


Figure 5.8 Side view of the angular rate test where the sensors are rotated about a tilted angle of 45° above the horizontal



a. Graph of IMU rate gyro vs. time

Figure 5.9 Graph of rate gyro angular rate against time for a planar rotation at 45° tilts.



a. Graph of Kawasaki tilt angles vs. time

b. Graph of Euler trans. vs. time

Figure 5.10 Graph of angular rate transformation against time

From figure 5.9, it can be seen that the tilted rotation only results in the motion of the body attached yaw rate. The Euler transformation is required to transform the tilt angles to the results obtained from the Kawasaki arm as shown in figure 5.10a. However, from the actual transformation results obtained from figure 5.10b, it can be seen that the results obtained from the Euler transformation have slight deviation

from the actual results. From the results, it can be seen that the error after a 360° rotation results in an error of about 7° while after it was rotated back, the error was about 9°. One of the possible causes of the error is in the numerical integration. Due to the large noise and discretisation of data, the effect would result in a large error.

Low pass filter test

To further improve on the data, a low pass filter was used to pre-process the angular rate before it is transformed using the angular rate transformation. As discussed in section 4.3.4, the low pass filter only allows the selected range of frequency to pass while filtering the higher frequency range. In the section, the difference in the butterworth and the Chebychev II was discussed. Figure 5.11 shows the comparison between the raw data, butterworth form and the Chebychev II form. From the results, the butterworth and the Chebychev II form, with the same design parameters, behaves almost the same way. Both forms are capable of filtering the noisy data with only slight difference in phase between the two filters. Therefore, either form could be used with in the URV. Since the butterworth form was selected earlier, tests are done using the butterworth form.

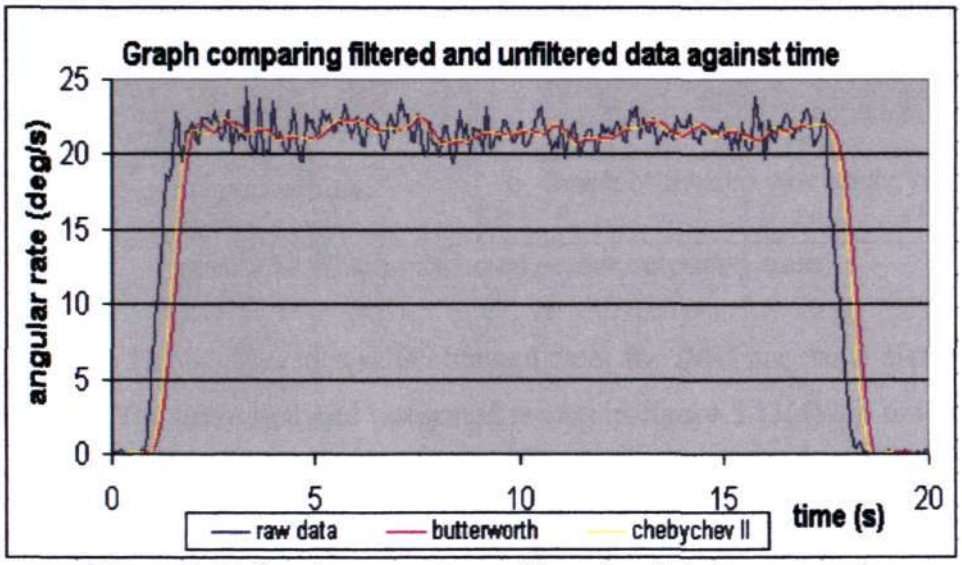


Figure 5.11 Graph comparing unfiltered and the butterworth and Chebychev II low pass filter form

The main task of the low pass filter test is to compare the filtered results with the unfiltered results. Therefore, the designed low pass filter was used to filter the graph in figure 5.19. The low pass filtered results is shown in figure 5.12. However, only the Euler algorithm was used to convert the body attached angular rate to the fixed frame angular rate. The filtered results are converted to the fixed-frame angular rate, integrated, and shown in figure 5.13.

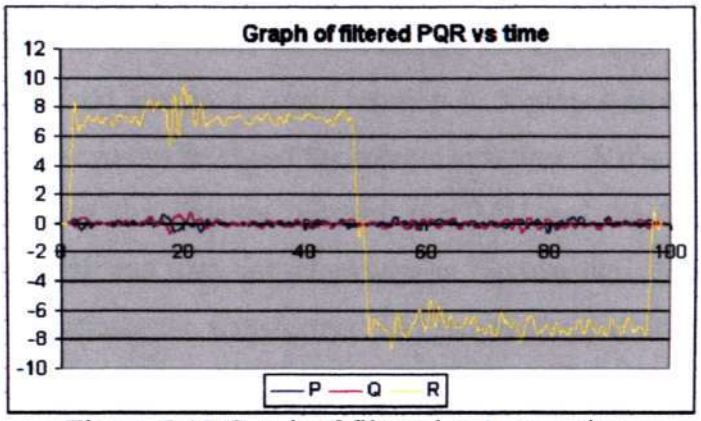
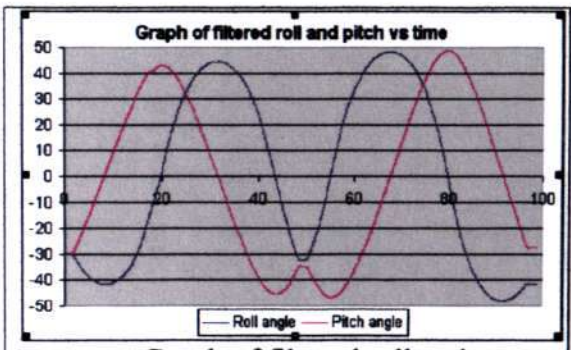
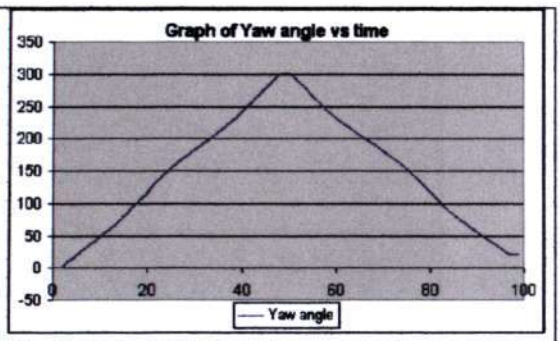


Figure 5.12 Graph of filtered p, q, r vs. time



a. Graph of filtered roll and pitch angles vs. time



b. Graph of filtered yaw angle vs. time

Figure 5.13 Graph of filtered sensor output vs. time

From figure 5.12, the filtered results obtained from the IMU are much cleaner than the raw data. The converted and integrated results in figure 5.13(a) are much closer to the actual angles as compared to the unfiltered results shown in figure 5.9. The filtered result shows improvement in the results obtained. However, even with the filtered results, there are discrepancies between the actual roll and pitch angles and those obtained from the IMU. For example, the final tilt angles should converge to -30°, but the results from the IMU shows roll and pitch angles of -42° and -27° respectively. While the tilt sensor comprised of accelerometers can provide the

angles, the result obtained from the tilt sensor will be inaccurate under heavy accelerations. To ensure the most accurate angles are obtained at any time, a kalman filter is required to merge the results from the various sensors to improve the performance of the system.

Sensor to body frame transformation test

In testing the sensor to body frame transformation, the algorithm was applied to the IMU. Using the robotic arm in a z-axis rotation in a planar motion, a distance of 0.25m was used between the IMU and the robotic arm base. A distance of 0.25m was chosen as it replicates the distance between the IMU and the URV C.G.. It was rotated 270° clockwise and then anti-clockwise in circular horizontal planar motion in 15s. Figure 5.14 shows the transformed and untransformed distance travelled in the X and Y-axis.

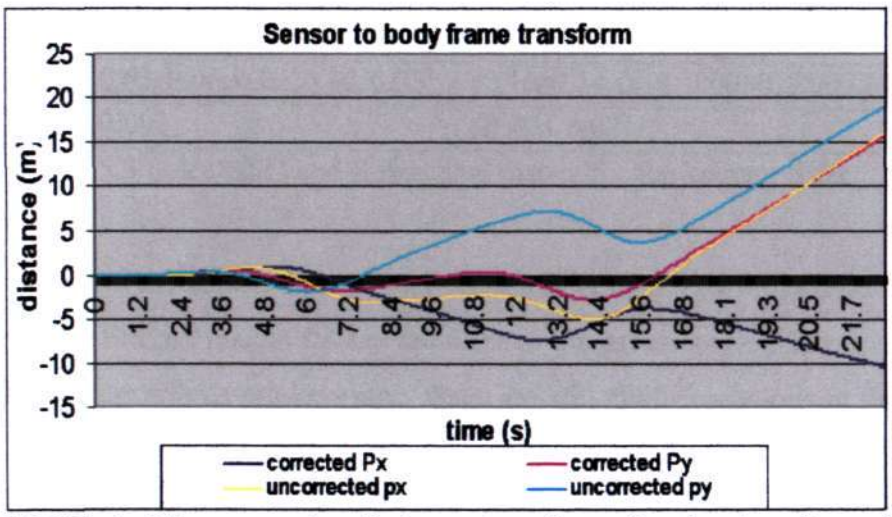


Figure 5.14 Plot of the comparison between transformed and untransformed results whereby limited improvement of the results were obtained from the transformation

From figure 5.14, it can be seen that applying the sensor to body frame transform slightly improve the results obtained from the IMU. Unfortunately, the drift in the IMU sensors is more significant than the error result in the offset of the center from URV C.G location. Since the data from the accelerometer needs to be integrated twice to obtain the change in position, a slight error in the acceleration results in a large difference in the positional change. The result shows it travelled a distance of ±7m at the time of 13s and the distance continue to drift once the IMU was allowed

to rest. Furthermore, the algorithms consume a large amount of CPU resources and its inclusion may result in poorer results due to increased processing time while giving only slight improvements to the system.

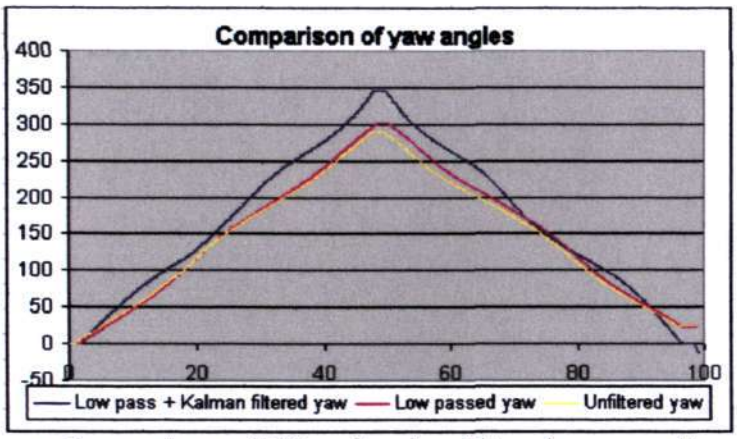
Kalman filter test

Using the processed results obtained from the angular rate transformation test, the results were further fed into the Kalman filter algorithm designed in chapter four to predict and correct the obtained data at every stage. The R matrices for the kalman filter are based on the sensor standard deviation obtained in section 5.3. The Q matrices on the other hand were selected based on assumption of the environmental conditions. The jitters due to the acceleration and angular rates are assumed to be larger as compared to the Doppler and tilt sensor. The values used are shown in table 5.1.

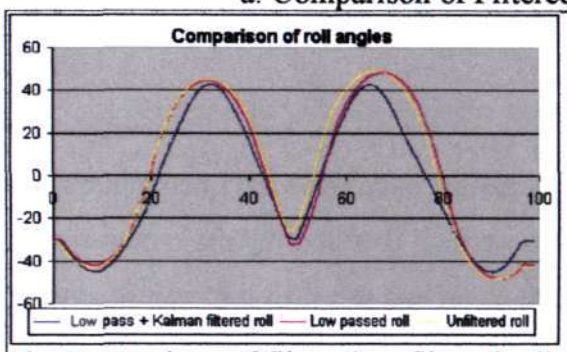
K.F.	$Q_{diagonal}$	$R_{diagonal}$
Angular	[0.03, 0.03, 0.06, 0.06, 0.06, 0.03]	[0.15, 0.14, 0.14, 0.0090, 0.0073, 0.093]
Linear	[0.03, 0.09]	[0.004, 0.02]

Table 5.3 Selected Q and R diagonal matrix for the kalman filter.

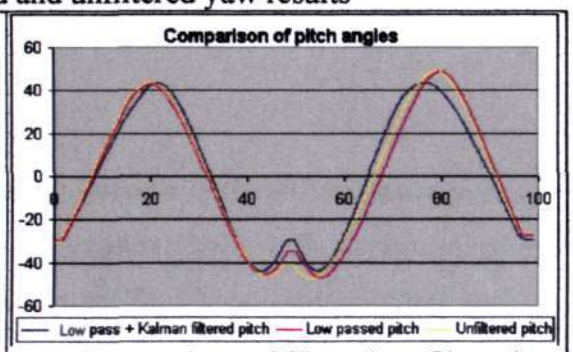
Figure 5.16 shows the results filtered and unfiltered results. From the results, it can be seen that the filtered results further conform to the actual angles. However, even with the all the algorithms implemented, there are still slight variations in the actual results. After the system turns 360°, the filtered result displays only 349°. This is a marked improvement over the unfiltered result of 300° whereby without the algorithms, the URV would have large drifts error. The errors in the roll and pitch angles were also improved. From the results in figure 5.15, the error after the rotation and counter rotation is less than 1°. This is a marked improvement over the unfiltered results of 9° and about 5° with the low pass filter. The implemented algorithms are suitable for the URV.



a. Comparison of Filtered and unfiltered yaw results



b. Comparison of filtered, unfiltered roll



c. Comparison of filtered, unfiltered pitch

Figure 5.15 Graph of filtered and unfiltered results, showing the filtered results is closer to the angles out from the tilt sensors due to the Kalman filtering algorithm.

5.4 Chapter summary

This chapter looked at the performance of the system in terms of processing and the sensors ability to track certain motion and decide the most suitable navigation algorithms to be included in the system. The architecture in the system was based on a multi process approach using a single processor. This is to ensure higher priority processes, will have an immediate control of the CPU. However, it is difficult to evaluate the overall system performance since the timing varies. Therefore, the timing performance was done to minimise the processing time by selecting the most suitable algorithms. Most of the common navigation algorithms that are required for the system are implemented on the system, while others such as sensor to body frame transformation were excluded. Filters such as the low pass filter and kalman filter were implemented to further improve the system. These are selected to ensure that when the algorithms are implemented, the system is able to meet the timing constraints.

CHAPTER SIX

NAVIGATION PERFORMANCE

In implementing the navigation system, the performance of the online system needs to be evaluated and fine-tuned. Experiments are required to test the capability of the system to function as an online navigation system. The designed algorithms need to be robust to prevent malfunction when the algorithms are implemented into the twin-barrel URV. Furthermore, the timings need to be precise to minimise error in data handling. The performance of the navigation system needs to be evaluated and the coefficients of the filters are required to be fine-tuned. This to ensure the navigation system is capable of tracking the actual motion when implemented in the URV.

6.1 Background

To test the navigation performance with online processing, the navigation system can be tested for its path tracking performance and its deviation between its starting and ending position. For this purpose various platform were used to test the system on land and in pool condition. The testing were done in stages, where at each stage, certain parameters were varied while keeping the rest constant. This is to ensure at each stage, proper information can be obtained which would allow further fine-tuning of the system.

6.2 Experiment: Land based navigation performance

Given the limited workspace of the robotic arm, a land-based tracking performance is designed to test the sensor system given a specific path of motion. The purpose of this experiment is to test the system and further fine-tune the online filters. With the system mounted on a trolley as shown in figure 6.1, the sensor system was pushed along a planar path as shown in figure 6.2, with a total distance of 370m.

A planar motion has only 2-DOF, which allows testing with lesser variables before more extensive test can be conducted. During the test, the trolley was maintained at a relatively constant velocity. The experiment was conducted by varying two parameters, namely the speed of motion and the type of turns, one at a time. Two speeds of motion were chosen and on-the-spot turning was compared against a smooth turning.

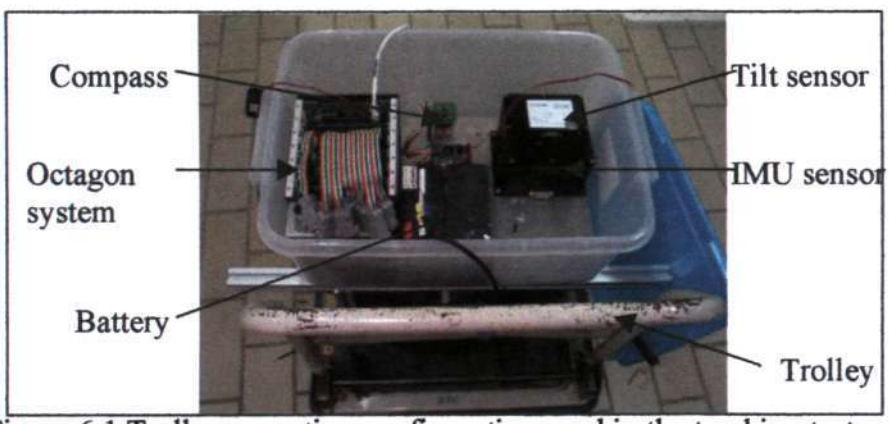


Figure 6.1 Trolley mounting configuration used in the tracking test

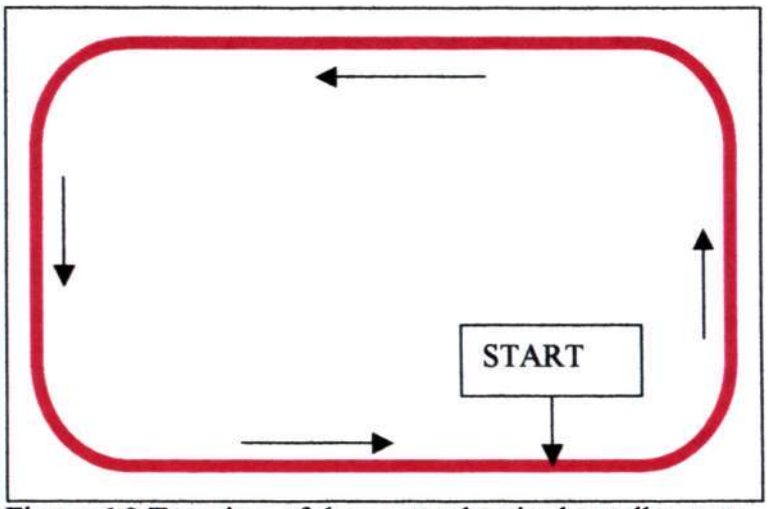


Figure 6.2 Top view of the route taken in the trolley test

On-the-spot turning vs. smooth turning

In this experiment, two different turning methods are compared. In both instances, the trolley is pushed at a relatively constant velocity of 1.82m/s. The first circuit uses an on the spot turning method while the second circuit follows the curved turning motion. Figure 6.3 shows the difference in the plot between the first and second run.

Effect in the speed of motion

In the second test, the speed of motion is varied. Using the smooth turning, the result obtained from the 1.76m/s run is compared against a 3.15m/s run. The results obtained are shown below in figure 6.3.

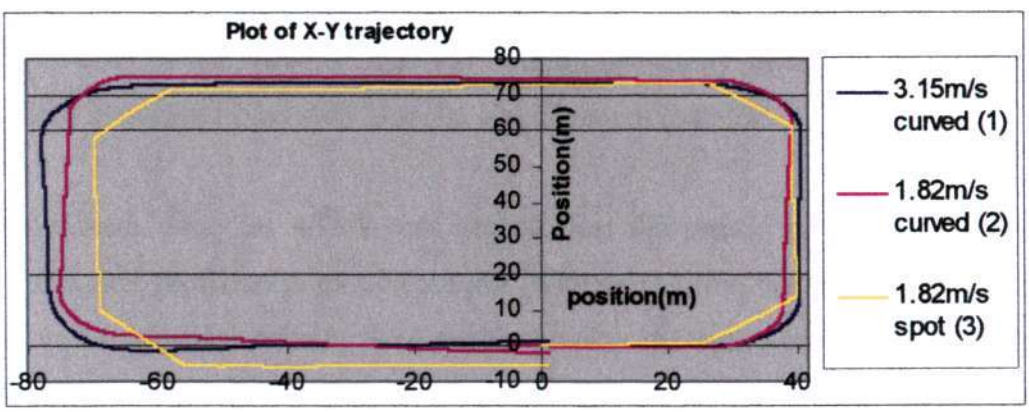


Figure 6.3 Plot of trolley test trajectory on an X-Y plane

Table 6.1 Tabulated error for the path tracking experiment

Path	Distance	Position error	Error WRT. distance	Heading error
1	383	1.45	0.387	2.60
2	380	2.03	0.534	3.40
3	371	5.37	1.44	1.10

The result shows the filtered sensors output is capable of maintaining a reliable heading with only slight divergence in the actual heading. The result from the on-the-spot turning tend to provide better heading as the changes in the yaw rate is of higher magnitude than that of using a smooth turn. Even so, both method of turning is usable for the system.

From the results, all three runs gave quite a reliable description of the tracked path, with minimal error in the track motion. Path two using on the spot turning with lower speed gave the most divergence over the actual heading as such a motion gave the least change in the yaw rate. In terms of final positional error, with the limited equipment, it is assumed that the trolley travels at a constant speed. However, in the final water-based setup, with the velocity Doppler providing the required velocity, improvement in terms of the final positional offset would be seen in the system.

Test using the ATRV

The trolley test has the limitations in which its linear tracking ability is limited and when the trolley is pushed, it is assumed the trolley is moving at a constant velocity. Unlike the trolley, the ATRV is equipped with odometer sensors that can provide the velocity status of the vehicle. This allows the opportunity in testing the merging algorithms between the velocity Doppler and the accelerometers.

An experiment using an ATRV was used to test the sensor system. The ATRV allows a higher precision in motion control. With this ability, the sensor system was tested on a non-planar terrain. Such terrain invokes the entire navigation variables and thus tests the system fully, simulating normal navigation circumstances. Looping a full path allows the comparison of the actual path taken by the ATRV and the path generated by the sensors and tests the deviation of the final position and orientation obtained from the sensors. With the system mounted on an ATRV as shown in figure

6.5(a) and connected as shown in figure 6.4, the sensor system was driven along a non-planar partially inclined car park. The odometry reading obtained from the ATRV is shown in figure 6.5(b).

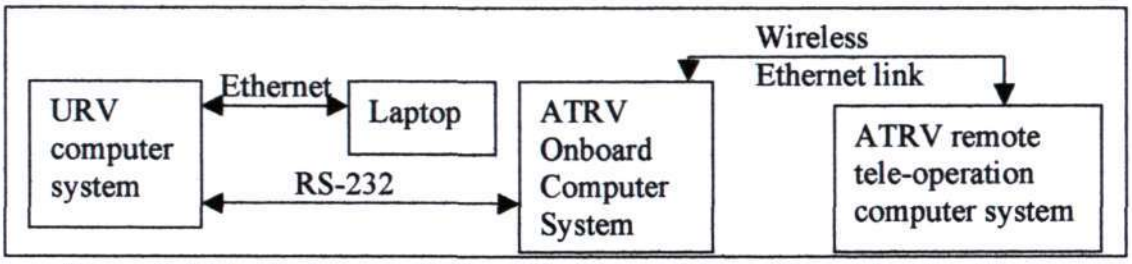
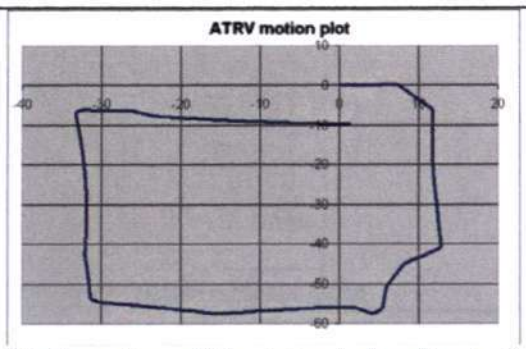


Figure 6.4 Physical connectivity of the Kawasaki experimental setup allowing positional and velocity feedback to the URV computer system

(a) ATRV experimental setup

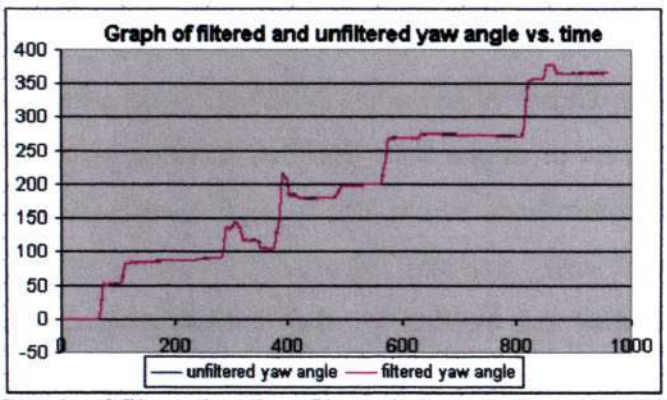


(b) Top view of the intended and actual route taken in the ATRV test

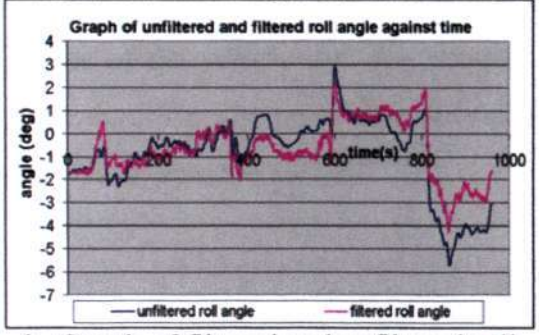
Figure 6.5 ATRV setup and output

Tilts conditions are introduced to the system in this experiment, as the system needs to provide the correct tilt values before the URV control system could provide the counter measure to zero the tilts. In conducting the test, ATRV is tele-operated to move at a speed of 0.2m/s via a remote computer using wireless Ethernet connection, taking an overall time of about 17 minute. At the point the test was conducted, the ATRV system does not have a complete navigation system to maintain a predefined path. The odometers do not take into account motion in the Z-axis, and wheel slippage, unbalanced tyre pressure results in drifts, and thus the recorded positional data obtained from the odometers do not directly describe the ATRV position. As shown in the figure, the actual test did not form a closed loop. The ATRV started on the left side of the road and ended on the right side, which is about 5 meters apart.

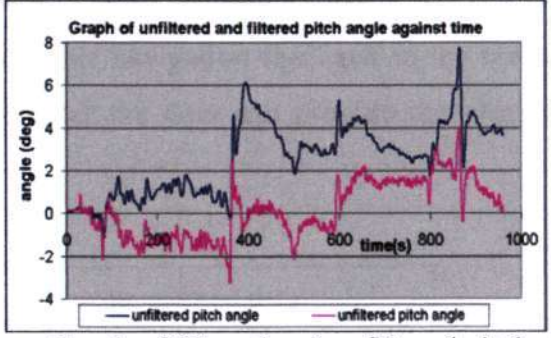
Figure 6.6 shows the results obtained from the angular states of the vehicle, showing the comparison between the unfiltered data and kalman-filtered data. Figure 6.7 shows the results obtained from the linear body attached X-axis motion, showing comparisons between raw acceleration, orientation corrected acceleration, band pass filtered acceleration and figure 6.7(b) shows the kalman-filtered velocity profile. Figure 6.8 shows the vehicle trajectory in the planar X-Y axis and the Z-axis profile against time.



a. Graph of filtered and unfiltered yaw angle against time



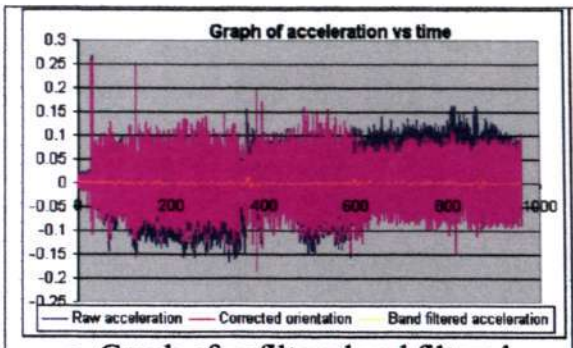
b. Graph of filtered and unfiltered roll angle against time



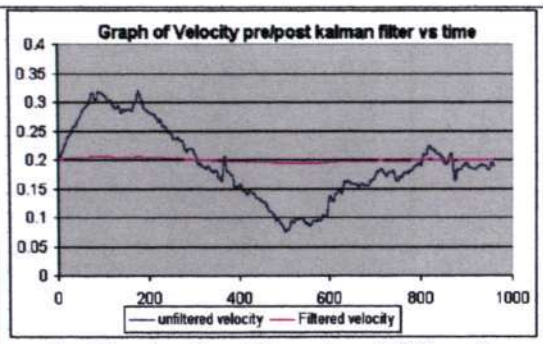
c. Graph of filtered and unfiltered pitch angle against time

Figure 6.6 Graph of angular displacement against time

The graphs shows the filtered data obtained from converting the Euler angular rate transformation and kalman filtered data combining the absolute angles obtained from the tilt sensor and compass. From the results, the integrated values show relatively accurate changes in angles but due to filtration and processing, there are losses in magnitude. The kalman filter result further improves the sensor data by merging the data with that of the tilt sensors and the compass. With the results obtained, it is used to further resolve the accelerations in converting from vehicle to the navigation frame.



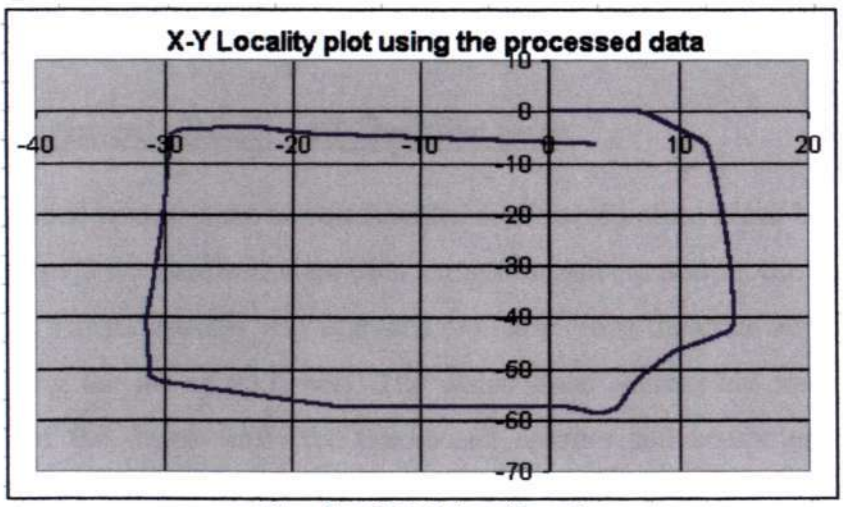
a. Graph of unfiltered and filtered acceleration against time



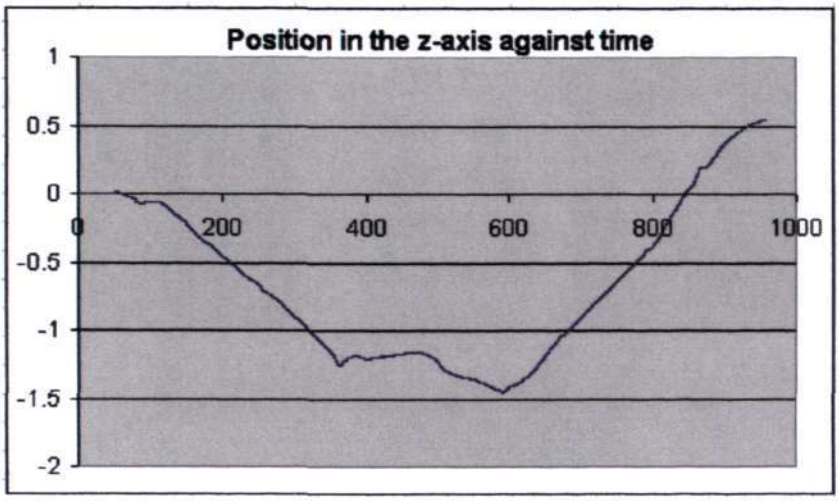
b. Graph of unfiltered and filtered velocity against time

Figure 6.7 Graph of velocity and acceleration against time

The acceleration obtained from the IMU is affected by the roll and pitch angles. The body frame acceleration uses the roll and pitch angles to remove the resolved G-vector. The band pass filtered data, shows minor acceleration readings since the ATRV is moving at a relatively steady velocity. This kalman-filtered data, combining the acceleration and the velocity reflects this with the filtered reading converging towards the steady feedback velocity obtained from the ATRV. This is important since inaccurate integrated results will result in divergence in positional states of the vehicle and thus provide improper navigation feedback to the remote operator. The designed algorithm should filter the data and provide the required means to navigate the URV.



a. Graph of X-Y locality plot



b. Graph of position in the z-axis against time
Figure 6.8 Graph of ATRV x-y-z localities

The locality results from the sensors fairly duplicates the results obtained from the ATRV odometers shown in figure 6.5(b). While the distance covered is only about 200m, the experiment takes about seventeen minutes to complete. Based on the time the sensors are in operation, the sensors and the algorithms used are capable to provide quite an accurate representation of the vehicle position. The z-axis displacement shows the tilted nature of the car park, where the experiment stated at the highest point and the results shows a dip in the tilt angles. Unfortunately, the final z-displacement is about 0.5m higher than the actual height. However, the depth sensor fitted on the URV is capable of correcting this data to obtain an accurate z-axis displacement.

6.1 Experiment: Depth and Doppler tests

The water-based sensors need to be calibrated and tested before it can be integration into the system. Two platforms were built using standard aluminium profile as shown in figure 6.9. The first profile was designed to mount along the drain and constrained to travel along the diving pool wall. The water-based sensors are mounted at the lower part of the frame while the land-based sensors and computer system are mounted above. The second platform is a floating platform, where the water-based sensors are mounted underwater while the land based sensor and computer are mounted in a plastic housing.

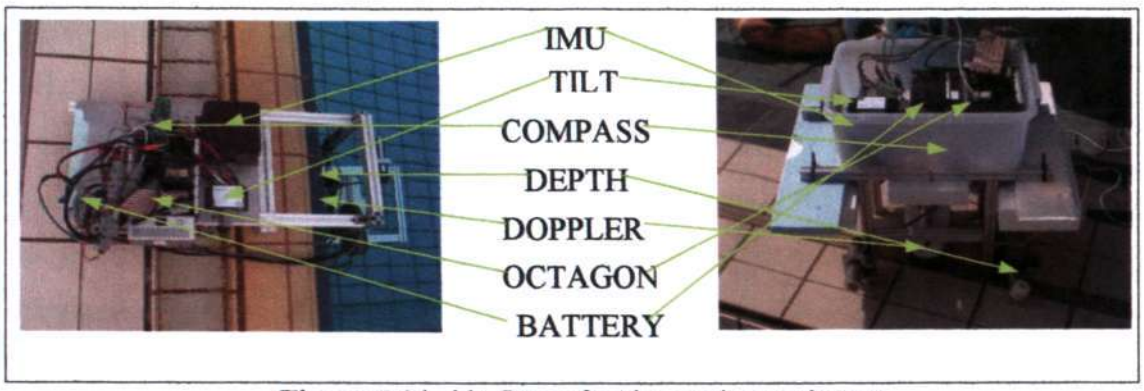


Figure 6.9(a-b): Setup for the pool experiment

The calibration of the sensors is shown in appendix B. The depth sensor requires an offset of positive 0.0134m to correct the zero error. Once the sensors are calibrated, using frame one, the depth sensors are tested under static condition and linear motion. Figure 6.10 shows the filtered and unfiltered depth reading left at a depth of 2m while figure 6.11 shows the depth sensor driven over a distance of 4m at a depth of 0.7m.

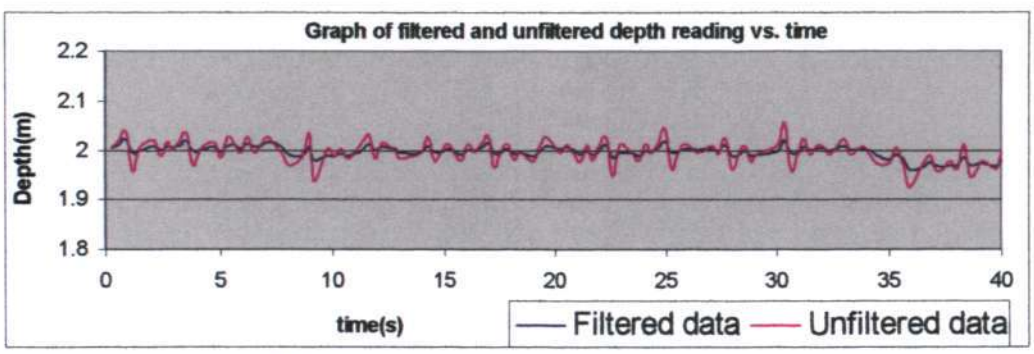


Figure 6.10 Graph of filtered and unfiltered depth reading against time

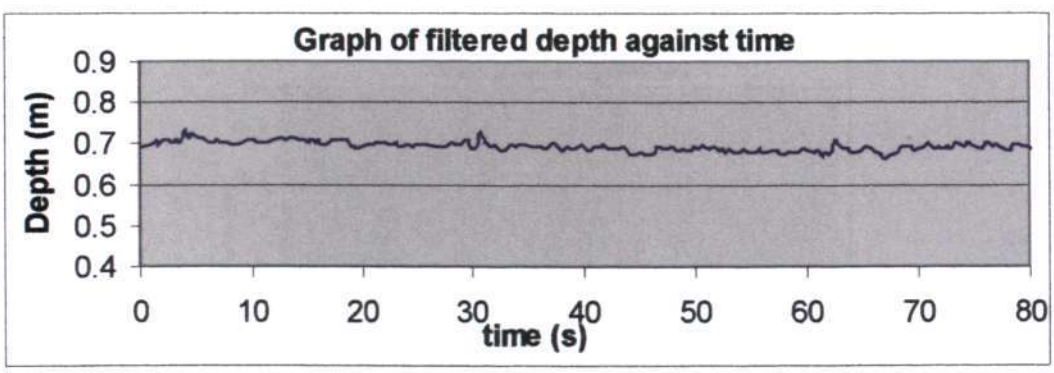
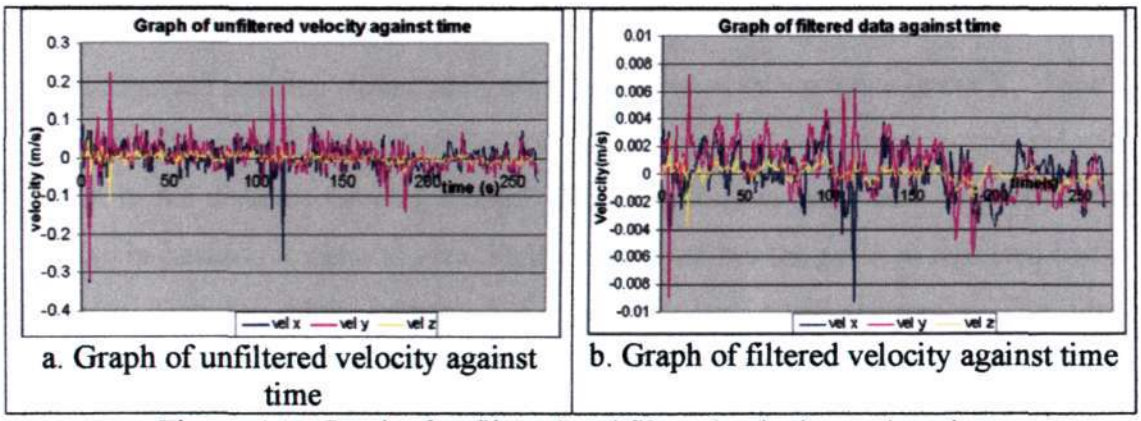


Figure 6.11 Graph of filtered depth against time

From figure 6.10, it can be seen that the filtered and unfiltered data maintained at a relatively constant value of 2m over the given time. From figure 6.11, the depth sensor also maintained a fixed depth while undergoing linear motion. In instances where the URV needs to hover or cruise at a fixed depth, the results shows that, the depth sensor together with the implemented algorithms is suitable for use in the navigation system.

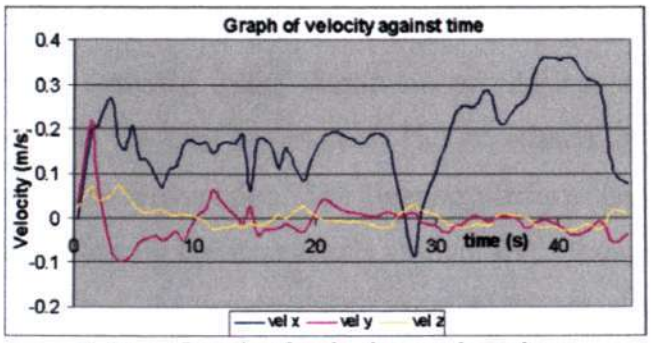
The static and linear motion tests are repeated for the velocity Doppler using the floating platform. In the static test, the floating platform is left in the center of the pool for five minutes, while the floating platform is driven over the diving pool length of 10m in the linear motion test. Figure 6.12 shows the filtered and unfiltered Doppler reading under static condition, while figure 6.13(a) shows the velocity reading driven over a distance of 10m. Figure 6.13(b) and (c) shows the position comparison against time and against the ranging altimeter.



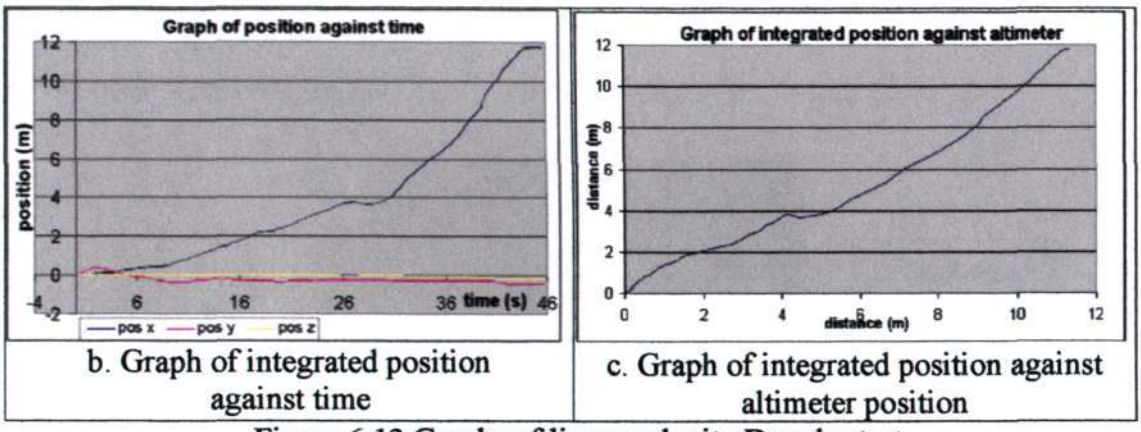
a. Graph of unfiltered velocity against time

b. Graph of filtered velocity against time

Figure 6.12 Graph of unfiltered and filtered velocity against time



6.13 a. Graph of velocity against time



b. Graph of integrated position against time

c. Graph of integrated position against altimeter position

Figure 6.13 Graphs of linear velocity Doppler test

From the results, it can be seen that with the filter, the performance improve from ±0.2 m/s to ±0.01m/s, thus reducing the error in generating the position obtained from the Doppler. In the linear motion test, as can be seen from the graph, since the velocity Doppler only moves in the forward direction, the velocity in the x-axis varied over time. Furthermore, the velocity in the Y-axis and the Z-axis remained nearly close to zero, except for the initial motion.

The integrated position shown in figure 6.13(b) shows the variation in the X, Y and Z position. The sensor gave positive results whereby the integrated x-axis velocity results in the distance traveled in the X direction while in the Y and Z direction, the change in position is close to zero. Figure 6.13(c) shows the position reflected by the velocity Doppler when compared to the altimeter position results. The ideal condition result is when the position displayed by both distance varies linearly, resulting in a straight line. The result obtained was close to being a straight line.

6.2 Experiment: Overall system performance

The floating platform designed using aluminium profile was as shown in figure 6.9(b) was used again in the final experiment. The experiment involves driving the platform under various looped motion. The floating platform has an unconstrained motion, except for a relatively constant z-axis motion. Even though there is minimal motion in the Z-axis, it tests the system ability to define the actual position. The main purpose of the test was to test the ability of the navigation system to function as a unit in tracking the various path of motion. To test the deviation of the, a marker was

used as shown in figure 6.14 whereby the initial and final position and orientation was set to be the same.

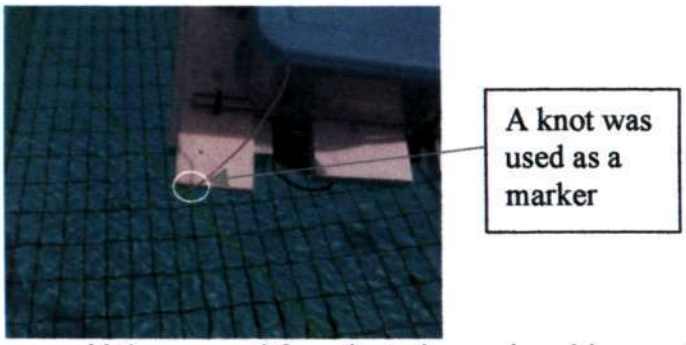
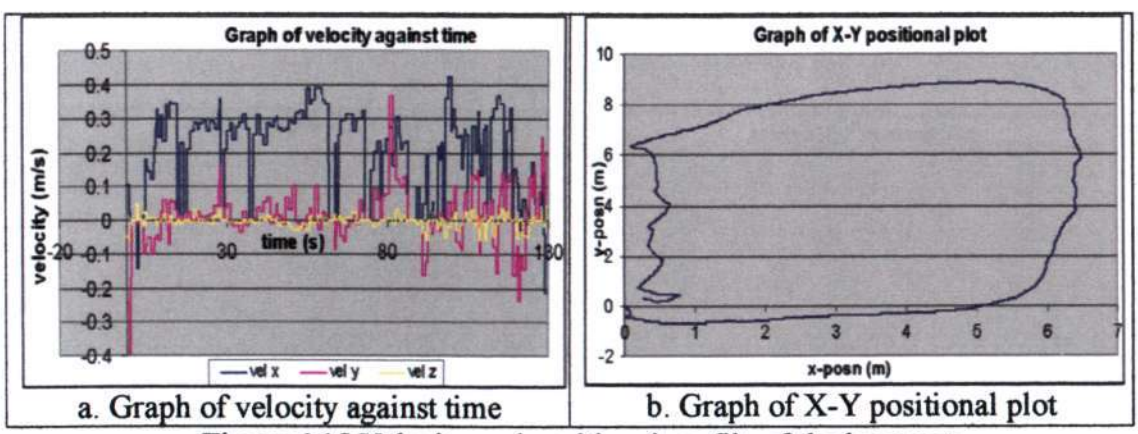


Figure 6.14 Knotted rope, which was used for orientation and position marker

Rectangular loop

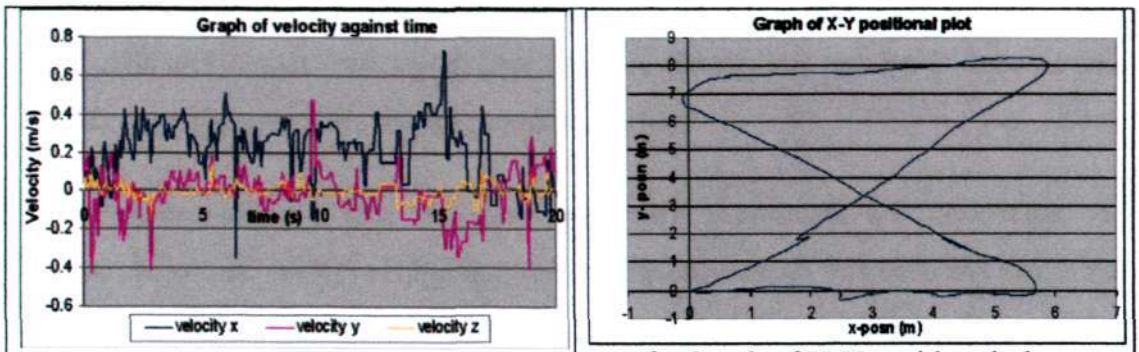
The graph below shows the velocity profile for all the velocity. As expected, Z-axis velocity remains fairly close to zero, as there is no z-axis motion. Since the platform is driven most in the X-axis, the velocity remained mainly between 0.2-0.5m/s. In the Y-axis, since, the motion is not constrained, velocity motion can also be seen. This is mainly result from turns, which also causes Y-axis motion. The graph above shows the X-Y positional plotting of the system. The experiment tests the ability for the system to close a path rather than following a path. Therefore, the shape of the chosen path is only a circular loop. From the test, which took about 2 minutes, the deviation from the original position is only about 0.43m.



a. Graph of velocity against time b. Graph of X-Y positional plot

Figure 6.15 Velocity and positional profile of the loop test

Hourglass loop



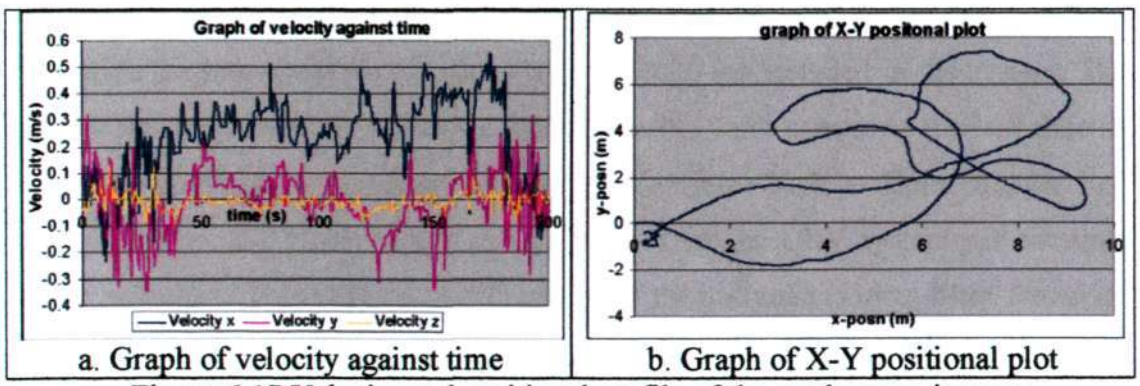
a. Graph of velocity against time b. Graph of X-Y positional plot

Figure 6.16 Velocity and positional profile of the hourglass loop test

The graph above shows the velocity profile for all the velocity. As with the loop test, Z-axis velocity remains fairly zero and the X-axis, the velocity remained mainly between 0.2-0.5m/s and minor motion in the Y-axis. The experiment tests linear motion and the ability for the system to close a path. From the test, which took about 2 minutes, the deviation from the original position is only about 0.122m.

Random motion

The objective of this test is to investigate the performance of the sensor system when it undergoes random motion. This test is done to test the ability of the system to track the motion with the ability of to track to the initial position when the system is returned to the initial position. Such a test will establish the performance of the sensor system.



a. Graph of velocity against time b. Graph of X-Y positional plot

Figure 6.17 Velocity and positional profile of the random motion test

From the above graph, as with the loop test, the velocity in the Z-axis remained relatively constant close to 0m/s. however, unlike the earlier test, the velocity of in the X-axis and Y-axis varies significantly since the random motion is done to take into account even motion driven by external forces.

In the experiment, the total time over which the experiment was conducted was 200sec. As can be seen in from the graph, a random motion was used and the initial and final point of the sensor system was compared. From the graph, the deviation from the initial position is only 0.575m. Compared to the total distance travelled of 57.9m, the deviation in position is only 0.992%. Therefore, as seen from the tabulated result, using the combination of the aided navigation system with the navigation algorithms running on the octagon micro PC is suitable for the primary inertial navigation system of the URV.

Table 6.2 Summary of the final pool path-tracking test

Path	Time (s)	Distance (m)	Position error (m)	Error wrt distance (%)	Heading error (°)
Rectangular	52	29.8	0.337	1.13	2.70
Hour glass	120	33.7	0.122	0.362	1.20
Random	200	57.9	0.575	0.992	2.40

6.3 Chapter summary

This chapter looked at the results obtained from various experiments. It looks at the performance of the system in terms of processing and the sensors ability to track certain motion. The computational timing and the algorithms performance are used to decide the most suitable navigation algorithms to be included in the system. The results show the designed computer software architecture consumes the less time as compared to sequential processing. Certain algorithms such as Euler angular rate transformation and block integration are suitable for the URV system yet consume fewer resources. This validates the suitability of the designed kalman filter. Based on the path tracking and looping tests, the designed filters and the selected algorithms prove to provide a reliable navigation system for the URV. The combined system can be directly ported to the URV for further development of the system in future projects.

CHAPTER SEVEN

DISCUSSION AND CONCLUSION

This chapter concludes the thesis by discussing on the project and problems in implementing the architecture and algorithms. Especially in a real time environment, implementing a multi processes system on a single processor will cause certain stability problems. Furthermore, with multiple processes reading and writing on the shared memory region, the stability may be more apparent. Therefore, this chapter describes the steps taken in overcoming the problems faced in carrying out the project.

6.1 Discussion: General project overview

PC selection

The criteria for PC selection usually depend on each individual application. For this instance, the octagon micro PC was used for the URV. For this application, the main criterion is the size of the PC. Standard and customised PC may have support latest processors and technology. However, the components require a much larger space than the URV permits. On the other hand, usage of microcontrollers requires the tasks of designing specific circuit boards from scratch. Furthermore, in writing the necessary software, most microcontrollers require low-level programming languages. Even if a higher-level compiler is available, available functions are usually limited.

Therefore, the available options are the embedded single board computers (SBC) PC104 and the micro PC that uses a card cage with ISA bus back plane. Both PC specifications are capable of being fitted into the URV. When required, self-designed microcontroller modules can be interfaced to the PC. However, PC104 can support an additional of only four expansion cards while the micro PC is limited only by the available address and interrupt lines. Furthermore, with the support of larger boards, more space is available to fit the circuits for the specialised modules.

Sensor selection

The sensors for the URV are based on a low cost AINS. The sensors used are meant for tracking the URV, which are mainly submerged underwater, therefore it lacks absolute positioning sensor such as GPS. While individually, the sensors are not capable of providing usable long-term data, with the implemented algorithms, it is capable of providing better tracking of the URV position and orientation. However, where long-term application is concerned, secondary sensors may be required to supplement additional information in determining the URV position. This should be the focus of future research for the URV.

OS selection

The main requirement for the OS is in its ability to provide real time processing and scheduling. While common operating systems such as LINUX and WIN NT support real time extensions, it is just an extension and not a legacy RTOS. This may not provide the performance required for the system and furthermore, it requires a large installation space. While there are quite a number of RTOS available, common industrial standard commonly uses either VxWorks or QNX. In comparing these two, VxWorks requires a host PC to code all the required software before it can be ported to an embedded board. On the other hand, QNX consist of a microkernel, which allows additional functions to be added or removed when required. This simplifies the development stage since minor changes are commonly required and it may not be feasible to always carry a host PC.

Serial server architecture

The serial server architecture was designed mainly to manage the sensors, which uses the serial interface. Due to the differences in the acquisition rate, baud rate and data size, the server is designed to manage these processes and prevent any possible system locks, corrupted data, or untimely acquisition. In comparison, common programming technique interfaces each serial port serially one after another. Furthermore, such technique is not suitable in the current application as it takes up a significant amount of processing time.

Shared memory architecture

The shared memory architecture is a common programming technique, which is even used in the Accelerated Graphics Port (AGP) in standard PC. It allows the allocation of certain memory region to be shared between multiple processes. A common technique requires information to be stored by one process and requires inter process communication to share the information between multiple processes. The shared memory architecture eliminates the processing time required for such technique thus improves on the system performance.

Serial connection –wiring and interrupt

The QNX operating system serial interface requires proper serial loop backs to be implemented before proper communication to take place. For the OS to detect and collect the incoming data pin one, four, and six corresponding to Data Carrier Detect, Data Terminal Ready, and Data Set Ready needs to be connected together. Pin seven and eight corresponding to Request To Send and Clear To Send also requires to be connected together in order to enable the data transfer.

On the serial PC interface, the serial server architecture implemented uses a combination of interrupt and polling in order to service the serial devices attached to the system. Implementing such an architecture prevents the need for the system to constantly service the serial port in a fully interrupt mode. By preventing continuous servicing of the serial port, time allocation for the system to service the ports can be balanced with its other processes. In a fully polling architecture, the processor will undergo idle cycles between a poll and the data receiving time of a system. Using the current architecture, this time can be used to service other process while the process enters an interrupt mode and the interrupt is only activated when data is received from the serial port.

Ethernet connection –wiring and protocol

The Ethernet connection for the octagon micro PC allows either RJ45 or thin coaxial cable. While the thin coaxial cable uses only a data and a ground line, it is seldom used in current computer systems. To implement the extended architecture, which was discussed in chapter three, using the RJ45 cable will result in inflexibility in the physical connection since the coaxial cable uses bus, interface and termination is required at both end of the terminal. For both the terminals to perform direct communication, both terminals need to be terminated. This will prevent additional computers to be connected to the onboard micro PC in order to perform any program and algorithm testing. Furthermore, RJ45 connection uses the Ethernet 10BaseT protocol support duplex bi-directional data communication.

Bootup: remote vs. local

QNX operating system allows remote bootup via bootup bios. Since both the systems have the base files of the QNX operating system, by setting one of the nodes as a boot up server and implementing the bootup bios on the other node, the client node can also load the base files from the server nodes. By doing so, space can be saved on the client node for other purposes. Unfortunately, this is not implemented to the system since the base files only occupies a small amount of the 2MB flash drive. Therefore, by using direct booting, time can be saved on the system bootup.

Plug and play architecture for the serial ports

Plug and play serial port architecture would allow the sensors to be detected and configured once it is connected to the system. Unlike the universal serial bus (USB), the serial ports do not have a plug in querying algorithm in order for the system to detect and identify the device. However, since all devices provide its own querying algorithms, this can be implemented into the architecture in order to provide a partial plug and play capability. This can be useful during debugging stage since the hardware are constantly added and removed from the system. However, with the implementation of such system double and multiple pointers are used in the system. This will result in a slow down of the system and thus affect the system performance. Since the final architecture has minimal changes to the hardware configuration, this architecture was not implemented.

Dead bands vs. low pass filters

Dead bands sets a region whereby within this region, the data is considered as noise. LPF on the other hand filters unwanted frequencies, which is considered as noise, thus reducing the overall noise amplitude. However, even with the reduced amplitude, over a long period of time, integrated data may drift due to the noise. Therefore, by implementing a combination of both, this problem will be reduced further. Furthermore, it minimise the possible loss of useful data with a smaller dead band region as compared to implementing a system with just a dead band.

Matrix inversion: Recursive vs. Iteration

In performing a matrix inversion, required for the Kalman filter, the approach used in the project was using the iteration method. However, initial design was done based on a recursive method. Since rounding off takes place when the data are processed, matrix singularity occurs quite often which may result in system locks or incorrect output. Therefore, it was later changed to the iteration method based on the lower upper (LU) decomposition method. Implementing such an algorithm not only results in a shorter programming code, it also increases the robustness and processing speed.

Optimisation of codes

To improve the performance of the system, the programming codes can be optimised to reduce the processing of the system. However, in optimising the codes to improve the performance, the readability of the codes will minimise even with proper documentation from the programmer. For example, a float data type can be replaced with a binary coded decimal. While the programmer can understand the codes, another programmer may require more time than usual in understanding the codes written by the original programmer. The new programmer may also change the codes, which may result in a less optimised code and thus a change of process timing. Therefore, in preventing this from happening, a balance between optimised codes and readability of the codes must be achieved in order to increase performance and yet having readable codes.

Online vs. offline processing

Offline processing always have the advantage during system design as raw data are collected and processing is done on a separate time. However, in most applications, processing is done online while the system is running. During the experimentation stage, the design stage using the Kawasaki robotic arm, processing is done offline to compare between the algorithms and obtain the best algorithms to be implemented to the system. This saves time and ensures consistent input stage. Once the main algorithms have been determined, the designed algorithms are implemented and online processing is done.

6.2 Discussion: Problems encountered

Stability issues (Shared memory), multi processing

One of the main problems of a multi-processes environment is in establishing a stable system for all the processes to work simultaneously and yet function in a stable manner without causing any random locks or system crash. This problem was seen especially in the shared memory architecture or lightly threaded environment. Since both architecture allows the system to share memory region between processes, stability problems may occur when multiple processes are accessing the same memory region simultaneously.

This was overcome by using semaphore combined with proper time allocation to allow each process to complete their critical operations. Even with these two implementations, proper fine-tuning is required in order to establish a fast running system and yet stable over a long period of time. Therefore, every time a new process is added to the system, proper testing is required in order to prevent any stability issues.

Message queue with compiling

QNX message passing implements the message queue as a separate object. In order to implement the message passing routine, a message queue server object is on one of the nodes, and the rest of the nodes are required to refer to this message queue object. Furthermore, when compiling, the program has to be compiled with a (–l mqueue) in order to link the software to the message queue object.

Slow integration rate

As seen from the results, due to the slow processor speed, the integration of the dynamic data such as acceleration, velocity and rotation rate is done at a very slow speed. The average achievable speed is about 40-50ms when the entire sensors programs are running and about 15ms when only the IMU program is running. At 40ms, the rate works out to only about 25 data per second. Since the URV is a sea-based vehicle, this rate should be sufficient since the URV moves at a very slow rate.

Multiple serial ports with limited IRQs

The 5558 Serial card only supports two IRQs to service all its eight ports. The main problem is in identifying which port is causing the interrupt if hardware interrupt routine was used. Besides the hardware interrupt routine, QNX also support software interrupt via proxy. By using proxy to service the serial RS232 interrupt, it will need a slightly higher processing time, but it also reduce the complexity in identifying the device that cause the interrupt.

Semaphore block

While using semaphore can reduce any stability problem, it can also add problem to the system. One of the main problems with using a semaphore is that if multiple processes are using a semaphore and a process do not exit the semaphore state, the rest of the process will enter a semaphore blocked state. This will prevent any other process from running and thus result in the devices attached to the process from being serviced. Furthermore, care is necessary in writing the program since if the semaphore is not released for process without intention, the same problem will also occur.

PCI to ISA bus mapping

The Octagon CPU card has PCI bus architecture, but the back plane is only using ISA bus architecture. In order for all the hardware to work together, the IRQs and the memory address needs to be mapped between the PCI and ISA bus architecture. Since the ISA bus have a limited amount of resources, mapping of resources between the PCI and the ISA bus caused a problem. The system can only address a limited number of devices and the onboard serial ports needs to be changed to non-standard IRQ number to free up the IRQs that can be mapped with the ISA bus.

Message passing synchronisation

In the multi processes environment, synchronisation between processes, especially one on a remote node is necessary in order to prevent process lock. In order to prevent the processes in a blocked state, the message passing routine is established by creating a process specially to be in a receive mode. By doing so, is a process on a remote node intends to send a message on the present node, the process will promptly service the message and reply, thus preventing any possible block states.

Time allocation for each sensor program

Each of the sensors has a specific update rate. Therefore, it is critical in allocating the correct amount CPU of resources in order to service the individual sensors. Improper allocation of resources will result imbalanced servicing between the more and less critical sensors. With a slow update rate, new data may not be updated. Therefore, care is required in the software design to allocate the correct amount of resources per process.

Undocumented functions on the QNX operating system

The QNX operating system that was used with the system is version 4 while the current version is version 6. It was difficult to get support for the operating system. Furthermore, since some functions were not fully described in the documentation, trial and error is used to test the function capabilities. One of these is in programming the RS232 serial ports. In the windows environment, all the RS232 flags can be configured but in QNX direct configuration of the flags can only be achieved by trial and error since it is not documented and the array used in addressing these flags are not labelled.

6.3 Conclusion

This project looked at the implementation of a suitable navigational system for the twin-barrel URV. It looked at the design of suitable software architecture in servicing all the sensors. The serial server architecture was designed to service all the sensors data acquisition by creating individual processes for each of the sensors. Each process was designed to optimise the processor timing by using a combination of interrupt and polling methods. This results in the processes releasing the computer resources while waiting for the full data to be obtained from the sensors.

The serial server was also designed to implement shared memory architecture to manage the data obtained from the sensors. In implementing this architecture, the serial server allows each of the sensor process to access pre-determined memory region. This prevents any error in the data management for the system. An external process can access the data only through the serial server process. This was done to reduce the possibility of error, which may lead to a failure in the system. Furthermore, various process management techniques were implemented such as allocating sufficient time, semaphores and by priority in stabilising the architecture.

Various navigational algorithms were also explored in finding a suitable set of navigational algorithms for the URV. Since certain algorithms require more processing time, the delay may cause higher error in determining the vehicle states. A few of the navigational algorithms were compared in terms of accuracy and impact on the computer system. Based on the performance, the navigation algorithms selected that includes the low pass filter and the Kalman filter were implemented on an identical system. The results obtained from the test shows that the architecture and navigation algorithms are suitable to be used on the actual URV. With improved navigation performance, further development to the URV can be implemented.

6.4 Recommended future works

In continuing with the research, with the implemented navigation system, tests in other areas such as motion control, path planning and implementation of a secondary navigation system can be done. With better determination of the vehicle states, developed control algorithms can be tested on the URV. With this combination, the actual motion and sensor output can be compared to improve the system performance. Based on this performance, better Kalman filter based on the vehicle model can be implemented on the system. From there, path-planning algorithms can be implemented to test the URV as a complete system. To further improve the navigation performance, implementation of secondary navigation sensors such as scanning sonar and GPS can be implemented before it can be deployed in the open waters. Therefore, most of the future works is shown below:

- Implement the architecture and navigation algorithms on the twin-barrel URV
- Implement and improve on the twin-barrel URV control system
- Improve on the Kalman filter based on a sensor-control model
- Implement secondary navigation system for the twin-barrel URV
- Test the twin-barrel URV under open conditions

REFERENCES:

[1] Doyle H., Underwater magazine: "Man & Machine: Divers, ROVs, and High-Tech Solutions to Undersea Missions", Doyle Publishing Company (1994).

[2] Harris B., Boyne W. J. The Navy Times Book of Submarines: A Political, Social, and Military History, Berkley Pub Group (2001).

[3] Yuh J. Design and control of underwater robots: A Survey. Autonomous Robots 8, 7-24, Kluwer Academic Publishers (2000).

[4] A brief history of ROV [online] http://www.rov.org/educational/pages/history.html (1998).

[5] Fong T., Vehicle teleoperation interfaces, IEEE International Conference on Robotics and Automation, Doyle Publishing Company (2000).

[6] Underwater magazine: "Underwater Vehicle Design: Hardware and Software Innovation for ROVs and AUVs", Doyle Publishing Company (1999).

[7] Foot D., Underwater magazine "Business is Booming under the Sea: Issues in Laying Submarine Cable", Doyle Publishing Company (1998).

[8] Kato C., Li L., Nogi Y., Nakamura Y., Tamaoka J., Horikoshi K., Extremely Barophilic Bacteria Isolated from the Mariana Trench, Challenger Deep, at a Depth of 11,000 Meters, Applied and environmental microbiology. Vol. 64, No. 4, p. 1510–1513 (1998)

[9] Westwood, J., Underwater magazine " What the future holds for the ROVs and AUVs", Doyle Publishing Company (2000).

[10] Krauthamer, J. T., Current Status, Future Projections and Inventory: Manned Undersea Vehicles, Remotely Operated Vehicles, Autonomous Undersea Vehicles, Sustained Ocean Observatories and Cruise Ships, Marine Technology Society (2002)

[11] Jones L. B., The future of manned undersea vehicle, Viewports (1997)

[12] Triton XL specifications [online] http://www.geoconsult.no/tritonxl37.htm

[13] Warner E. P., Norton F. H., Accelerometer design, Langley Memorial Aeronautical Laboratory (1921)

[14] Brian C., John D., Optical Fiber Sensors: Applications, Analysis and Future Trends, TA 1800 .O 666 v.4, p.129-157 (1988)

[15] Tan C., Design of an integrated GPS, Gyro-free INS, University of California, California Path (2001)

[16] Yuh J., Chi S. K., Ikehara C., Kim G. H. McMartny G., Design of a semi-autonomous Underwater vehicle for intervention missions (SAUVMI), IEEE symposium on AUV tech. (1998)

[17] Johansen J. History of navigation [Online] Available: http://isa.dknet.dk/_janj/navigation.html, (2001).

[18] Galiano R., Navigate! [Online] Available http://home.att.net/~agligani/navigation/navigate.html, 2000

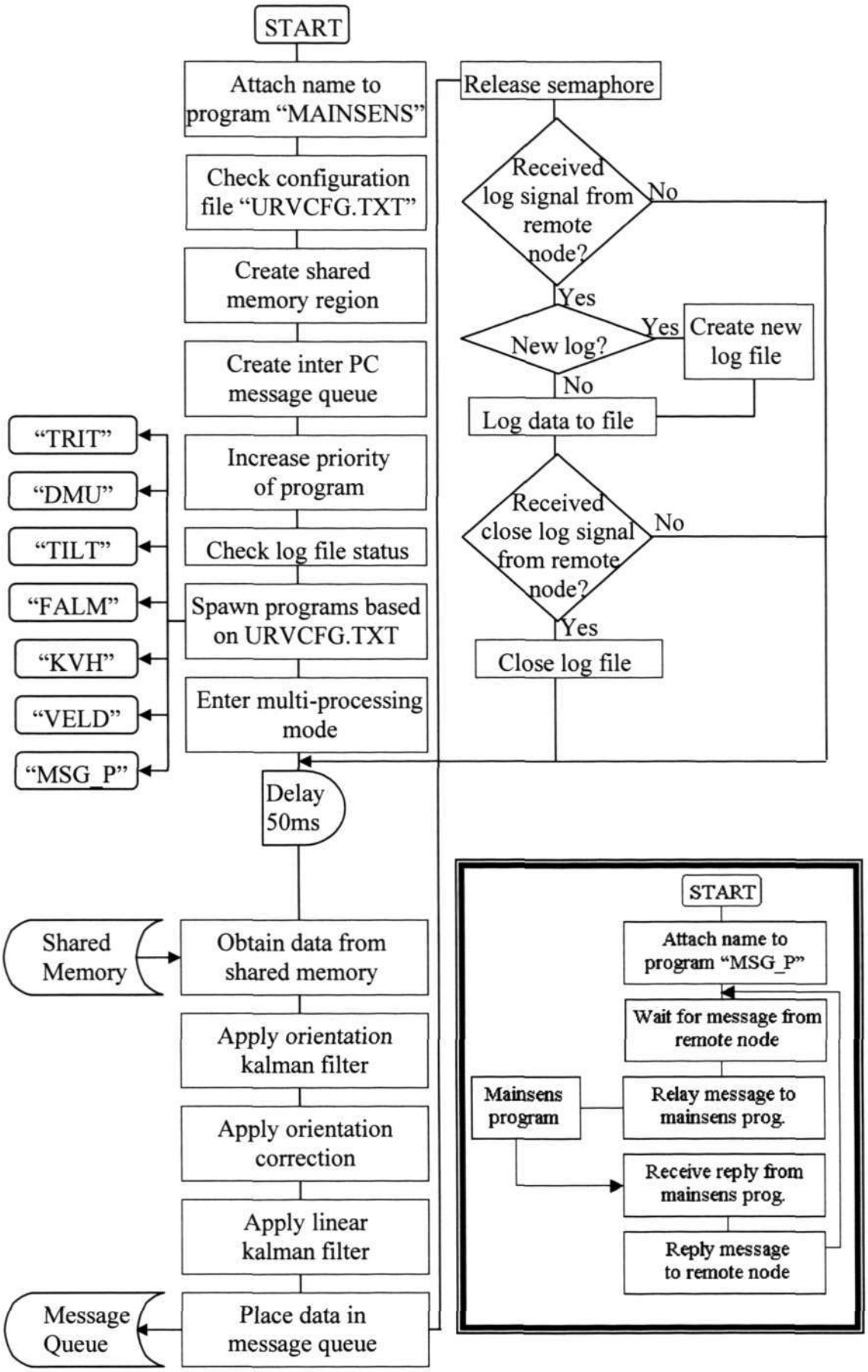
[19] Page S., Frost G., Lachow I., Frelinger D., The Global Positioning System, Critical Technologies Institute, RAND, 1995.

[20] Griffiths G., McPhail S. D., Rogers R., Meldrum D. T. Leaving and returning to harbour with an Autonomous Underwater Vehicle Proceedings of Oceanology International '98 Brighton, UK, Spearhead Exhibitions Ltd (1998)

[21] Charles C. E., Osse T. J., Russell D. L., Timothy Wen, Thomas W. L., Peter L. S., John W. B., Andrew M. C. Seaglider: A Long-Range Autonomous Underwater Vehicle for Oceanographic Research, IEEE journal of oceanic engineering, vol. 26, no. 4, (2001)
[22] Bellingham J., Underwater Robot Tested Beneath the Arctic Ice Sheet, MBARI news (2002)
[23] Robert L. W. AUV's The maturity of technology, Oceans magazine, (1999)
[24] Lucido L., Opderbecke, Rigaud J, Deriche V. Zhang R., An integrated multi-scale approach for terrain referenced underwater navigation, IEEE (1996)
[25] Fong T., Thorpe C., Video teleoperation interface, Autonomous Robots 11, 9–18, 2001, Kluwer Academic Publishers (2001).
[26] G. Griffiths, N. W. Millard, S. D. McPhail, P. Stevenson and P. G. Challenor, On the Reliability of the Autosub Autonomous Underwater Vehicle, Underwater technologyh (2002)
[27] Somajyoti Majumder, Julio Rosenblatt, Steve Scheding, Hugh F. Durrant-Whyte: Map Building and Localization for Underwater Navigation. ISER 2000: 511-520
[28] Marks R., Rock S., and Lee M. Using visual sensing for control of an underwater robotic vehicle. IARP Second Workshop on Mobile Robots for Subsea Environments, Monterey, USA (1994).
[29] Amat, J.; Batlle, J.; Casals, A. & Forest, J.; "GARBI: the UPC Low Cost Underwater Vehicle", Joint US / Portugal Workshop in: Undersea Robotics and Intelligent Control, march, 2 - 3 / 1995, Lisboa, Portugal, pp. 91 - 97.
[30] Mcphail S. Development of a Simple Navigation system for the Autosub Autonomous Underwater Vehicle, IEEE Oceans, Volume 2 (1993)
[31] Ferguson, J., Pope, A., Butler, B., Verrall, R., "Theseus AUV - Two Record Breaking Missions", Sea Technology Magazine, pp. 65-70, February, 1999.
[32] F. Napolitano, T. Gaiffe, Y. Cottreau, T. Loret, PHINS: the first high performances inertial navigation system, based on fibre optic gyroscopes, Proceedings St Petersbourg International Conference on Navigation Systems. 2002
[33] Brandon T., Navigation (Theory and practice), Portland state university (2000)
[34] Dorota A. Grejner-Brzezinska and Jin Wang, Gravity Modeling for High-Accuracy GPS/INS Integration, Navigation, 1998, Vol. 45, No. 3, pp.209-220 (1998)
[35] Giaffe T., Underwater Magazine "Ixsea's AUV navigation system" Available [online] http://www.diveweb.com/rovs/features/021.02.htm (2002)
[36] NCAR Advanced Study Program, available [online] http://www.asp.ucar.edu/colloquium/1992/notes/part1/node1.html (1992)
[37] Kopetz, H., Real-Time Systems, Design Principles for Distributed Embedded Applications, Klower Academic Publishers, (1997)
[38] Juvva K., Real time systems, Carnegie Mellon University (1998)
[39] T. Bihari, R. McGhee, Luqi, & Y. Lee -- "Applying a Computer Aided Prototyping System to the Software of an Autonomous Underwater Vehicle," Position paper at Workshop on Software Tools for Distributed Intelligent Control, (1990)
[40] K. Vestgård, R. A. Klepaker, N. Størkersen, HUGIN 3000 AUV For Deep Water Surveying, OTC 2000, (2000)

[41] J. Rosenblatt, S.B. Williams and H.F. Durrant-Whyte, "A Behavior-Based Architecture for Autonomous Underwater Exploration", International Journal of Information Sciences, Vol. 145(1-2), pages 69-87, (2002)
[42] Chanop Silpa-Anan and Alexander Zelinsky. "Kambara, Past, Present, and Future" Australian Conference on Robots and Automation, Sydney, Australia, November 2001.
[43] Williams S. B., Newman P., Rosenblatt J., Dissanayake G., and Durrant-Whyte H. F., "Autonomous Underwater Navigation and Control", Robotica, vol.19, no.5, pp.481-496, (2001).
[44] C. D. Richards, "The Engineering Handbook", CRC Press, Section 21, LLC , (2000)
[45] Kaliardos, W., Cosgrove, M., Chow, T., Steiner, S., Farritor, S., "Companion - Overview of a Telerobotic Ground Vehicle," Association for Unmanned Vehicle Systems International (AUVSI), Orlando, FL, (1996)
[46] E. S. Christopher, "The RS232 Standard" CAMI Research Inc., Lexington, Massachusetts, (1993)
[47] Octagon computer systems corporation, Available [online] http://www.octagonsystems.com (1993)
[48] QNX software systems limited, Available [online] http://www.qnx.com (1997)
[49] M. Lyu, "Requirement specifications for a redundant strapped down inertial measurement unit", Chinese University of Hong Kong, 2004
[50] W. Lowrie, "Fundamentals of Geophysics", Cambridge University Press, 1997
[51] J. Rife and S. M. Rock, "Field Experiments in the Control of a Jellyfish Tracking ROV", IEEE/MTS OCEANS, 2002.
[52] Rife, J. and S.M. Rock "A pilot-aid for ROV-based tracking of gelatinous animals in the midwater". MTS/IEEE Oceans, 2001.
[53] B. T. Boulter, "Writing Difference Equations For Digital Filters", applied industrial control solutions, ApICS LLC., 2000
[54] B. W. Bomar, L. M. Smith "Digital Filters", The Engineering Handbook, Chapter 125, CRC Press LLC, 2000.
[55] S. G. Mohinder, R. W. Lawrence, "Global positioning systems, Inertial Navigation and Integration", John Willey & Sons, Inc 2001.
[56] D. H. Titterton, J. L. Weston, "Strapdown Inertial Navigation Technology", IEE Radar, Sonar, Navigation and Avionics, No 5, 1997.
[57] L. H. David, L. James, "Handbook of multi sensor data fusion", CRC press, 2001.
[58] E.A. Wan and R. van der Merwe, "The Unscented Kalman Filter for Nonlinear Estimation," IEEE Symposium 2000 (AS-SPCC) 2000.
[59] E. Nebot and H. F. Durrant-Whyte, "Initial Calibration and Alignment of Low Cost Inertial Navigation Units For Land Vehicle Applications." Journal of Robotic Systems, Vol. 16, No. 2, pages 81 to 92, 1999.
[60] B. Barshan and H. F. Durrant-Whyte "Inertial Navigation Systems for Mobile Robots", IEEE Transactions on Robotics and Automation, Vol. 11, No. 3, 1995
[61] P. Luethi, T. Moser "Design and Characterisation of a Strap down Inertial Navigation System Based on Low Cost Sensors" Electronics Laboratory of the Swiss Federal Institute of Technology, Zurich, Switzerland, 2000

APPENDIX A
PRIMARY
PROGRAM CODES



A1) Serial server program

```
#include "kalman.h"
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>
#include <sys/vc.h>
#include <sys/name.h>
#include <time.h>

#define BILLION  1000000000L
#define deg2rad 0.0174533

float           *dmu, *trit, *falm, *kvh, *svn, *tlt, *vel, *dta;
unsigned short  *sptr;
mqd_t           mq_des, sn_des;
char            stack[4096], vcr[3], vcs[3], writeflag=0;

//Create a threaded process for passing signals between the navigation and control pod
void thread(void*arg){
        pid_t           ND1_PD;
        qnx_name_attach(0, "SENSPOD");
        ND1_PD = Receive(0, &vcr, sizeof(vcr));
        Reply(ND1_PD, &vcs, sizeof(vcs));
        for(;;){
                Receive(ND1_PD, &vcr, sizeof(vcr));
                vcs[0]=vcr[0];
                Reply(ND1_PD, &vcs, sizeof(vcs));
                writeflag = vcr[0];}}

//******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag
//data structure 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn, 6x:tlt, 7x:vel
//******************
//Create a shared memory area for other processes to access
//it checks if the memory object is available and map it. If not it creates a new one
//also maps the semaphore protocol
void shm_make(void){
```

```
        int shm_fd1;
        shm_unlink("SHAR");
        shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777);
        ltrunc(shm_fd1,40960,SEEK_SET);
        sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0);
        dmu=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,4096);
        trit=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,8192);
        falm=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,12288);
        kvh=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,16384);
        svn=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,20480);
        tlt=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,24576);
        vel=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,28672);
        dta=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,32768);
        sem_init(&sptr[2000],1,1);
        return;}

//********************** main program***********************************

//main program
//checks how the program is started via the argc, argv
//Create the threaded process, shared memory and intialise the data logging algorithm
//Spawn all the required processes
//Obtain the data deposited by the process in the shared memory in a timely manner
//Apply kalman filter and process the data into the navigational frame to obtain the position
//Log the data to file if signal is activated by the user and end loop
void main(int argc, char *argv[]){
        pid_t           KVH_PD, FALM_PD,TRIT_PD, DMU_PD, SVN_PD,TLT_PD,
VEL_PD, TFK_PD;
        char            buffer1[25], buffer2[5], filein=0;
        float           kalm6[6], kalm2[6];
        float           sx, cx,   sy, cy,   sz, cz, dx1, dy1, dz1;
        int                     fileno;
        struct timespec started, stoped;
        unsigned short  jj;
        FILE            *fp1;
        qnx_name_attach(0, "MAINSENS");

        if(argc>1) cprintf("*** kalman ***\n");
        shm_make();
        TFK_PD=tfork(&stack, sizeof(stack), &thread,(void*)1,0);
        setprio(TFK_PD,(getprio(0)+1)); //tfork to receive signal from the remote computer

        for(jj=1;jj<65535;jj++){
                fileno =jj*100;
                itoa(fileno,buffer2,10);
                strcpy(buffer1, "//3/home/LOG/");
                strcat(buffer1, buffer2);
                strcat(buffer1, ".txt");
                fp1 = fopen(buffer1, "r");
                if(fp1 == NULL) break;
                fclose(fp1);} //create file

        if((filein & 1) == 0)       DMU_PD = spawnl( P_NOWAIT, "//3/PODPROGS/dmupool",
"dmupool", "1", NULL);
        if((filein & 2) == 0)       TRIT_PD = spawnl( P_NOWAIT, "//3/PODPROGS/tritpool",
"tritpool", "1", NULL);
        if((filein & 4) == 0)       FALM_PD = spawnl( P_NOWAIT, "//3/PODPROGS/falmpool",
"falmpool", "1", NULL);
```

```
        if((filein & 8) == 0)       KVH_PD = spawnl( P_NOWAIT, "//3/PODPROGS/kvhpool",
"kvhpool", "1", NULL);
        if((filein & 16) == 0)      SVN_PD = spawnl( P_NOWAIT, "//3/PODPROGS/svnpool",
"svnpool", "1", NULL);
        if((filein & 32) == 0)      TLT_PD = spawnl( P_NOWAIT, "//3/PODPROGS/tiltpool",
"tltpool", "1", NULL);
        if((filein & 64) == 0)      VEL_PD = spawnl( P_NOWAIT, "//3/PODPROGS/veldpool",
"velpool", "1", NULL);
        delay(280); //spawn all the necessary data acquisition programs
        clock_gettime(CLOCK_REALTIME, &started);

        for(;;){
                delay(50);
                sem_wait(&sptr[2000]);
                clock_gettime(CLOCK_REALTIME, &stoped);
                dta[15]=(float)(stoped.tv_sec-started.tv_sec)+(float)(stoped.tv_nsec-
started.tv_nsec)/BILLION;
                kalm6[0]=dmu[(sptr[11])]; kalm6[1]=dmu[(sptr[11])+1];
kalm6[2]=dmu[(sptr[11])+2];
                kalm6[3]=tlt[(sptr[61])]; kalm6[4]=tlt[(sptr[61])+1]; kalm6[5]=kvh[(sptr[41])];
                if(argc>1) kalman6(kalm6, dta[15]);

                // do sin/cos conversion here
                sx = sin(kalm6[3]*deg2rad); cx = cos(kalm6[3]*deg2rad);
                sy = sin(kalm6[4]*deg2rad); cy = cos(kalm6[4]*deg2rad);
                sz = sin(kalm6[5]*deg2rad); cz = cos(kalm6[5]*deg2rad);

                // do IMU AX, AY, Az orientation correction here
                dmu[(sptr[11]+3)] -= sy; dmu[(sptr[11]+4)] += cy*sx; dmu[(sptr[11]+5)] -= cy*cx;

                kalm2[0]=dmu[(sptr[11]+3)]; kalm2[1]=vel[(sptr[71])];
kalm2[2]=dmu[(sptr[11]+4)];
                kalm2[3]=vel[(sptr[71]+1)]; kalm2[4]=dmu[(sptr[11]+5)];
kalm2[5]=vel[(sptr[71]+2)];
                if(argc>1) kalman2(kalm2, dta[15]);

                dta[0]=kalm6[0];        //rr IMU
                dta[1]=kalm6[1];        //pr IMU
                dta[2]=kalm6[2];        //yr IMU
                dta[3]=kalm6[3];        //roll TILT
                dta[4]=kalm6[4];        //pitch TILT
                dta[5]=kalm6[5];        //yaw KVH
                dta[6]=kalm2[0];        //ax IMU
                dta[7]=kalm2[2];        //ay IMU
                dta[8]=kalm2[4];        //az IMU
                dta[9]=kalm2[1];        //vx VEL
                dta[10]=kalm2[3];               //vy VEL
                dta[11]=kalm2[5];               //vz VEL
                dx1=dta[9]*dta[15];             //delta px intgrated from vx
                dy1=dta[10]*dta[15];            //delta py intgrated from vy
                dz1=dta[11]*dta[15];            //delta pz intgrated from vz
                dta[12]+= (dx1*cy*cz + dy1*(sx*sy*cz-cx*sz) + dz1*(cx*sy*cz+sx*sz));
        //px + orientation corrected delta px
                dta[13]+= (dx1*cy*sz + dy1*(sx*sy*sz+cx*cz) + dz1*(cx*sy*sz-sx*cz));
        //py + orientation corrected delta px
                dta[14] = falm[(sptr[31])] + (-sy*dx1 + dy1*(sx*cy) + dz1*(cx*sy));    //depth
(from pressure)
                dta[16]=trit[(sptr[21])];    //trit altimeter
```

```
                sem_post(&sptr[2000]);
                mq_send(mq_des, dta, 80, 0);

                started.tv_sec = stoped.tv_sec;
                started.tv_nsec = stoped.tv_nsec;
                if(writeflag == 1){
                        dta[12] = 0; dta[13] = 0;
                        fclose(fp1);
                        strset(buffer1, ' ');
                        itoa(fileno,buffer2,10);
                        strcpy(buffer1, "//3/home/LOG/");
                        strcat(buffer1, buffer2);
                        strcat(buffer1, ".txt");
                        fp1 = fopen(buffer1, "a");
                        fprintf( fp1, "rr, pr, yr, r, p, y, ax, ay, az, vx, vy, vz, px, py, pz, ts, alt.\n");
                        writeflag=2;
                        fileno++;}
                if(writeflag == 2){
                                sem_wait(&sptr[2000]);
                                for(jj=0;jj<17;jj++) fprintf( fp1, "%f, ", dta[jj]);
                                fprintf( fp1, "\n");
                                fflush(fp1);
                                sem_post(&sptr[2000]);}
                if(writeflag == 3){
                        fclose(fp1);
                        writeflag=0;}} //write to file
        return;}
```

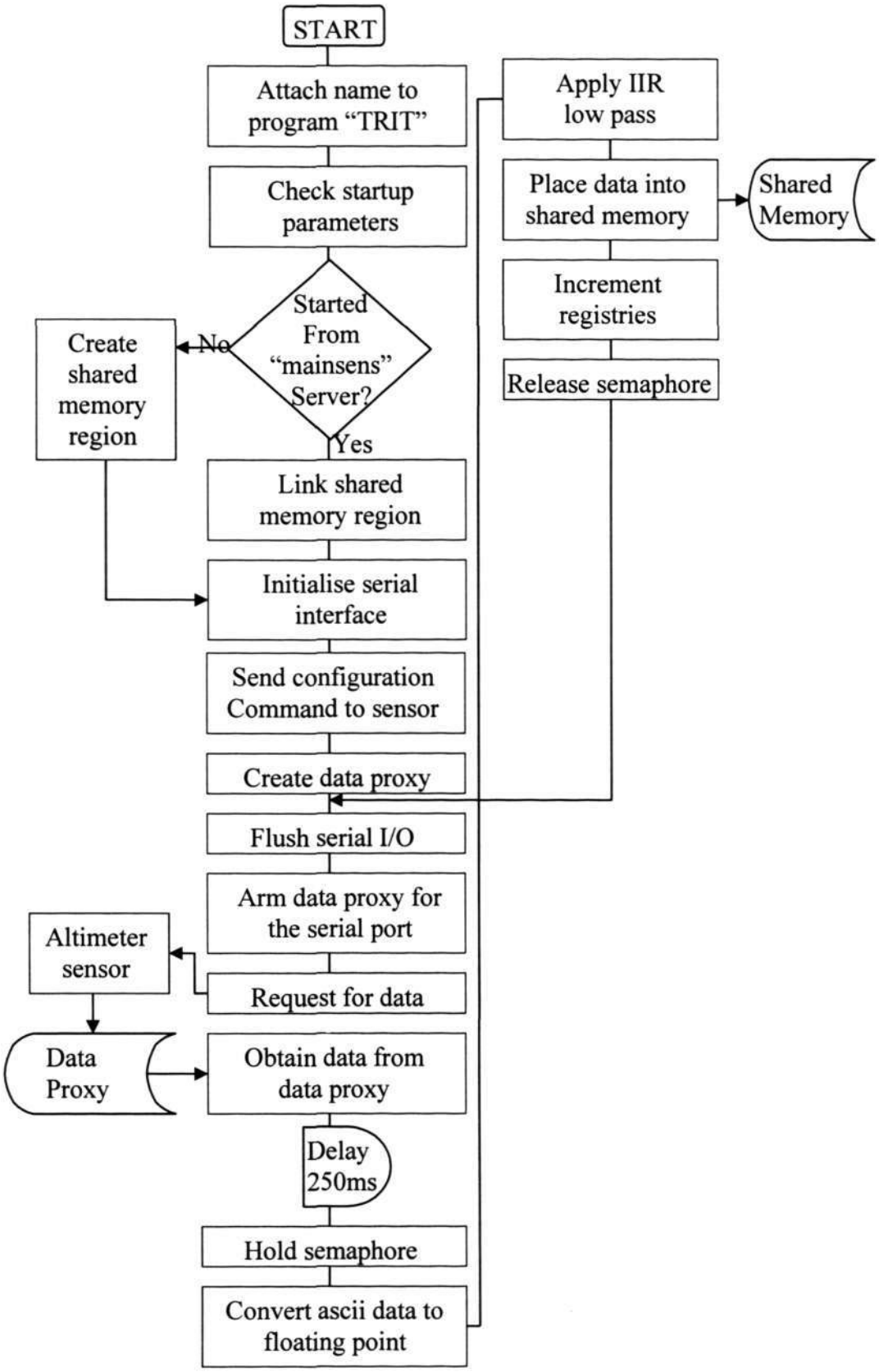
KALMAN.H

```
#include <stdio.h>
#include <math.h>
#define TINY 1.5e-16
#define DEG2RAD 0.01745328

int LUDCMP(float *A, int *INDX, int *d)  {
        float AMAX,DUM, SUM, VV[6];
        int  I,IMAX,J,K;
        *d = 1;

        for  (I=0; I<6; I++){
                AMAX=0.0;
                for (J=0; J<6; J++) if (fabs(A[I*6+J]) > AMAX)  AMAX=fabs(A[I*6+J]);
                if(AMAX < TINY)  return 1;
                VV[I] = 1.0 / AMAX;} // i loop
        for (J=0; J<6;J++){
                for (I=0; I<J; I++){
                        SUM = A[I*6+J];
                        for (K=0; K<I; K++) SUM = SUM - A[I*6+K]*A[K*6+J];
                        A[I*6+J] = SUM;} // i loop
                AMAX = 0.0;
                for (I=J; I<6; I++){
                        SUM = A[I*6+J];
                        for  (K=0; K<J; K++) SUM = SUM - A[I*6+K]*A[K*6+J];
                        A[I*6+J] = SUM;
                        DUM = VV[I]*fabs(SUM);
                        if (DUM >= AMAX){
                                IMAX = I;
                                AMAX = DUM;}} // i loop
```

```
                if (J != IMAX){
                        for (K=0; K<6; K++){
                                DUM = A[IMAX*6+K];
                                A[IMAX*6+K] = A[J*6+K];
                                A[J*6+K] = DUM;} // k loop
                        *d = -*d;
                        VV[IMAX] = VV[J];}
                INDX[J] = IMAX;
                if (fabs(A[J*7]) < TINY) A[J*7] = TINY;
                if (J != 6){
                        DUM = 1.0 / A[J*6+J];
                        for (I=J+1; I<6; I++) A[I*6+J] *= DUM;}} // j loop
        return 0;}

void LUBKSB(float *A, int *INDX, float *B){
        float SUM;
        int  I,II=0,J,LL;
        for (I=0; I<6; I++){
                LL = INDX[I];
                SUM = B[LL];
                B[LL] = B[I];
                if (II != -1) for (J=II; J<I; J++) SUM = SUM - A[I*6+J]*B[J];
                else if (SUM != 0.0)  II = I;
                B[I] = SUM;} // i loop
        for (I=5; I>-1; I--){
                SUM = B[I];
                if (I < 5) for (J=I+1; J<6; J++) SUM = SUM - A[I*6+J]*B[J];
                B[I] = SUM / A[I*7];}} // i loop

void matinv(float *AA){
        float A1[36],SUM;     // copy of matrix A
        float Y[36];      // vector n+1
        float temp[6];    // vector n+1
        int  INDX[6];     // vector n+1
        int d=0, i, j, k, rc;
        for(i=0;i<6;i++){
                for(j=0;j<6;j++){
                        A1[i*6+j]=AA[i*6+j];
                        Y[i*6+j]=0.0;}
                Y[i*7] = 1.0;}
        rc = LUDCMP(AA,INDX,&d);
        if (rc==0){
                for (j=0; j<6; j++){
                        for (i=0; i<6; i++) temp[i]=Y[i*6+j];
                        LUBKSB(AA,INDX,temp);
                        for (i=0; i<6; i++) Y[i*6+j]=temp[i];}}
        for(i=0;i<6;i++){
                for(j=0;j<6;j++){
                        AA[i*6+j]=Y[i*6+j];}}
        return;}

void kalman6(float *y, float dt){
        char i,j, k;
        static char l=1;
        float kA[36], t1[36], P0[36], A[36];
        unsigned short Q[6] = {0.03,0.03,0.03,0.06,0.06,0.06};
        unsigned short R[6] = {0.15,0.14,0.14,0.0090,0.0073,0.093};
        static float xA0[6], xA[6], P[36];
```

```
//get initial sensor readings put into y
        if(l){    for(i=0;i<6;i++) xA0[i] = y[i];
                  l=0;}
        for(i=0;i<36;i++) A[i]=0; for(i=0;i<36;i+7) A[i]=1;
        A[4*6+0] = dt;
        A[4*6+1] = sin(xA0[3]*DEG2RAD) * tan(xA0[4]*DEG2RAD) * dt;
        A[4*6+2] = cos(xA0[3]*DEG2RAD) * tan(xA0[4]*DEG2RAD) * dt;
        A[5*6+1] = cos(xA0[3]*DEG2RAD) * dt;
        A[5*6+2] = -sin(xA0[3]*DEG2RAD) * dt;
        A[6*6+1] = sin(xA0[3]*DEG2RAD) * dt / cos(xA0[4]*DEG2RAD) ;
        A[6*6+2] = cos(xA0[3]*DEG2RAD) * dt / cos(xA0[4]*DEG2RAD);

        for(i=0;i<6;i++){
                xA[i]=0;
                for(j=0;j<6;j++) xA[i] += A[i*6+j] * xA0[j];}  //xA = A*xA0; {11111}

        A[4*6+3]=xA0[1]*cos(xA0[3]*DEG2RAD)*tan(xA0[4]*DEG2RAD)*dt
-xA0[2]*sin(xA0[3]*DEG2RAD)*tan(xA0[4]*DEG2RAD))*dt;
        A[4*6+4]=A0[1]*sin(xA0[3]*DEG2RAD)/cos(xA0[4]*DEG2RAD)*cos(xA0[4]*DEG2RA
D)+xA0[2]*cos(xA0[3]*DEG2RAD)/cos(xA0[4]*DEG2RAD)*cos(xA0[4]*DEG2RAD))*dt;
        A[5*6+4] = (-xA0[2]*cos(xA0[4]*DEG2RAD))*dt;
        A[6*6+3] = (xA0[1]*cos(xA0[3]*DEG2RAD)*cos(xA0[4]*DEG2RAD)
-xA0[2]*sin(xA0[3]*DEG2RAD)*cos(xA0[4]*DEG2RAD))*dt;
        A[6*6+4] = (-xA0[1]*sin(xA0[3]*DEG2RAD)*sin(xA0[4]*DEG2RAD)
-xA0[2]*cos(xA0[3]*DEG2RAD)*sin(xA0[4]*DEG2RAD))*dt;

        for(i=0;i<6;i++){
                for(j=0;j<6;j++){
                        t1[i*6+j]=0.0;
                        for(k=0;k<6;k++) t1[i*6+j] += A[i*6+k] * P[k*6+j];}}
        for(i=0;i<6;i++){
                for(j=0;j<6;j++){
                        P[i*6+j]=0.0;
                        for(k=0;k<6;k++) P[i*6+j] += t1[i*6+k] * A[j*6+k];}}
        for(i=0;i<6;i++) P[i*7] += Q[i]; //P0= A*P*AT + Q {22222}
        for(i=0;i<6;i++) for(j=0;j<6;j++) t1[i*6+j] = P[i*6+j];
        for(i=0;i<6;i++) t1[i*7] += R[i];
        matinv(t1);
        for(i=0;i<6;i++){
                for(j=0;j<6;j++){
                        kA[i*6+j]=0.0;
                        for(k=0;k<6;k++) kA[i*6+j] += P[i*6+k] * t1[k*6+j];}} // kA =
P*iNV(P+R)

//read sensor value put insude y(j)
        for(i=0;i<6;i++){
                xA0[i]=xA[i];
                for(j=0;j<6;j++) xA0[i] += (kA[i*6+j] * (y[j] - xA[j]));}  //xA0= xA + kA * (y - xA)
        for(i=0;i<6;i++) y[i] = xA0[i];
        for(i=0;i<6;i++) kA[i*7] = kA[i*7]-1;
        for(i=0;i<36;i++) kA[i] = -kA[i];

        for(i=0;i<6;i++){
                for(j=0;j<6;j++){
                        t1[i*6+j]=0.0;
                        for(k=0;k<6;k++) t1[i*6+j] += kA[i*6+k] * P[k*6+j];}}
        for(i=0;i<6;i++) for(j=0;j<6;j++) P[i*6+j] = t1[i*6+j];
```

```
        return;} //P = (I - kA) * P

void kalman(float *kalm2, float dt, float *xA, float *xA0, float *R, float *Q, float *P){
        float t1[4], kA[4], inv;
        xA[0] = xA0[0] + xA0[1]*dt;    xA[1] = xA0[1]; //X=A.X0
        t1[0]=P[0];    t1[1]=P[1];    t1[2]=P[2];    t1[3]=P[3]; //P=APA+Q
        P[0] = t1[0] + t1[2]*dt+Q[0];    P[1]=t1[0]*dt+t1[2]*dt*dt+t1[1]+t1[3]*dt;
        P[2] = t1[2];    P[3] = t1[2] + t1[3] +Q[1];
        t1[0]=P[0] + R[0];    t1[1]=P[1]; t1[2]=P[2];    t1[3]=P[3] + R[1];    inv= t1[0]*t1[3]-
t1[1]*t1[2];//K=P/(P+R)
        kA[0] = inv*(P[0]*t1[3]-P[1]*t1[2]);    kA[1] = inv*(P[1]*t1[0]-P[0]*t1[1]);
        kA[2] = inv*(P[2]*t1[3]-P[3]*t1[2]);    kA[3] = inv*(P[3]*t1[0]-P[2]*t1[1]);
        xA0[0] = xA[0] + kA[0] * (y[0] - xA[0]) + kA[1] * (y[1] - xA[1]);
        xA0[1] = xA[1] + kA[2] * (y[0] - xA[0]) + kA[3] * (y[1] - xA[1]);//X=X+K(Z-X)
        kalm2[0] = xA0[0];    kalm2[1] = xA0[1];
        t1[0]=P[0];    t1[1]=P[1];    t1[2]=P[2];    t1[3]=P[3];//P=(I-K).P
        P[0]=(1-kA[0])*t1[0]+kA[1]*t1[2];    P[1]=(1-kA[0])*t1[1]+kA[1]*t1[3];
        P[2] = kA[2]*t1[0]+(1-kA[3])*t1[2];    P[3] = kA[2]*t1[1]+(1-kA[3])*t1[3];
        return;}

void kalman2(float *y, float dt){
        static float Pax[4] = {1,0,0,1}, xA[2], xA0[2]; float R[2] = {0.004,0.02}, Q[2]{0.03,0.09},
kalmax[2];
        static float Pay[4] = {1,0,0,1}, xA[2], xA0[2]; float R[2] = {0.004,0.02}, Q[2]{0.03,0.09},
kalmay[2];
        static float Paz[4] = {1,0,0,1}, xA[2], xA0[2]; float R[2] = {0.004,0.02}, Q[2]{0.03,0.09},
kalmaz[2];

        kalmax[0]=y[0]; kalmax[1]=y[1];    kalman(kalmax, dt, xA, xA0, R, Q, Pax);
        kalmay[0]=y[2]; kalmay[1]=y[3];    kalman(kalmay, dt, xA, xA0, R, Q, Pay);
        kalmaz[0]=y[4]; kalmaz[1]=y[5];    kalman(kalmaz, dt, xA, xA0, R, Q, Paz);
        y[0]=kalmax[0]; y[1]=kalmax[1]; y[2]=kalmay[0]; y[3]=kalmay[1];
        y[4]=kalmaz[0];y[5]=kalmaz[1];
        return;}
```

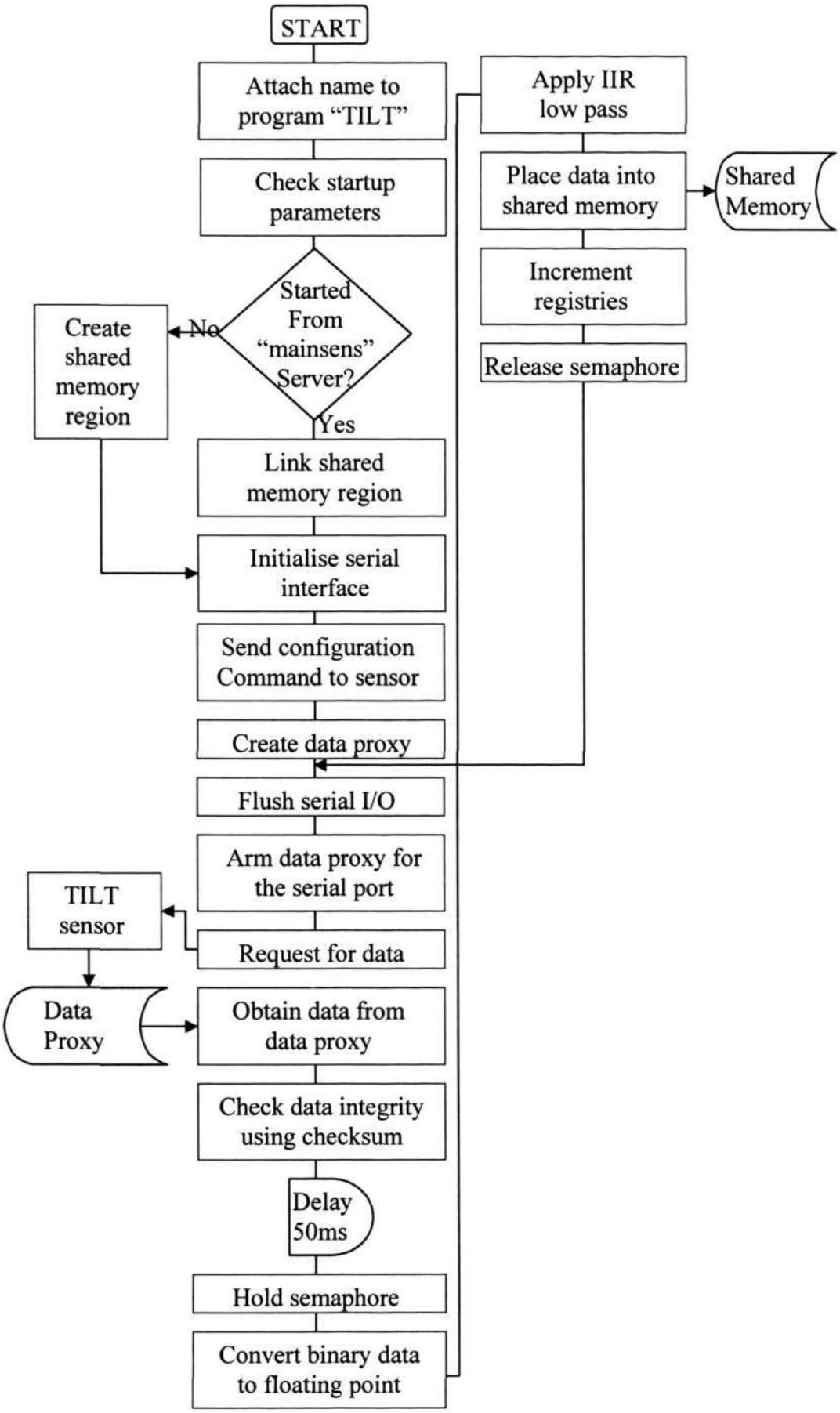


A2) Tritech Altimeter Program

```
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>

//declare global variables
#define   RS232_SPEED2 9600
float     *trit;
unsigned short int *sptr;

//******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag
//data structure 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn
//******************
void shm_make(char mainin){
        int shm_fd1;
        if((shm_fd1=shm_open("SHAR",O_RDWR,0777)) == -1){
                perror("shm_open failed"); exit(EXIT_FAILURE);}
        if(mainin==0){
                shm_unlink("SHAR");
                if((shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777)) == -1){
                        perror("shm_open failed"); exit(EXIT_FAILURE);}
                if((ltrunc(shm_fd1,32768,SEEK_SET))==-1){
                        perror("ltrunc failed"); exit(EXIT_FAILURE);}}
        if((sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0))==(unsig
ned short int*)-1){
                perror("mmap1 failed");  exit(EXIT_FAILURE);}
        if((trit=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,8192))==(flo
at*)-1){
                perror("mmap3 failed");  exit(EXIT_FAILURE);}
        sem_init(&sptr[2000],1,1);
        sptr[23] = 500;
        return;}

// initialise the serial port. It checks the availability of the serial port
// modify the protocol to the one used by the sensor
int ser_init(void){
```

```
        unsigned short int RS232_fd2;
        unsigned RS232_oldmode2;
        struct termios RS232_mode2;
        if((RS232_fd2=open("//2/dev/ser6",O_RDWR))==-1){
                perror("Error:/dev/ser6 failed to open"); exit(1);}
        if((RS232_oldmode2=dev_mode(RS232_fd2,0,_DEV_MODES))==-1) {
                perror("Error:Failed to set raw mode"); exit(1);}
        tcgetattr(RS232_fd2,&RS232_mode2);
        cfsetispeed(&RS232_mode2,RS232_SPEED2);
        cfsetospeed(&RS232_mode2,RS232_SPEED2);
        RS232_mode2.c_cc[16] = 15;
        RS232_mode2.c_cc[17] = 1;
        tcsetattr(RS232_fd2,TCSADRAIN,&RS232_mode2);
        return(RS232_fd2);}

//if program is not started from the server, it ends all the other program
void slayall(void){
        sptr[23] = 0;
        slayed=qnx_name_locate(0,"SENSPOD",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"DMUOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"FALMOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TRITOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"KVHOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TILTOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"VELDOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"MAINSENS",0,NULL);
        kill(slayed,SIGKILL);
        qnx_name_attach(0, "TRITOK");
        return;}

//main program
//checks how the program is started via the argc, argv
//calls the serial initialisation and attach a name, proxy to the program
//enter into a loop; send a read command from the device and release the program
//once proxy is actvated, it regains control and check the intregety of the data
//converts the data to the float format and calls the low pass filter
//place the data into the shared memory and end loop
void main(int argc, char *argv[]){
        unsigned state;
        char data[15], mainin=0;
        pid_t RS232_proxy2, slayed;
        unsigned short int RS232_fd2, ii;
        FILE *fp1;
        if(argc > 1) mainin = atoi(argv[1]);
        shm_make(mainin);
        if(mainin==0){
                slayall();
                fp1 = fopen("//4/home/tritok.log", "w");}
        RS232_fd2 = ser_init();
        RS232_proxy2 = qnx_proxy_attach(0, 0, 0, (getprio(0)+1));
        dev_state(RS232_fd2,0,_DEV_EVENT_INPUT);
```

```
for(;;){
        tcflush(RS232_fd2,TCIOFLUSH);
        dev_arm(RS232_fd2,RS232_proxy2,_DEV_EVENT_INPUT);
        Receive(RS232_proxy2,0,0);
        state = dev_state(RS232_fd2,0,_DEV_EVENT_INPUT);
        if(state&_DEV_EVENT_INPUT){
                read(RS232_fd2, &data, 15);
                for(ii=0; data[ii] != 0x0A;ii++);
                sem_wait(&sptr[2000]);
                sptr[21]=sptr[20];
                trit[(sptr[20])] = atof(&data[ii+1]);
                if(mainin==0){
                        cprintf("%3.2f\n",trit[sptr[20]]);
                        gcvt(trit[sptr[20]], 5, data);
                        fprintf( fp1, "%s \n", data);}
                if(++sptr[20]>=1024) sptr[20]=0;
                ++sptr[22];
                sem_post(&sptr[2000]);
                delay(sptr[23]);}}
return;}
```

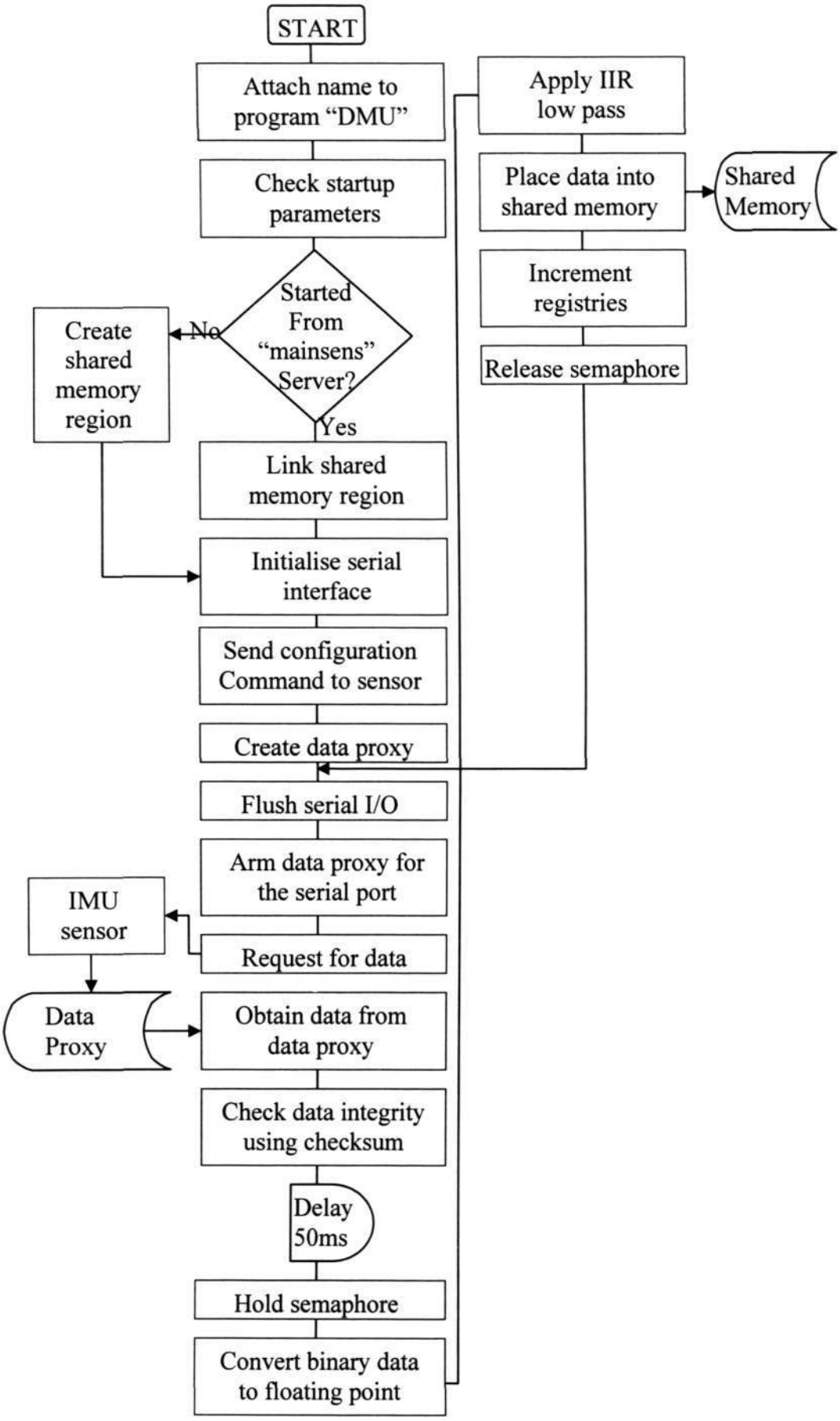


A3) Tilt sensor program

```
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>

#define   RS232_SPEED1 9600

//declare global variables
float     *tlt;
unsigned short int *sptr;
unsigned short aa=0, bb=0, cc=0, dd=0;
//*******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag
//data structure 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn 6x:tilt 7x:veldoppler
//*******************
//map the shared memory area created by the main program
//it checks if the memory object is available and map it. If not it creates a new one
//also maps the semaphore protocol
void shm_make(char mainin){
        int shm_fd1;
        if((shm_fd1=shm_open("SHAR",O_RDWR,0777)) == -1){
                perror("shm_open failed"); exit(EXIT_FAILURE);}
        if(mainin==0){
                shm_unlink("SHAR");
                if((shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777)) == -1){
                        perror("shm_open failed"); exit(EXIT_FAILURE);}
                if((ltrunc(shm_fd1,32768,SEEK_SET))==-1){
                        perror("ltrunc failed"); exit(EXIT_FAILURE);}}
        if((sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0))==(unsig
ned short int*)-1){
                perror("mmap1 failed");  exit(EXIT_FAILURE);}

        if((tlt=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,24576))==(flo
at*)-1){
                perror("mmap6 failed");  exit(EXIT_FAILURE);}
        sem_init(&sptr[2000],1,1);
        sptr[63] = 50;
        return;}
```

```
// initialise the serial port. It checks the availability of the serial port
// modify the protocol to the one used by the sensor
int ser_init(void){
        short int RS232_fd1;
        unsigned RS232_oldmode1;
        struct termios RS232_mode1;
        if((RS232_fd1=open("//2/dev/ser1",O_RDWR))==-1){
                perror("Error:/dev/ser1 failed to open"); exit(1);}
        if((RS232_oldmode1=dev_mode(RS232_fd1,0,_DEV_MODES))==-1) {
                perror("Error:Failed to set raw mode"); exit(1);}
        tcgetattr(RS232_fd1,&RS232_mode1);
        cfsetispeed(&RS232_mode1,RS232_SPEED1);
        cfsetospeed(&RS232_mode1,RS232_SPEED1);
        RS232_mode1.c_cc[16] = 6;
        RS232_mode1.c_cc[17] = 1;
        tcsetattr(RS232_fd1,TCSADRAIN,&RS232_mode1);
        return(RS232_fd1);}

//sensor low pass filter
void IIR(char ofset){
        tlt[sptr[60]+512] = 0.0214*tlt[sptr[60]+ofset] + 0.0442*tlt[aa+ofset] + 0.0590*tlt[bb+ofset]
+ 0.0442*tlt[cc++ofset] + 0.0214*tlt[dd+ofset]/ (1 + 2.2*tlt[aa+512+ofset] - 2.48*tlt[bb+512+ofset] +
1.46*tlt[cc+512+ofset] - 0.389*tlt[dd+512+ofset]);
        return;}

//if program is not started from the server, it ends all the other program
void slayall(void){
        sptr[63] = 0;
        slayed=qnx_name_locate(0,"SENSPOD",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"DMUOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"FALMOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TRITOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"KVHOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TILTOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"VELDOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"MAINSENS",0,NULL);
        kill(slayed,SIGKILL);
        qnx_name_attach(0, "TILTOK");
        return;}

//main program
//checks how the program is started via the argc, argv
//calls the serial initialisation and attach a name, proxy to the program
//enter into a loop; send a read command from the device and release the program
//once proxy is actvated, it regains control and check the intregety of the data
//converts the data to the float format and calls the low pass filter
//place the data into the shared memory and end loop
void main(int argc, char *argv[]){
        unsigned state;
        pid_t RS232_proxy1, slayed;
```

```
short int RS232_fd1, temp;
char TILT[6], mainin=0, csum;
FILE *fp1;
if(argc > 1) mainin = atoi(argv[1]);
shm_make(mainin);
if(mainin==0){
        slayall();
        fp1 = fopen("//4/home/kvhok.log", "w");}
qnx_name_attach(0, "TILTOK");
RS232_fd1 = ser_init();
RS232_proxy1 = qnx_proxy_attach(0, 0, 0, (getprio(0)+1));
dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
for(;;){
        tcflush(RS232_fd1,TCIOFLUSH);
        dev_arm(RS232_fd1,RS232_proxy1,_DEV_EVENT_INPUT);
        write(RS232_fd1, "G", 1);
        Receive(RS232_proxy1,0,0);
        state = dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
        if(state&_DEV_EVENT_INPUT){
                read(RS232_fd1, TILT, 6);
                if(((TILT[1] + TILT[2] + TILT[3] + TILT[4]) & 255) == TILT[5]){
                        sem_wait(&sptr[2000]);
                        sptr[61]=sptr[60];
                        dd=cc; cc=bb; bb=aa; aa=sptr[61];
                        if((temp = (TILT[3] << 8) | TILT[4])>32767) temp = (temp-
65535);
                        tlt[(sptr[60])]  = (temp * 90 / 32767.0) + 1.27;
                        IIR(0);
                        if((temp = (TILT[1] << 8) | TILT[2])>32767) temp = (temp-
65535);
                        tlt[(++sptr[60])]= -temp * 90 / 32767.0 + 0.0320;
                        IIR(1);
                        if(|tlt[(sptr[60])]| > 70 || |tlt[(sptr[61])]| > 70); //do overload routine
here;
                        if(mainin==0){
                                cprintf("%f %f\n", tlt[(sptr[61])], tlt[(sptr[60])]);
                                gcvt(tlt[sptr[61]], 4, TILT);
                                fprintf( fp1, "%s ", TILT);
                                gcvt(tlt[sptr[60]], 4, TILT);
                                fprintf( fp1, "%s \n", TILT);}
                        if(++sptr[60]>=1024) sptr[60]=0;
                        ++sptr[62];
                        sem_post(&sptr[2000]);
                        delay(sptr[63]);}}}
return;}
```



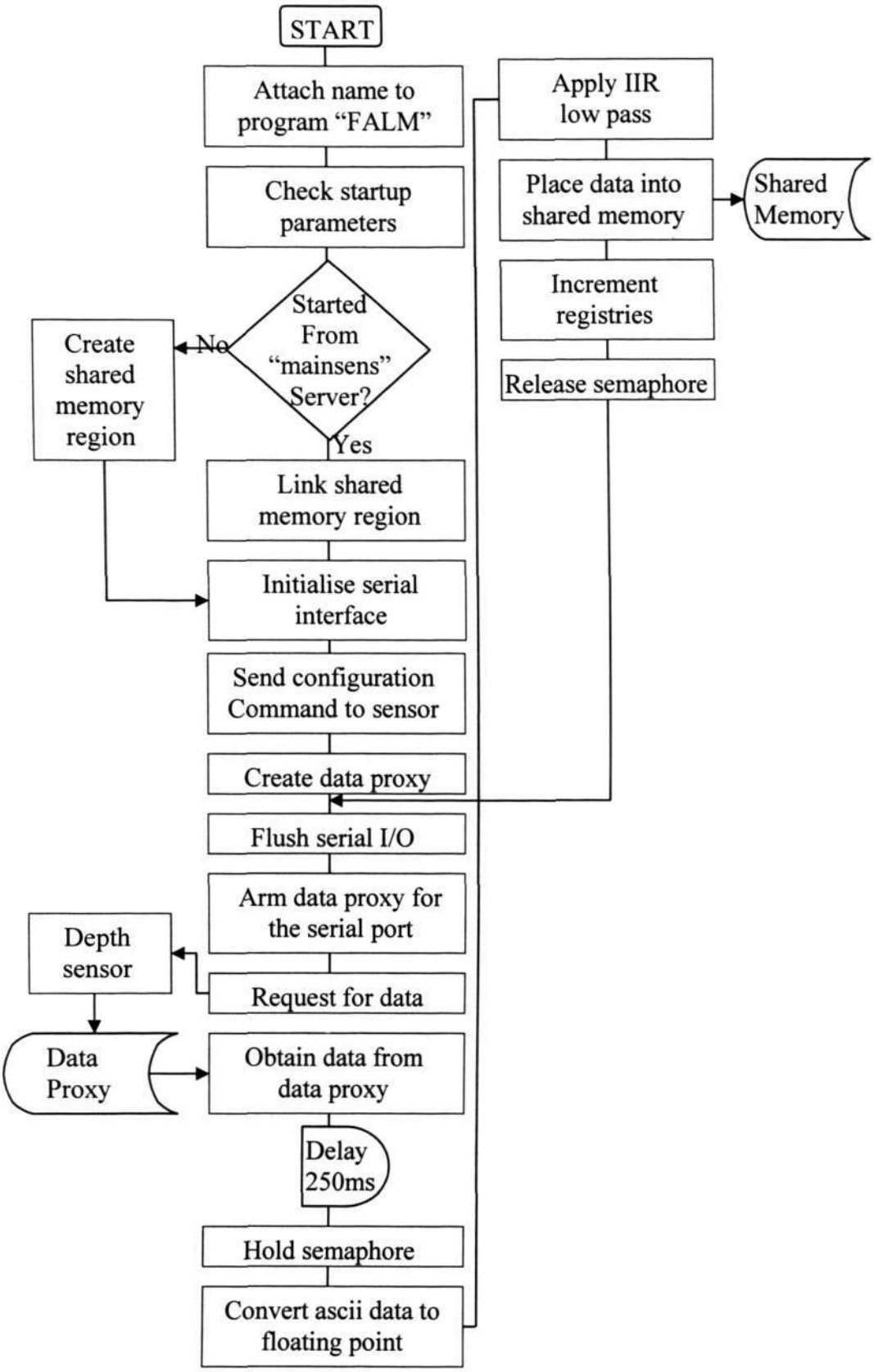
A4) IMU program

```
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>
#include <sys/vc.h>
#include <sys/name.h>


#define RS232_SPEED1  38400
#define BILLION  1000000000L
#define deg2rad 0.0174533

float *dmu;
unsigned short int *sptr;
unsigned short aa=0, bb=0, cc=0, dd=0;

//******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag x5: Baud rate x6: x7:
x8: x9:
//data structure 0x: Reserved 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn 6x: 7x: 8x: 9x:
//******************
//map the shared memory area created by the main program
//it checks if the memory object is available and map it. If not it creates a new one
//also maps the semaphore protocol
void shm_make(char mainin){
        int shm_fd1;
        if((shm_fd1=shm_open("SHAR",O_RDWR,0777)) == -1){
                perror("shm_open failed"); exit(EXIT_FAILURE);}
        if(mainin==0){
                shm_unlink("SHAR");
                if((shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777)) == -1){
                        perror("shm_open failed"); exit(EXIT_FAILURE);}
                if((ltrunc(shm_fd1,32768,SEEK_SET))==-1){
                        perror("ltrunc failed"); exit(EXIT_FAILURE);}}
        if((sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0))==(unsig
ned short int*)-1){
                perror("mmap1 failed");  exit(EXIT_FAILURE);}
```

```
        if((dmu=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,4096))==(fl
oat*)-1){
                perror("mmap2 failed");  exit(EXIT_FAILURE);}
        sem_init(&sptr[2000],1,1);
        sptr[13] = 50;
        return;}

//sensor low pass filter
void IIR(char ofset){
        dmu[sptr[10]+512] = 0.0214*dmu[sptr[10]+ofset] + 0.0442*dmu[aa+ofset] +
0.0590*dmu[bb+ofset] + 0.0442*dmu[cc++ofset] + 0.0214*dmu[dd+ofset]/ (1 +
2.2*dmu[aa+512+ofset] - 2.48*dmu[bb+512+ofset] + 1.46*dmu[cc+512+ofset] -
0.389*dmu[dd+512+ofset]);
        return;}

//if program is not started from the server, it ends all the other program
void slayall(void){
        slayed=qnx_name_locate(0,"SENSPOD",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"DMUOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"FALMOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TRITOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"KVHOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TILTOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"VELDOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"MAINSENS",0,NULL);
        kill(slayed,SIGKILL);
        qnx_name_attach(0, "DMUOK");
        sptr[13] = 0;
        return;}

// initialise the serial port. It checks the availability of the serial port
// modify the protocol to the one used by the sensor
int ser_init(void){
        unsigned short int RS232_fd1;
        unsigned RS232_oldmode1;
        struct termios RS232_mode1;
        if((RS232_fd1=open("//3/dev/ser1",O_RDWR))==-1){
                perror("Error:/dev/ser1 failed to open"); exit(1);}
        if((RS232_oldmode1=dev_mode(RS232_fd1,0,_DEV_MODES))==-1) {
                perror("Error:Failed to set raw mode"); exit(1);}
        tcgetattr(RS232_fd1,&RS232_mode1);
        cfsetispeed(&RS232_mode1,RS232_SPEED1);
        cfsetospeed(&RS232_mode1,RS232_SPEED1);
        RS232_mode1.c_cc[16] = 18;
        RS232_mode1.c_cc[17] = 1;
        tcsetattr(RS232_fd1,TCSADRAIN,&RS232_mode1);
        return(RS232_fd1);}

//main program
//checks how the program is started via the argc, argv
//calls the serial initialisation and attach a name, proxy to the program
```

```
//enter into a loop; send a read command from the device and release the program
//once proxy is actvated, it regains control and check the intregety of the data
//converts the data to the float format and calls the low pass filter
//place the data into the shared memory and end loop
void main(int argc, char *argv[]){
        short numerator;
        unsigned int chk;
        unsigned state;
        pid_t RS232_proxy1;
        unsigned short int RS232_fd1, ii;
        char dmud[18];
        if(argc > 1) mainin = atoi(argv[1]);
        shm_make(mainin);
        if(mainin==0){
                slayall();
                fp1 = fopen("//4/home/dmuok.log", "w");}
        qnx_name_attach(0, "DMUOK");
        RS232_fd1 = ser_init();
        tcflush(RS232_fd1,TCIOFLUSH);
        write(RS232_fd1,"RcP",3);
        read(RS232_fd1, &dmud, 2);
        RS232_proxy1 = qnx_proxy_attach(0, 0, 0, (getprio(0)+1));
        dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
        delay(300);
        for(;;){
                tcflush(RS232_fd1,TCIOFLUSH);
                dev_arm(RS232_fd1,RS232_proxy1,_DEV_EVENT_INPUT);
                delay(sptr[13]);
                write(RS232_fd1,"G",1);
                Receive(RS232_proxy1,0,0);
                state = dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
                if(state&_DEV_EVENT_INPUT){
                        chk = 0;
                        read(RS232_fd1, dmud, 18);
                        for(ii=1;ii<17;ii++) chk = chk + dmud[ii];
                        if(chk%256==dmud[17]){
                                sem_wait(&sptr[2000]);
                                dd=cc; cc=bb; bb=aa; aa=sptr[11];
                                sptr[11]=sptr[10];
                                numerator  = (dmud[1]<<8) |(dmud[2]);
                                dmu[(sptr[10])] = numerator*(0.011901855) - 0.012959;
//roll rate 1.5*260/2^15
                                IIR(0);
                                numerator = (dmud[3]<<8) |(dmud[4]);
                                dmu[(++sptr[10])] = numerator*(0.011901855) + 0.032826;
//pitch rate 1.5*260/2^15
                                IIR(1);
                                numerator = (dmud[5]<<8) |(dmud[6]);
                                dmu[(++sptr[10])] = numerator*(0.011901855) - 0.026896;
//yaw rate 1.5*260/2^15
                                IIR(2);
                                numerator = (dmud[7]<<8) |(dmud[8]);
                                dmu[(++sptr[10])] = numerator*(0.0000915527) + 0.0103;
//ax 2*1.52^15;
                                IIR(3);
                                numerator = (dmud[9]<<8) |(dmud[10]);
                                dmu[(++sptr[10])] = numerator*(0.0000915527) - 0.00281;
//ay 2*1.5/2^15
```

```
                                IIR(4);
                                numerator = (dmud[11]<<8) |(dmud[12]);
                                dmu[(++sptr[10])] = numerator*(0.0000915527) + 0.00304;
//az 2*1.5/2^15
                                IIR(5);
                                if(mainin==0){
                                        cprintf("%f %f %f %f %f %f\n",dmu[(++sptr[15])],
dmu[(sptr[14])], dmu[(sptr[13])], dmu[(sptr[12])], dmu[sptr[11])], dmu[(sptr[10])]);
                                        gcvt(tlt[sptr[15]], 4, TILT); fprintf( fp1, "%s ", TILT);
                                        gcvt(tlt[sptr[14]], 4, TILT); fprintf( fp1, "%s ", TILT);
                                        gcvt(tlt[sptr[13]], 4, TILT); fprintf( fp1, "%s ", TILT);
                                        gcvt(tlt[sptr[12]], 4, TILT); fprintf( fp1, "%s ", TILT);
                                        gcvt(tlt[sptr[11]], 4, TILT); fprintf( fp1, "%s ", TILT);
                                        gcvt(tlt[sptr[10]], 4, TILT); fprintf( fp1, "%s \n", TILT);}

                                if(++sptr[10]>=510) sptr[10]=0;
                                sem_post(&sptr[2000]);}}}
        return;}
```

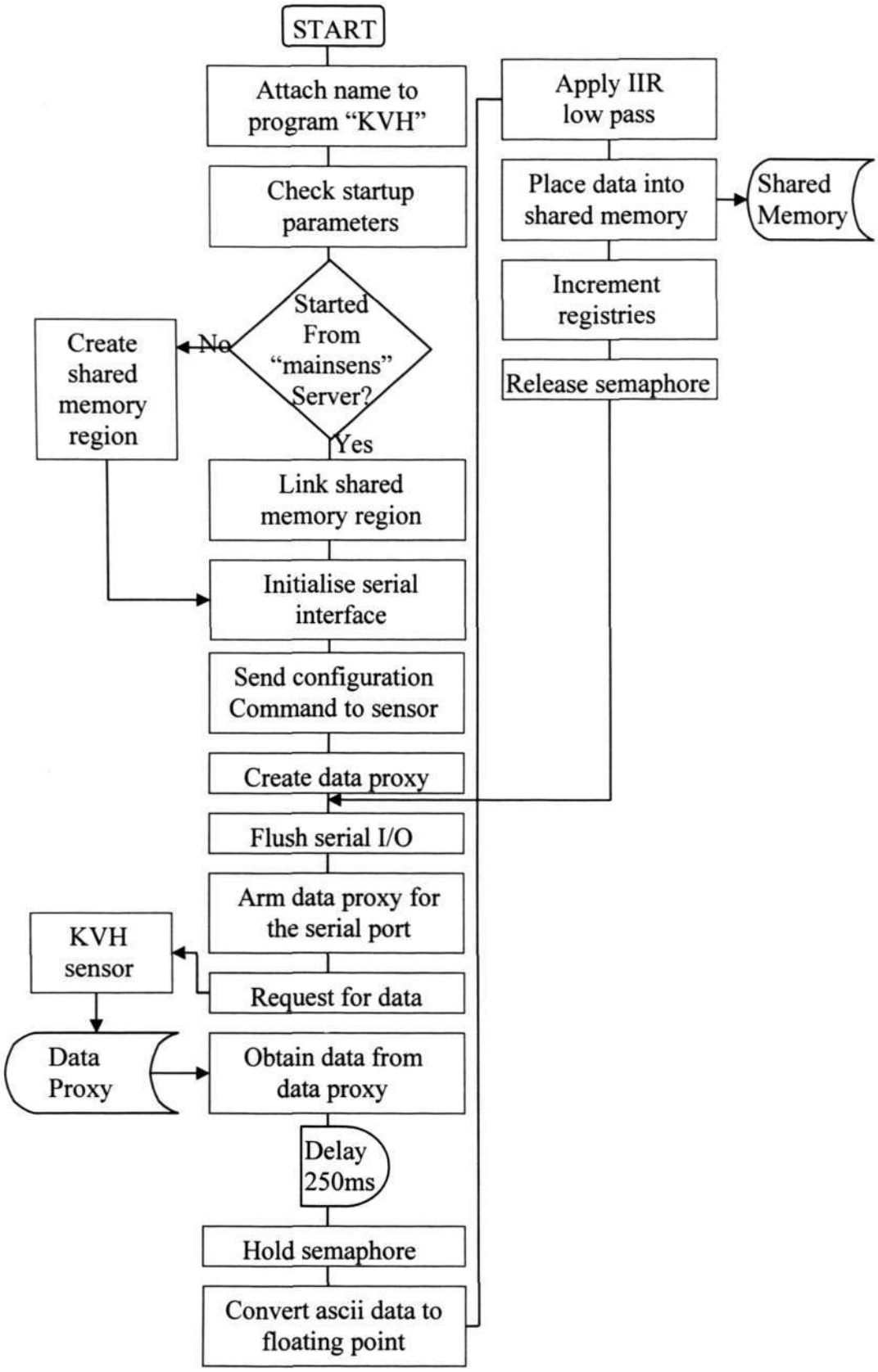


A5) Depth sensor program

```
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>

#define   RS232_SPEED1 9600

float      *falm;
unsigned short int *sptr;
unsigned short aa=0, bb=0;

//******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag
//data structure 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn
//******************
//map the shared memory area created by the main program
//it checks if the memory object is available and map it. If not it creates a new one
//also maps the semaphore protocol
void shm_make(char mainin){
        int shm_fd1;
        if((shm_fd1=shm_open("SHAR",O_RDWR,0777)) == -1){
                perror("shm_open failed"); exit(EXIT_FAILURE);}
        if(mainin==0){
                shm_unlink("SHAR");
                if((shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777)) == -1){
                        perror("shm_open failed"); exit(EXIT_FAILURE);}
                if((ltrunc(shm_fd1,32768,SEEK_SET))==-1){
                        perror("ltrunc failed"); exit(EXIT_FAILURE);}}
        if((sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0))==(unsig
ned short int*)-1){
                perror("mmap1 failed");  exit(EXIT_FAILURE);}
        if((falm=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,12288))==(
float*)-1){
                perror("mmap4 failed");  exit(EXIT_FAILURE);}
        sem_init(&sptr[2000],1,1);
        sptr[33] = 333;
        return;}
```

```
//sensor low pass filter
void IIR(void){
        falm[sptr[30]+512] = 0.00624*falm[sptr[30]] + 0.0125*falm[aa] + 0.00624*falm[bb]/ (1 -
1.176*falm[aa+512] + 0.789*falm[bb+512];
        return;}

// initialise the serial port. It checks the availability of the serial port
// modify the protocol to the one used by the sensor
int ser_init(void){
        unsigned short int RS232_fd1;
        unsigned RS232_oldmode1;
        struct termios RS232_mode1;
        char outdata[5], buffer1[28];
        if((RS232_fd1=open("//2/dev/ser5",O_RDWR))==-1){
                perror("Error:/dev/ser5 failed to open"); exit(1);}
        if((RS232_oldmode1=dev_mode(RS232_fd1,0,_DEV_MODES))==-1) {
                perror("Error:Failed to set raw mode"); exit(1);}
        tcgetattr(RS232_fd1,&RS232_mode1);
        cfsetispeed(&RS232_mode1,RS232_SPEED1);
        cfsetospeed(&RS232_mode1,RS232_SPEED1);
        RS232_mode1.c_cc[16] = 35;
        RS232_mode1.c_cc[17] = 1;
        tcsetattr(RS232_fd1,TCSADRAIN,&RS232_mode1);
        delay(5);
        strcpy(outdata,"***O");                                  //set to open mode to change
the update rate
        outdata[4] = 0x0D;
        write(RS232_fd1, outdata, 5);
        dev_read(RS232_fd1, &buffer1, 28, 28, 1, 1, 0, 0);
        delay(5);
        strcpy(outdata,"ST=4");                                  //update the update rate to
5.5hz
        outdata[4] = 0x0D;
        write(RS232_fd1, outdata, 5);
        dev_read(RS232_fd1, &buffer1, 28, 28, 1, 1, 0, 0);
        delay(5);
        strcpy(outdata, "***R");                                 //go back to the reading mode
        outdata[4] = 0x0D;
        write(RS232_fd1, outdata, 5);
        dev_read(RS232_fd1, &buffer1, 28, 28, 1, 1, 0, 0);
        return(RS232_fd1);}

//if program is not started from the server, it ends all the other program
void slayall(void){
        sptr[33] = 0;
        slayed=qnx_name_locate(0,"SENSPOD",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"DMUOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"FALMOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TRITOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"KVHOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TILTOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"VELDOK",0,NULL);
```

```
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"MAINSENS",0,NULL);
        kill(slayed,SIGKILL);
        qnx_name_attach(0, "FALMOK");
        return;}

//main program
//checks how the program is started via the argc, argv
//calls the serial initialisation and attach a name, proxy to the program
//enter into a loop; send a read command from the device and release the program
//once proxy is actvated, it regains control and check the intregety of the data
//converts the data to the float format and calls the low pass filter
//place the data into the shared memory and end loop
void main(int argc, char *argv[]){
        unsigned state;
        pid_t RS232_proxy1, slayed;
        unsigned short int RS232_fd1;
        char falmth[28], mainin=0;
        FILE *fp1;
        if(argc > 1) mainin = atoi(argv[1]);
        shm_make(mainin);
        if(mainin==0){
                slayall();
                fp1 = fopen("//4/home/falmok.log", "w");}
        RS232_fd1 = ser_init();
        RS232_proxy1 = qnx_proxy_attach(0, 0, 0, (getprio(0)+1));
        dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
        for(;;){
                tcflush(RS232_fd1,TCIOFLUSH);
                dev_arm(RS232_fd1,RS232_proxy1,_DEV_EVENT_INPUT);
                write(RS232_fd1, "\n",1);
                Receive(RS232_proxy1,0,0);
                state = dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
                if(state&_DEV_EVENT_INPUT){
                        read(RS232_fd1, falmth, 28);
                        sem_wait(&sptr[2000]);
                        sptr[31] = sptr[30];
                        falm[(sptr[30])] = (atof(&falmth[20]) * 1.02377) + 0.30;
                        IIR();
                        if(mainin==0){
                                cprintf("%3.2f\n", atof(&falmth[20]));
                                gcvt(depth, 5, falmth);
                                fprintf( fp1, "%s \n", falmth);}
                        if(++sptr[30]>=1024) sptr[30]=0;
                        ++sptr[22];
                        sem_post(&sptr[2000]);
                        delay(sptr[33]);}}
        return;}
```

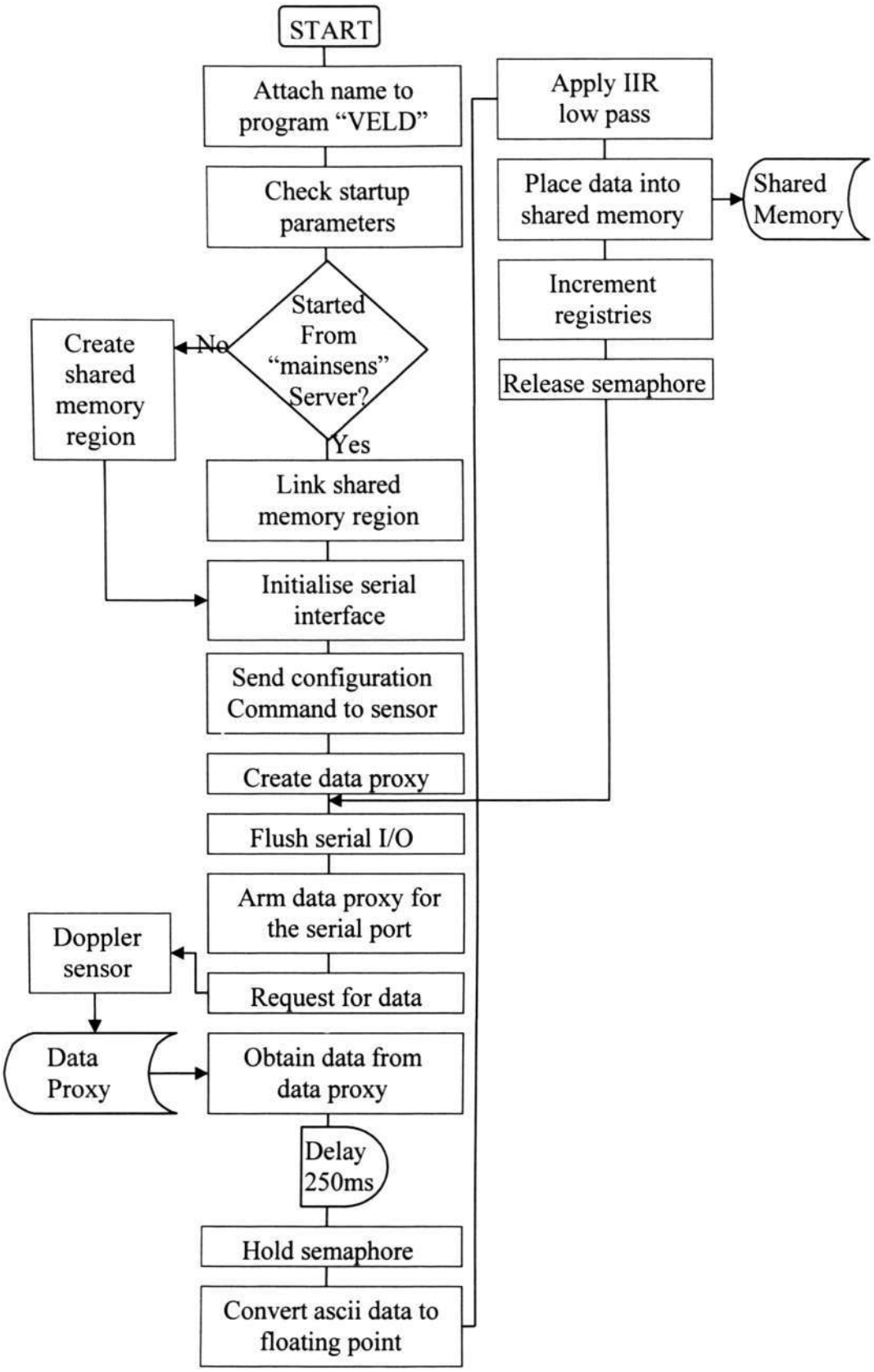


A6) Compass program

```
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>

#define   RS232_SPEED1 9600

//declare global variables
float      *kvh;
unsigned short int *sptr;
unsigned short aa=0, bb=0;

//******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag
//data structure 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn
//******************
//map the shared memory area created by the main program
//it checks if the memory object is available and map it. If not it creates a new one
//also maps the semaphore protocol
void shm_make(char mainin){
          int shm_fd1;
          if((shm_fd1=shm_open("SHAR",O_RDWR,0777)) == -1){
                    perror("shm_open failed"); exit(EXIT_FAILURE);}
          if(mainin==0){
                    shm_unlink("SHAR");
                    if((shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777)) == -1){
                              perror("shm_open failed"); exit(EXIT_FAILURE);}
                    if((ltrunc(shm_fd1,32768,SEEK_SET))==-1){
                              perror("ltrunc failed"); exit(EXIT_FAILURE);}}
          if((sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0))==(unsig
ned short int*)-1){
                    perror("mmap1 failed");  exit(EXIT_FAILURE);}
          if((kvh=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,16384))==(f
loat*)-1){
                    perror("mmap5 failed");  exit(EXIT_FAILURE);}
          sem_init(&sptr[2000],1,1);
          sptr[43] = 1000;
          return;}
```

```
//sensor low pass filter
void IIR(void){
        kvh[sptr[40]+512] = 0.957*kvh[sptr[40]] + 1.91*kvh[aa] + 0.957*kvh[bb]/ (1 +
1.91*kvh[aa+512] + 0.915*kvh[bb+512];
        return;}

// initialise the serial port. It checks the availability of the serial port
// modify the protocol to the one used by the sensor
int ser_init(void){
        unsigned short int RS232_fd1;
        unsigned RS232_oldmode1;
        struct termios RS232_mode1;
        if((RS232_fd1=open("//2/dev/ser3",O_RDWR))==-1){
                perror("Error:/dev/ser3 failed to open"); exit(1);}
        if((RS232_oldmode1=dev_mode(RS232_fd1,0,_DEV_MODES))==-1) {
                perror("Error:Failed to set raw mode"); exit(1);}
        tcgetattr(RS232_fd1,&RS232_mode1);
        cfsetispeed(&RS232_mode1,RS232_SPEED1);
        cfsetospeed(&RS232_mode1,RS232_SPEED1);
        RS232_mode1.c_cc[16] = 14;
        RS232_mode1.c_cc[17] = 1;
        tcsetattr(RS232_fd1,TCSADRAIN,&RS232_mode1);
        return(RS232_fd1);}

//if program is not started from the server, it ends all the other program
void slayall(void){
        sptr[43] = 0;
        slayed=qnx_name_locate(0,"SENSPOD",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"DMUOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"FALMOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TRITOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"KVHOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TILTOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"VELDOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"MAINSENS",0,NULL);
        kill(slayed,SIGKILL);
        qnx_name_attach(0, "KVHOK");
        return;}

//main program
//checks how the program is started via the argc, argv
//calls the serial initialisation and attach a name, proxy to the program
//enter into a loop; send a read command from the device and release the program
//once proxy is actvated, it regains control and check the intregety of the data
//converts the data to the float format and calls the low pass filter
//place the data into the shared memory and end loop
void main(int argc, char *argv[]){
        unsigned state;
        pid_t RS232_proxy1, slayed;
        unsigned short int RS232_fd1;
```

```
char KVHC100[14], mainin=0, kvhout[3];
FILE *fp1;
if(argc > 1) mainin = atoi(argv[1]);
shm_make(mainin);
if(mainin==0){
        slayall();
        fp1 = fopen("//4/home/kvhok.log", "w");}
KVHout="d0"; KVHout[2]=0x0D;
RS232_fd1 = ser_init();
RS232_proxy1 = qnx_proxy_attach(0, 0, 0, (getprio(0)+1));
dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
for(;;){
        tcflush(RS232_fd1,TCIOFLUSH);
        dev_arm(RS232_fd1,RS232_proxy1,_DEV_EVENT_INPUT);
        tcflush(RS232_fd1,TCIOFLUSH);
        write(RS232_fd1, &kvhout, 3);
        Receive(RS232_proxy1,0,0);
        state = dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
        if(state&_DEV_EVENT_INPUT){
                read(RS232_fd1, KVHC100, 14);
                sem_wait(&sptr[2000]);
                sptr[41]=sptr[40];
                kvh[(sptr[40])] = atof(&KVHC100[9]);
                IIR();
                if(kvh[sptr[40]]>799.0) cprintf("overload");
                if(mainin==0){
                        cprintf("%3.2f\n",kvh[sptr[40]]);
                        gcvt(kvh[sptr[40]], 5, KVHC100);
                        fprintf( fp1, "%s \n", KVHC100);}
                if(++sptr[40]>=1024) sptr[40]=0;
                ++sptr[42];
                sem_post(&sptr[2000]);
                delay(sptr[43]);}}
return;}
```



```
#include "mqueue.h"
#include <conio.h>
#include <unistd.h>
#include <termios.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <fcntl.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dev.h>
#include <sys/proxy.h>
#include <sys/kernel.h>
#include <sys/irqinfo.h>
#include <sys/name.h>
#include <sys/dev.h>
#include <sys/uio.h>
#include <sys/sched.h>
#include <sys/vc.h>
#include <sys/name.h>


#define   RS232_SPEED1 57600

float      *vel;
unsigned short int *sptr;
unsigned short aa=0, bb=0;
//******************
//data structure x0: head, x1: tail, x2: no of sets of data, x3:pause, x4: data flag
//data structure 1x: dmu, 2x:trit, 3x:falm, 4x:kvh, 5x:svn 6x:tilt 7x:veldoppler
//******************
//map the shared memory area created by the main program
//it checks if the memory object is available and map it. If not it creates a new one
//also maps the semaphore protocol
void shm_make(char mainin){
        int shm_fd1;
        if((shm_fd1=shm_open("SHAR",O_RDWR,0777)) == -1){
                perror("shm_open failed"); exit(EXIT_FAILURE);}
        if(mainin==0){
                shm_unlink("SHAR");
                if((shm_fd1=shm_open("SHAR",O_RDWR|O_CREAT,0777)) == -1){
                        perror("shm_open failed"); exit(EXIT_FAILURE);}
                if((ltrunc(shm_fd1,32768,SEEK_SET))==-1){
                        perror("ltrunc failed"); exit(EXIT_FAILURE);}}
        if((sptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,0))==(unsig
ned short int*)-1){
                perror("mmap1 failed");  exit(EXIT_FAILURE);}

        if((tlt=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd1,28672))==(flo
at*)-1){
                perror("mmap6 failed");  exit(EXIT_FAILURE);}
        sem_init(&sptr[2000],1,1);
        sptr[63] = 500;
```

```
        return;}

//sensor low pass filter
void IIR(char ofset){
        vel[sptr[70]+512] = 0.802*vel[sptr[70]+ofset] + 1.60*vel[aa+ofset] + 0.802*vel[bb+ofset] /
(1 + 1.56*vel[aa+512+ofset] - 0.644*vel[bb+512+ofset]);
        return;}

// initialise the serial port. It checks the availability of the serial port
// modify the protocol to the one used by the sensor
int ser_init(void){
        short int RS232_fd1;
        unsigned RS232_oldmode1;
        struct termios RS232_mode1;
        if((RS232_fd1=open("//3/dev/ser4",O_RDWR))==-1){
                perror("Error:/dev/ser1 failed to open"); exit(1);}
        if((RS232_oldmode1=dev_mode(RS232_fd1,0,_DEV_MODES))==-1) {
                perror("Error:Failed to set raw mode"); exit(1);}
        tcgetattr(RS232_fd1,&RS232_mode1);
        cfsetispeed(&RS232_mode1,RS232_SPEED1);
        cfsetospeed(&RS232_mode1,RS232_SPEED1);
        RS232_mode1.c_cc[16] = 82;
        RS232_mode1.c_cc[17] = 1;
        tcsetattr(RS232_fd1,TCSADRAIN,&RS232_mode1);
        return(RS232_fd1);}

//if program is not started from the server, it ends all the other program
void slayall(void){
        sptr[63] = 0;
        slayed=qnx_name_locate(0,"SENSPOD",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"DMUOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"FALMOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TRITOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"KVHOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"TILTOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"VELDOK",0,NULL);
        kill(slayed,SIGKILL);
        slayed=qnx_name_locate(0,"MAINSENS",0,NULL);
        kill(slayed,SIGKILL);
        qnx_name_attach(0, "TILTOK");
        return;}

//main program
//checks how the program is started via the argc, argv
//calls the serial initialisation and attach a name, proxy to the program
//enter into a loop; send a read command from the device and release the program
//once proxy is actvated, it regains control and check the intregety of the data
//converts the data to the float format and calls the low pass filter
//place the data into the shared memory and end loop
void main(int argc, char *argv[]){
        unsigned state;
        pid_t RS232_proxy1, slayed;
```

```
short int RS232_fd1;
char mainin=0, veld[82], veldata[6];
FILE *fp1;
if(argc > 1) mainin = atoi(argv[1]);
shm_make(mainin);
if(mainin==0){
        slayall();
        fp1 = fopen("//4/home/kvhok.log", "w");}
shm_make();
qnx_name_attach(0, "VELDOK");
RS232_fd1 = ser_init();
RS232_proxy1 = qnx_proxy_attach(0, 0, 0, (getprio(0)+1));
dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
for(;;){
        tcflush(RS232_fd1,TCIOFLUSH);
        dev_arm(RS232_fd1,RS232_proxy1,_DEV_EVENT_INPUT);
        write(RS232_fd1, "O",1);
        Receive(RS232_proxy1,0,0);
        state = dev_state(RS232_fd1,0,_DEV_EVENT_INPUT);
        if(state&_DEV_EVENT_INPUT){
                read(RS232_fd1, veld, 82);
                sem_wait(&sptr[2000]);
                bb=aa; aa=sptr[71];
                sptr[71]=sptr[70];
            if(veld[80]=='1'){
                vel[sptr[70]]=atof(&veld[51])*0.01;
                IIR(0);
                vel[++sptr[70]]=atof(&veld[60])*0.01;
                IIR(1);
                vel[++sptr[70]]=atof(&veld[69])*0.01;
                IIR(2);
                ++sptr[72];
                if(++sptr[70]>=1023) sptr[70]=0;}
            else{   if(veld[49]=='1'){
                        vel[sptr[70]]=atof(&veld[20])*0.01;
                        IIR(0);
                        vel[++sptr[70]]=atof(&veld[29])*0.01;
                        IIR(1);
                        vel[++sptr[70]]=atof(&veld[38])*0.01;
                        IIR(2);
                        ++sptr[72];
                        if(++sptr[70]>=1023) sptr[70]=0;}}
                if(mainin==0){
                        cprintf("%f %f %f \n", vel[sptr[72]], vel[sptr[71]],
vel[sptr[70]]);
                        gcvt(vel[sptr[72]], 4, veldata); fprintf( fp1, "%s ",
veldata);
                        gcvt(vel[sptr[71]], 4, veldata); fprintf( fp1, "%s ",
veldata);
                        gcvt(vel[sptr[70]], 4, veldata); fprintf( fp1, "%s \n",
veldata);}
            sem_post(&sptr[2000]);
            delay(sptr[73]);}}
    return;}
```

APPENDIX B
SENSOR CALIBRATION AND SECONDARY TESTS

B1 Static state calibration

Before the sensors can be used, calibration is necessary to correct any inherent errors. Using the Kawasaki robotic arm setup as shown in Figure 5.8 of chapter 5, the sensors undergoes various calibration. Mounted on the end effector, the sensors are left, static reading are taken for one hour for five days and averaged out. A spirit level was used to ensure its flatness. Table B1 shows the results. The zero errors are corrected before performing other tests and the variance is used in the kalman filter.

Table B1 Sensors zero error and standard deviation

Sensors	Data	Zero error	Std deviation
IMU	P	0.0130	0.153
	Q	-0.0328	0.139
	R	-0.0269	0.144
	Ax	-0.0103	0.0287
	Ay	0.00281	0.0221
	Az	-0.00304	0.0244
Tilt sensor	Roll	-1.27	0.00904
	Pitch	-.0320	0.00731
Compass	Heading	N.A.	0.0926

B2 Stepped rotation test

The stepped angle test involves holding two of the three angles and changing the remaining angles at a step of one degree. By fixing the two angles, the remaining angle directly translates to the navigational frame of motion. This allows the following test to be conducted:

1 Static Ax, Ay, Az, gravity-based calibration

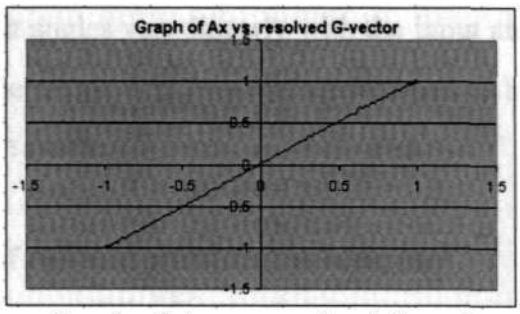
2 Calibration of Tilt sensor

3 Effect of roll and pitch for the compass

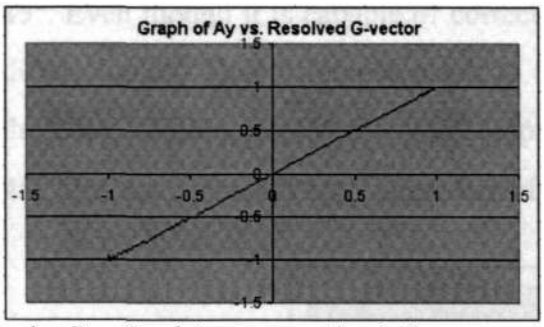
4 Calibration of compass

B2.1 Static accelerometer gravity-based calibration

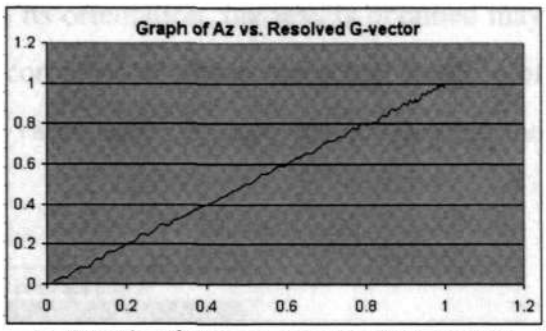
The static accelerometer calibration involves using the natural gravity vector to calibrate the accelerometer. Since the gravity vector always points towards the earth, tilting the accelerometer at various angles with respect to the earth will result in a partially resolved gravity vector. This property is used to vary against a resolved angle to compare the data from the sensor and the expected resolved gravity vector.



a. Graph of Ax vs. resolved G-vector



b. Graph of Ay vs. resolved G-vector



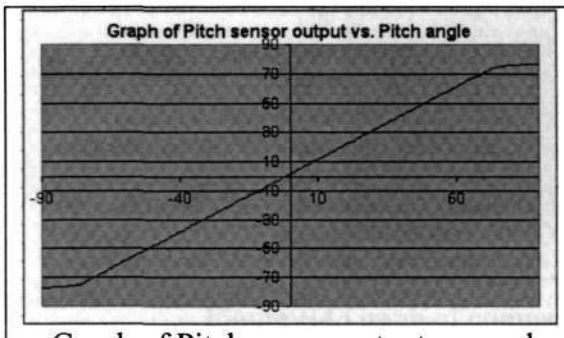
c. Graph of Az vs. resolved G-vector

Figure B1 Calibration of the accelerometers under static condition using the G-vector as a comparison

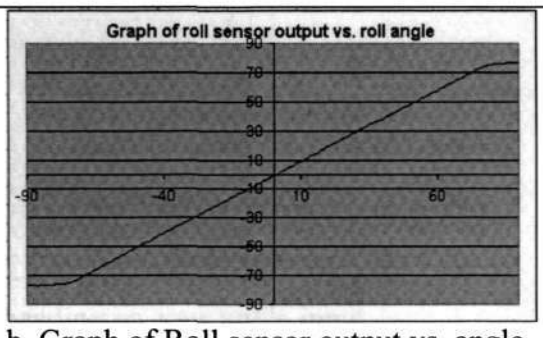
The test conducted varied the Ax and Ay from –1G to +1G while the Az was varied from 0 to +1G since the Az will always be maintained downwards thus will never reach the –G region. From the three graphs, it can be seen that the IMU accelerometers vary linearly when compared under static condition. From the results, no further correction is necessary to compensate any linearity error.

B2.2 Calibration of Tilt sensor

As with the compass, the tilt sensor may not be linear in its output. As such using the stepped change in the angle allows the calibration of the tilt sensor, which would allow any linearity errors to be corrected before it can be used.



a. Graph of Pitch sensor output vs. angle



b. Graph of Roll sensor output vs. angle

Figure B2 Calibration of the tilt sensors output against the tilt angle input

From the results, the tilt angles vary linearly with the input angles between –70° and +70°. Therefore, all operations using the tilt sensor need to be restrained within the operation angles of the sensors.

B2.3 Effect of roll and pitch for the compass

The gimbaled compass is capable of correcting its orientation caused by a tilt of up to 45°. Even though it is capable of correcting its orientation, the results obtained may differ slightly. This difference needs to be corrected to obtain the actual heading of the URV. Within the 45° tilts, the compass value varies by only ±two°. Beyond the 45° tilts, the compass varies unpredictably.

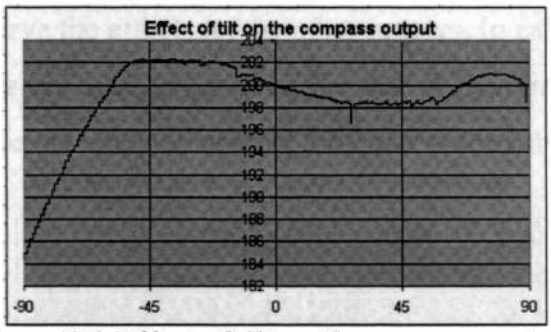


Figure B3 Effect of tilt on the compass output

B2.4 Calibration of compass

The compass reading depends on the earth magnetic field. Unfortunately, the earth magnetic field is not constant throughout the world. As such, the compass may not always display the correct heading. The data output by the compass needs to be calibrated by correcting and linearising the heading before the compass can be used.

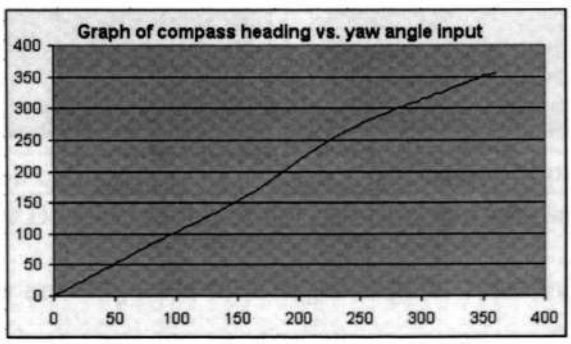


Figure B4 Graph of compass heading vs. yaw angle input

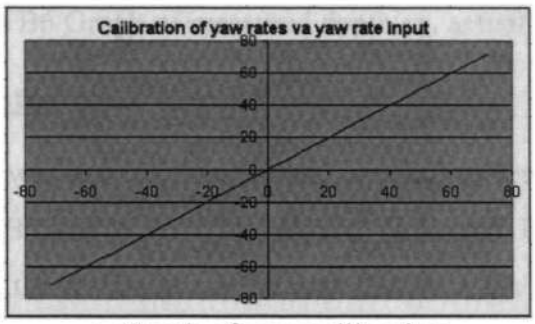
From the results, the error from the compass varies by up to 20°. In order for the compass to output the correct heading, output from the compass needs to be corrected before it can be used. By using curve fitting, the following equation needs to be deducted:

$$Y(x)=-3.01e\text{-}10x^5+2.84e\text{-}7x^4-9.04e\text{-}5x^3+0.0111x^2-0.492x^1+3.42 \quad \text{-- B1}$$

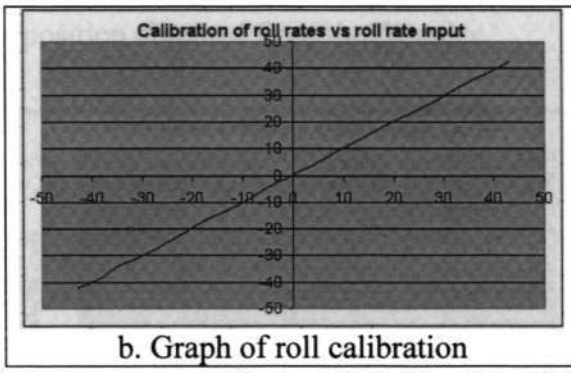
Where Y is the error that needs to be deducted and x is the sensor output.

B3 Normal orientation, constant angular rotation

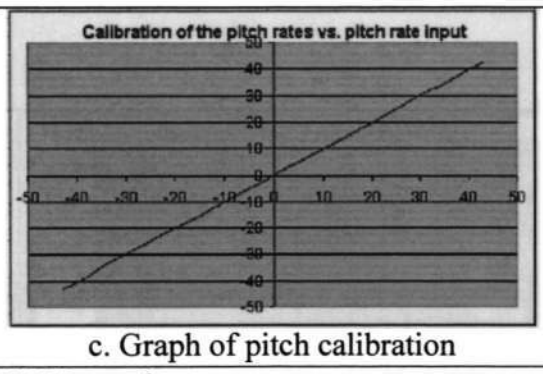
In this test, again two of the three angles are fixed while the third angle is allowed to rotate. While the earlier test increment the angles by steps, this test involves fixing the angular rate of the robotic arms. The angular rate is varied to calibrate the rate gyroscope and to observe the effects on the other sensors. In calibrating the rate gyro, two of the three angles are fixed and the angular rate of the third angle is varied. The output of the gyroscope is compared against the input to the robotic arm.



a. Graph of yaw calibration



b. Graph of roll calibration



c. Graph of pitch calibration

Figure B5 Calibration of the IMU angular rate sensors

Since the yaw angles are subjected to a larger angular rates the maximum yaw rate input are set to 72°/s while the roll and pitch rates are subjected to a maximum rate of 43°/s. From the angular sensors calibration, all the angular rate sensors vary linearly within the calibration limits.

B4 Depth sensor calibration

In this experiment, the depth sensor was lowered from the top at 0.25 meter apart up to a depth of 4.0m. The readings from the sensors are then plotted against the depth. As it can be seen, the depth sensor remained linear over the tested depth. An offset correction of +0.0314 is required correct the reading obtained from the sensor so as to display the correct depth value.

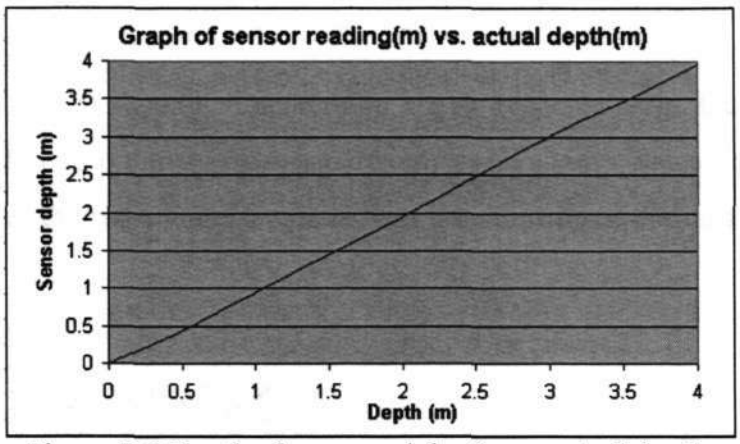
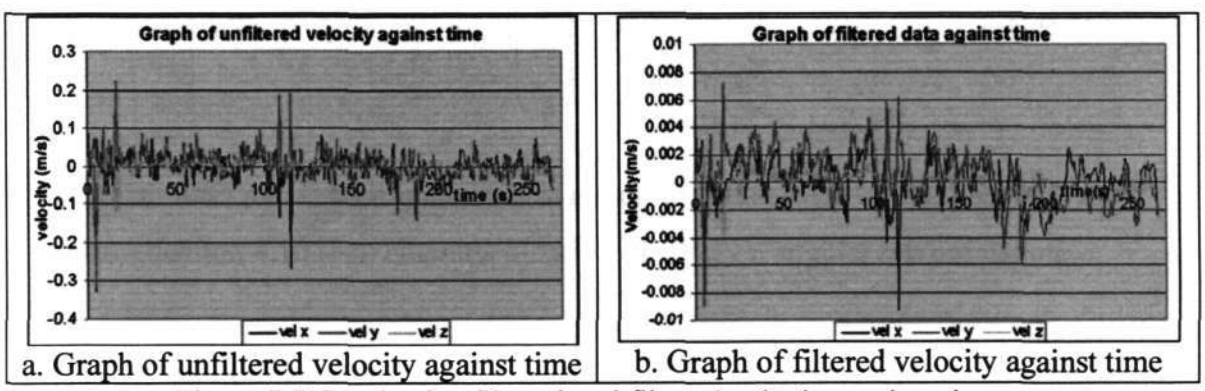


Figure B6 Graph of measured depth vs. actual depth.

B5 Doppler static test

The floating platform was used in this test and was left in the center of the pool for five minutes. Figure B7 shows the filtered and unfiltered Doppler reading under static condition. From the results, it can be seen that with the filter, the performance improve from ±0.2 m/s to ±0.01m/s, thus reducing the error in generating the position obtained from the Doppler.



a. Graph of unfiltered velocity against time

b. Graph of filtered velocity against time

Figure B7 Graph of unfiltered and filtered velocity against time

B6 Quaternion transformation test

Quaternion transformation involves transformation based on the quaternion angular representation. Quaternion transformation requires more equations to execute the transformation but uses less maths function calls such as sin and cosine. This results in the elimination of error states such as when the pitch is at 90° where Euler transformation will fail. Unlike Euler transformation, quaternion transformation includes data normalisation to reduce intermediate processing errors.

Equation 4.10 (a – d) shows the quaternion angles representation. Quaternion terms are represented by four terms whose square terms will always equates to unity. Using equation 4.10(e) the p-q-r angular rate is then transformed to the quaternion angular rate.

$$e_1 = C\frac{\phi}{2}C\frac{\theta}{2}C\frac{\psi}{2} + S\frac{\phi}{2}S\frac{\theta}{2}S\frac{\psi}{2} \quad \text{-- B2(a)}$$

$$e_2 = -S\frac{\phi}{2}C\frac{\theta}{2}C\frac{\psi}{2} - C\frac{\phi}{2}S\frac{\theta}{2}S\frac{\psi}{2} \quad \text{-- B2(b)}$$

$$e_3 = C\frac{\phi}{2}S\frac{\theta}{2}C\frac{\psi}{2} + S\frac{\phi}{2}C\frac{\theta}{2}S\frac{\psi}{2} \quad \text{-- B2(c)}$$

$$e_4 = C\frac{\phi}{2}C\frac{\theta}{2}S\frac{\psi}{2} - S\frac{\phi}{2}S\frac{\theta}{2}C\frac{\psi}{2} \quad \text{-- B2(d)}$$

$$\begin{pmatrix} \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \\ \dot{e}_4 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & p & 0 \end{pmatrix}\begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{pmatrix} \quad \text{-- B2(e)}$$

The quaternion rate is then integrated to obtain the quaternion terms. Based on these angles, equation 4.10 (f-h) converts it back to the R-P-Y angular representation.

$$\theta = S^{-1}[2(e_1e_3 - e_2e_4)] \quad \text{-- B2(f)}$$

$$\phi = T^{-1}\left[\frac{2(e_1e_2 + e_3e_4)}{e_1^2 - e_2^2 - e_3^2 + e_4^2}\right] \quad \text{-- B2(g)}$$

$$\psi = T^{-1}[\frac{2(e_2e_3 + e_1e_4)}{e_1^2 + e_2^2 - e_3^2 - e_4^2}] \qquad \text{-- B2(h)}$$

Since $E^2 = e_1^2 + e_2^2 + e_3^2 + e_4^2$ should be equal to unity, the error obtained from the addition of the quaternion term will be represented by ε. Therefore, each of the terms can be normalised by using the following equation:

$$e'_i = \frac{e_i}{\sqrt{1-\varepsilon}} \qquad \text{for i= 1-4} \qquad \text{-- B2(i)}$$

Timings for gyroscope angular rate and data integration

In converting the body frame angular rate to the angular rate and angle with respect to the local fixed frame, the Euler and the quaternion transformation that was discussed in section 4.4.3 was analysed. Since both transformations require numerical data integration, timings for various integration means as discussed in section 4.3.3 were included. As in the previous experiments, the algorithms were looped for a million cycles. The result shows the Euler transformation requires 20% less time than the quaternion transformation, while the block integration requires three times less processing than trapezoidal integration. To ensure soft-real time compatibility, the Euler transformation and the block integration are more suitable for the system. Furthermore, lower processing time will allow for faster loops and thus better performance.

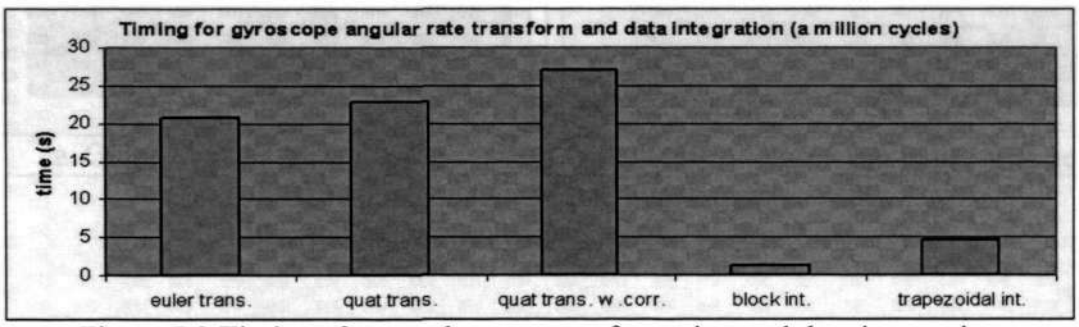
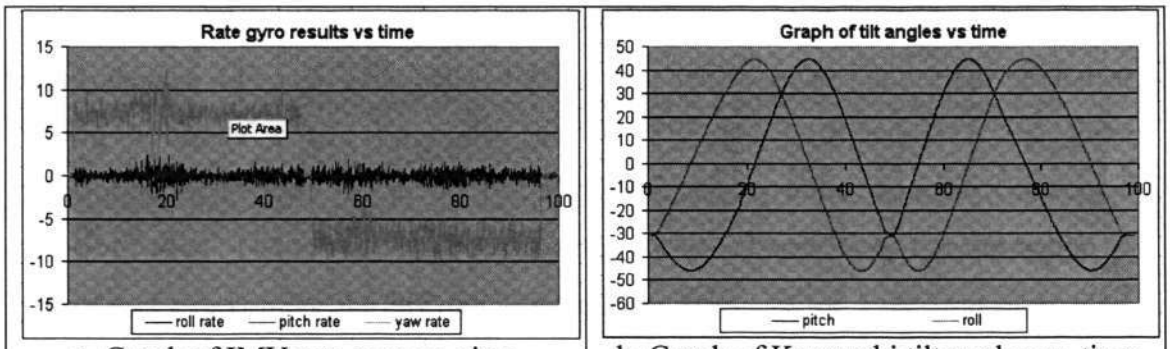


Figure B8 Timings for angular rate transformation and data integration

Angular rate transformation test (Euler vs. quaternion)

In section 4.4.3, the algorithm to convert the body attached angular rate to the fixed frame angular rate was discussed. For the URV, the Euler transformation should be

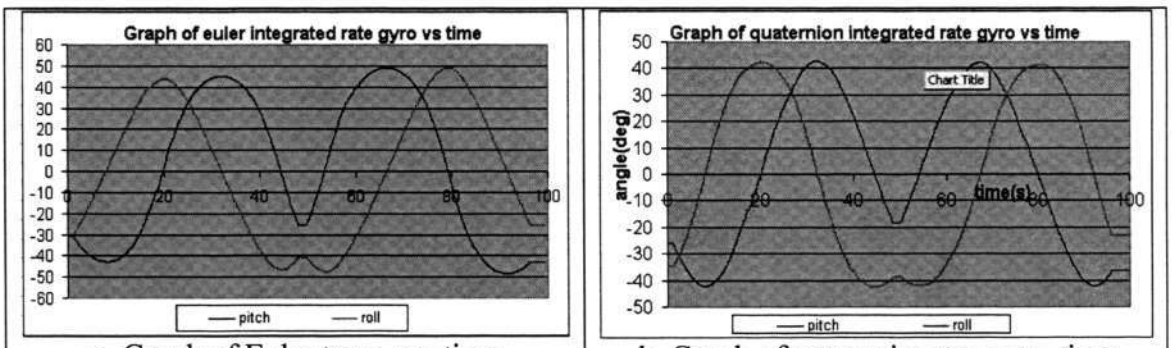
sufficient since it is extremely unlikely that the URV will pitch to 90°, which results in a singularity. This experiment was conducted to compare the output from the Euler and quaternion to convert the angular rate from the body frame to the fixed frame. The robotic arm was used to rotate the sensors clockwise from 0° to 360° and counter clockwise back to 0° about a tilting angle of 45° above the horizon. Figure 5.8 of chapter 5 illustrates this experiment. Figure B9 shows the roll and pitch motion of the robotic arm and the results obtained from the IMU, while figure B10 and B11 shows the transformed results and its error.



a. Graph of IMU rate gyro vs. time

b. Graph of Kawasaki tilt angles vs. time

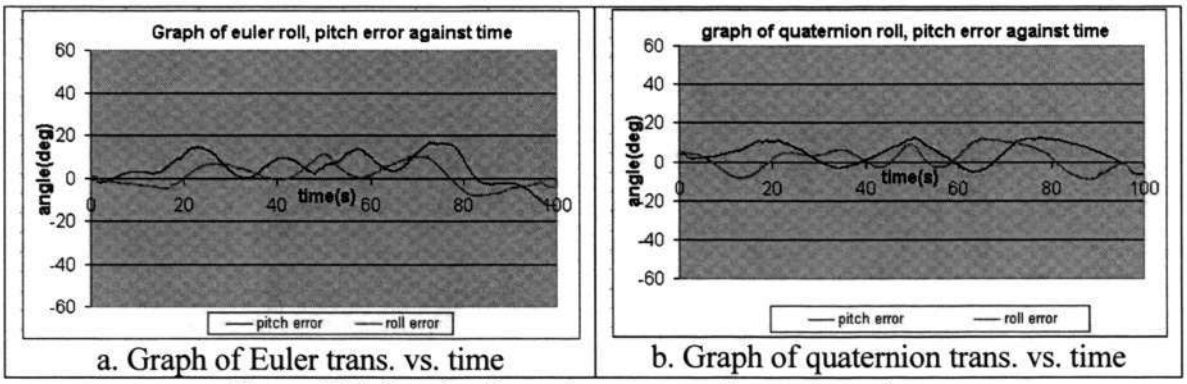
Figure B9 Graph of rate gyro and Kawasaki tilt angles against time for a planar rotation at 45° above the horizontal.



a. Graph of Euler trans. vs. time

b. Graph of quaternion trans. vs. time

Figure B10 Graph of angular rate transformation using Euler and quaternion transformation against time



a. Graph of Euler trans. vs. time

b. Graph of quaternion trans. vs. time

Figure B11 Graph of angular rate transformation using Euler and quaternion transformation against time

From figure B9 it can be seen that the IMU only produce yaw rate. However, since it is tilted about a 45° angle, the actual motion also results in roll and pitch motion as shown in figure B9(b). From the results, in figure B10 and B11, the Euler transformation reflects the actual roll and pitch angles more than the Quaternion transformation. Numerous documentations state the Quaternion transformation is capable of overcoming the 90° pitch angle error faced by the Euler transformation. However, as it undergoes more transformation, it is more susceptible to errors, which is reflected in the results obtained. In the URV application, the probability of the URV pitching at an angle of 90° is almost impossible. Therefore, the Euler transformation, with its better accuracy and lower computation time is suitable for the application.

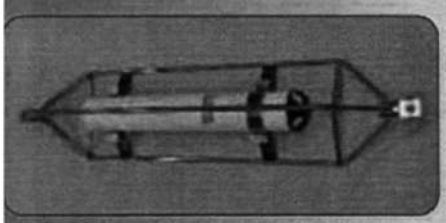
APPENDIX C
DATA SHEETS

PRODUCT SPECIFICATIONS

SonTek ARGONAUT®-MD

Argonaut-MD Housing Configurations
A. Standard MD
B. Short Term
C. Real Time

All Argonaut-MDs have a diameter of 10.2 cm (4.0"), and their length is affected only by the size of their power supply. The Standard Argonaut-MD has sufficient power for multi-year deployments and is 67.5 cm (26.6") long. The Short Term has power for deployments of a few weeks and is 44.5 cm (17.5") long. The Real Time system relies on outside power and is 35.6 cm (14.0") long.



Delrin Argonaut-MD in optional in-line load cage.

ARGONAUT-MD SPECIFICATIONS

Physical Parameters
- Weight in air: 5.6 kg / 12.5 lb. (Delrin housing)
  11.8 kg / 26 lbs. (Titanium housing)
- Weight in water: 1.4 kg / 3.0 lb. (Delrin housing)
  7.7 kg / 17 lbs. (Titanium housing)
- Pressure rating: 600 m (Delrin housing)
  6000 m (Titanium housing)

Standard Features
- Supports RS232, RS422, RS485 and SDI-12 communication protocols
- Flexible sampling strategies for reduced duty cycle operation and extended deployments
- 2 Mb internal memory (over 100,000 samples)
- Temperature sensor for automatic sound speed compensation
- Compass/tilt sensor to report velocity in Earth (East-North-Up) coordinates
- Integrated battery pack for autonomous operation (battery life sufficient for multi-year deployments dependent on sampling scheme)
- Battery Capacity:
  Standard: 368 W-hr
  Short Term: 67.2 W-hr

Optional Features
- Integrated CTD
- Integrated pressure sensor
- Inductive Modem
- Short Term Deployment and Real Time Packages
- 4MB Memory
- Mounting Clamps
- In-Line Load Cage

Velocity
- Range ±6 m/s
- Resolution 0.1 cm/s
- Accuracy ±1% of measured velocity, ±0.5 cm/s*
- User programmable data output rate from 10 seconds to 12 hours

Temperature Sensor
- Resolution 0.01°C
- Accuracy ±0.1°C

Compass/Tilt Sensor
- Heading ±2°, 2-axis tilt ±1°
- Built-in calibration procedure to compensate for ambient magnetic fields

Pressure Sensor
- Absolute accuracy ±0.1% of full scale
- Available full scale pressure ranges: 10 m, 20 m, 50 m, 100 m, 200 m, 600 m (call for depths greater than 600 m)
- Maximum deployment depth without sensor damage: 200% of full scale

Environmental
- Operating temperature -5° to 40°C
- Storage temperature -10° to 50°C

Power Requirements
- Input power 6-16 VDC
- Typical power consumption: 0.2 W (continuous operation); 0.001 W (stand-by)

SonTek's customer support is unsurpassed in the industry. Our experienced and professional staff is ready to assist you with the use and application of the Argonaut-MD.

SonTek®
The World Leader in Water Velocity Measurement

CORPORATE HEADQUARTERS
6837 Nancy Ridge Drive, Suite A
San Diego, CA 92121
Tel: (858) 546-8327
Fax: (858) 546-8150
e-mail: sales@sontek.com
website: www.sontek.com

MADE IN U.S.A.

Argonaut-MD v.6 • 6/00

Tilt Sensors

HIGH ACCURACY, DIGITAL SERIES

- Digital Roll and Pitch Output
- RS-232 Interface
- Programmable Resolution and Settling Time

Applications

- Platform Leveling
- Precision Tilt Measurements
- Geo-mechanical Leveling
- Lab Instrumentation

CXTILT02E

The CXTILT02E inclinometer offers outstanding resolution, dynamic response and accuracy. The CXTILT02E measures the tilt angle of an object with respect to the horizontal in a static environment. To measure tilt, also called roll and pitch, the sensor makes use of two micro-machined accelerometers, one oriented along the X-axis and one along the Y-axis.

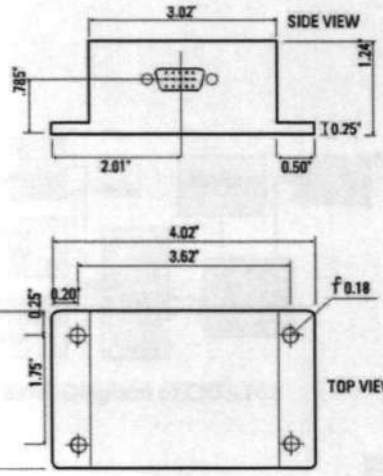
The CXTILT02E is a "smart" sensor with an embedded micro-controller, and A/D converter. The combination of sensing elements and digital electronics yields a system requiring no user calibration. The sensor's resolution and settling time are programmable, allowing the CXTILT02E to be customized for various applications.

CXTILT02EC

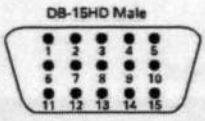
The CXTILT02EC provides superior performance in more demanding measurement applications, where high accuracy must be maintained over a wide temperature range of -40 to +85°C. The CXTILT02EC employs Crossbow's Softsensor® calibration and an onboard temperature sensor to internally compensate for temperature induced drift.

LINEARIZED

The CXTILT02EC is also characterized by very high linearity and designed specifically for use in construction environments. This extremely high linearity is achieved through Crossbow's proprietary Softsensor linearization calibration.

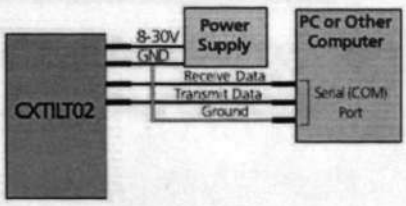


Aluminum Package



Specifications	CXTILT02E	CXTILT02EC	Remarks
Performance			
Accuracy (°)	±0.4	±0.1	at ± 20° (at 25°C)
Angular Range (°)	±75	±75	From horizontal
Angular Drift w/Temp (°)	1.5	0.7	Up to ± 20° Tilt
Angular Resolution (°)	See Table 1	See Table 1	
Settling Time (s)	See Table 1	See Table 1	
Null Angular Offset (°)	<0.5	<0.5	Actual value provided w/sensor
Non-Linearity (± 45°) (%)	<3	<0.3	Measured at 25° C
Transverse Sensitivity (%)	1	1	Typical
Environment			
Temperature Range (°C)	0 to +70	-40 to +85	
Electrical			
RS-232 Interface (Baud)	9600	9600	
Supply Voltage (Volts DC)	8-30	8-30	
Supply Current (mA)	60	60	
Physical			
Size	4.02 x 2.26 x 1.24" (10.21 x 5.74 x 3.15 cm)		
Weight	2.9 oz (90 gm)	2.9 oz (90 gm)	
Cable Length (feet)	2.5	2.5	

Specifications subject to change without notice



Typical CXTILT02 Configuration

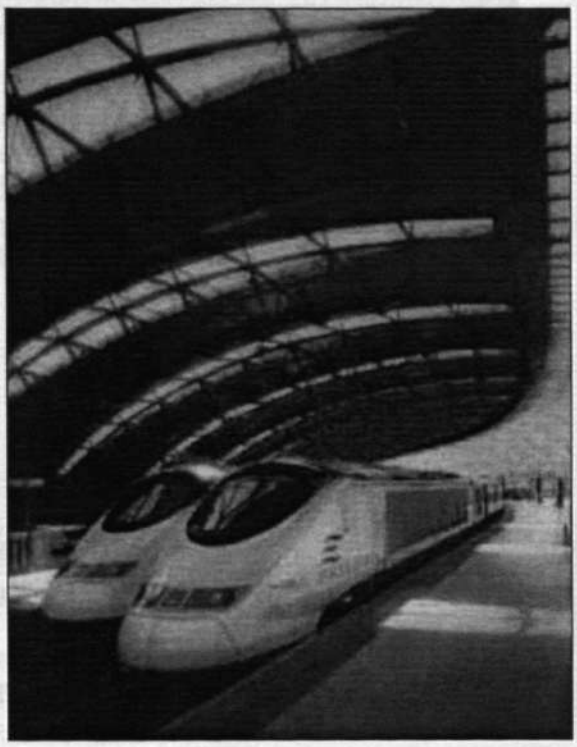
Pin	Function
4	Transmit Data
3	Receive Data
1	GND
11	Input Power

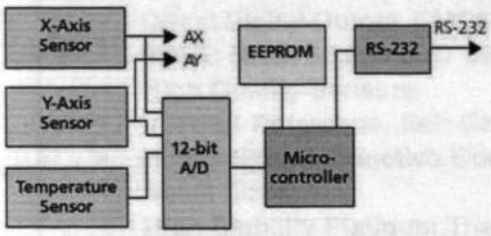
Pin Diagram

Filter	Fc(Hz)	Tc(s)	Typical Resolution (°)
0	none	none	0.320
1	12.5	0.025	0.101
2	9.9	0.029	0.091
3	5.0	0.042	0.064
4	2.5	0.074	0.045
5*	1.2	0.14	0.032
6	0.64	0.26	0.023
7	0.33	0.52	0.016
8	0.17	1.03	0.012
9	0.09	2.05	0.009

Note
Filter 0 implies no digital IIR filter, but there is a 125 Hz ± 35 Hz single pole response from the MEMS sensor
*Default setting

Table 1. CXTILT02 Programmable Filter Settings





Block Diagram of CXTILT02

Ordering Information

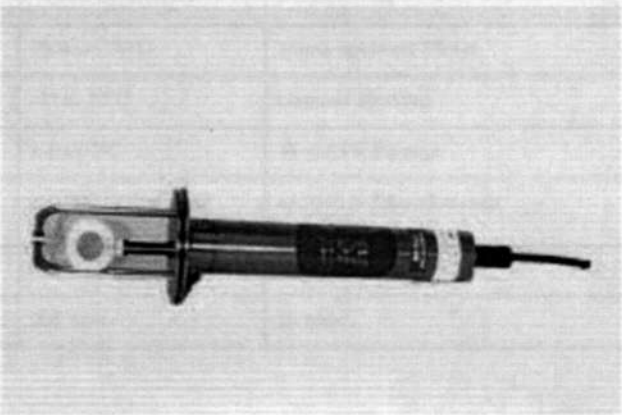
Model	Description	Angular Range	Temperature Range	Accuracy (± 20°)	Accuracy (± 45°)
CXTILT02E	Enhanced	± 75°	0 to +70°C	± 1.5°	± 3°
CXTILT02EC	Enhanced & Compensated	± 75°	-40 to +85°C	± 0.7°	± 1.2°

CALL FACTORY FOR OTHER CONFIGURATIONS

EXCELL® 2" MICRO CTD

Ultra-Low-Power Digital Conductivity, Temperature and Depth Measurement

The Excell® Micro-CTD combines innovative sensors and patented electronics to achieve the highest levels of accuracy and stability on a very low power budget (< 70 mW). All Excell® Micro CTD sensors are mounted in the flow, eliminating the requirement for a pump to feed sample water to the sensor (no pumps or other devices are required to correct for poor sensor flushing characteristics).



The innovative electronics use precise fixed references to provide continuous internal calibration for simplicity and increased reliability. Calibration constants are stored in internal non-volatile memory and can be user updated as required. The instrument can store up to 58,000 CTD data sets in full 18 bit resolution.

The Excell® 2" Micro CTD can also be used as a direct reading instrument outputting Conductivity Temperature and Depth in ASCII encoded physical units. The Excell® 2" Micro CTD uses a low power micro-controller to collect, scale, and transmit data using built-in CMOS, RS-485 or RS-232 interfaces. This instrument comes complete with Windows 95/NT Configuration / Acquisition Software and Post Processing Software for aid in data acquisition, real time graphing, and data analysis.

FEATURES

- **Ultra-Low-Power, <70 mW**
- **High Accuracy**
  - **± 0.0005 S/m Conductivity**
  - **± 0.005° C**
  - **± 0.03% Full Scale Pressure**
- **Direct Digital Output, CMOS, RS-232 or RS-485**
- **0.5MB RAM, 58,000 CTD Data Set Storage**
- **High Quality Sensors**
- **Internal Reference, Self-Calibrating Electronics**
- **Highly Stable Inductive Conductivity Sensor**
- **Small Size**
- **High Stability Platinum Thermometer**
- **Micro-Machined Silicon Pressure Sensor**
- **No Electrodes to Foul**
- **No Pumps to Fail**

Excellence In Instrumentation

FSI Falmouth Scientific, Inc.

1400 Route 28A, P.O. Box 315 • Cataumet, MA 02534-0315 • email: fsi@falmouth.com
Telephone: 508/564-7640 • Facsimile: 508/564-7643 • Website: www.falmouth.com

Excell® 2" Micro CTD Specifications

Sensors

	Conductivity (S/m)	Temperature	Pressure
Sensor Type	FSI Inductive Cell	Platinum RTD	Micro-machined Silicon
Range	0 - 7.0 S/m (0 - 70 mS/cm)	-2° to 32°C	customer specified
Accuracy	± 0.0005 S/m (±0.005 mS/cm)	± 0.005°C	+/- 0.03% full scale
Stability	± 0.0005 S/m/month (±0.00005 S/m/month)	± 0.0005°C/month	+/- 0.01% full scale/month
Resolution	0.00001 S/m (0.0001 mS/cm)	0.001°C	0.001% full scale
Response	5.0 cm at 1 m/sec flow	500 msec	25 msec

System

Power	6 to 14 VDC at 12 mA, 70 mW maximum
Depth Rating	500 meter Delrin housing or Deep - 7000 meter Titanium housing
Dimensions	2.04 inch od x13.8 inches long
Sample Rate	Fixed 1.83 CTD frames per second logging, - 2.0 Frames Direct Readout
Resolution	20 Bits A/D, Memory Stored to 18 Bits
Memory	512K Bytes SRAM, 10 Year Battery Backed, 56,000 CTD Data Sets
Data Format	Conductivity - mS/cm Temperature - ° Celsius (ITS-90) Pressure - decibars, (SNNNNN.NN) All data in ASCII, 8 data bits, one stop bit, no parity CMOS or RS-232 or RS-485
Baud Rate	9600 Baud, 14400, or 28,800
Calibration	NIST Traceable @ FSI Per UNESCO 44 Requirements

Specifications Subject to Change without Notice

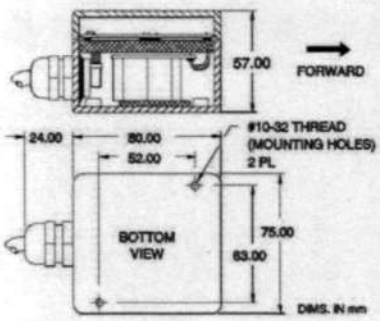
Excellence In Instrumentation

Rev. V(2/27/02)
1400 Route 28A, P.O. Box 315 • Cataumet, MA 02534-0315 • email: fsi@falmouth.com
Telephone: 508/564-7640 • Facsimile: 508/564-7643 • Website: www.falmouth.com

KVH's C100™ Compass Engine is an innovative stand-alone design offering small size, low cost, and system flexibility to meet your demanding heading sensor applications.

KVH C100™ Specifications

KVH C100™ Compass Engine Technical Specifications



Optional aluminum housing

Physical	
Dimensions	114 mm(l) x 46 mm(w) x 28 mm(h) (4.5" x 1.8" x 1.1")
Weight	64 grams (2.25 oz.) – with SE-25 coil
Input Voltage	+8 to +18VDC or +18 to +28VDC (user selectable)
Current Drain	40 mA DC; maximum
Optional Aluminum Housing[1]	
Dimensions	80 mm(l) x 75 mm(w) x 57 mm(h) (3.15" x 2.95" x 2.25")
Weight	400 grams (14 oz.) excluding cable
Performance	
Accuracy[2]	±0.5° or ±10 mils RMS (SE-25 coil assembly and digital outputs)
Repeatability	±0.2° or ±5 mils (SE-25 coil assembly and digital outputs)
Resolution	0.1° or 1 mil
Dip Angle	±80° (maintains stated accuracy after auto-compensation up to ±80° magnetic dip angle)
Tilt Angle[2]	±16°; Dev .= ±0.3° RMS (SE-25) ±45°; Dev. = ±0.5° RMS (SE-10)
Declination/Variation	±180.0° adjustment range (offset) (user selectable)
Index Error Correction	±180.0° adjustment range (offset) (user selectable)
Response Time	0.1 to 24 seconds (user selectable)
Environmental	
Operating Temp	-40° C to +65° C (-40° F to +150° F)
Storage Temp	-57° C to +71° C (-71° F to +160° F)
Shock/Vibration	Designed to meet MIL-STD-810 shock and vibration requirements
Altitude	Designed to meet 12,192 meters (40,000 ft.) MSL (operating/non operating)
Reliability	MTBF calculated to >30,000 hours

[1] *(SE-25 coil option only)*

[2] *Accuracy measurements apply to a level compass module after compensation in a free magnetic field. After installation and auto-compensation, typical accuracies of ±0.5° are achievable on most platforms.*

KVH C100™ Compass Engine Technical Specifications – continued

Outputs and Interfaces

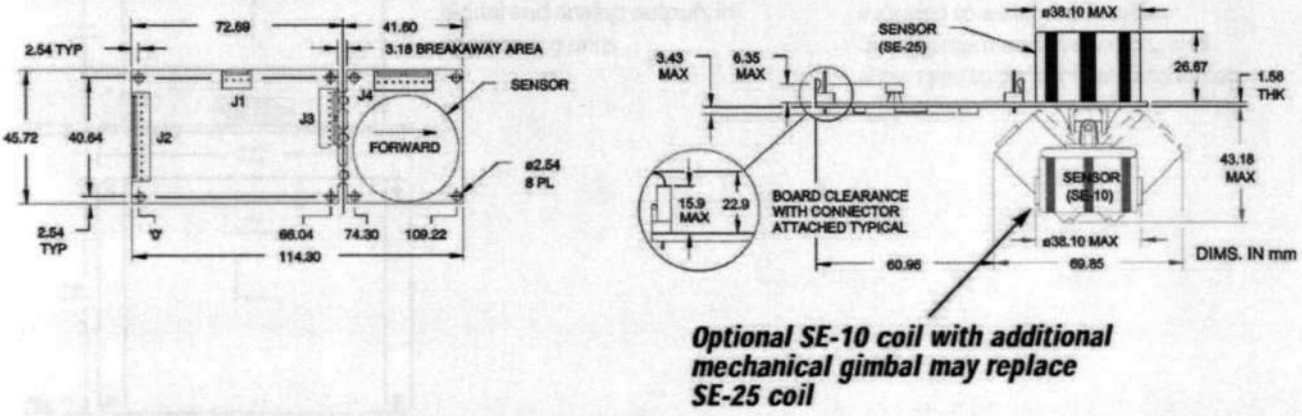
Digital Interfaces[3]

RS232 Compatible	Bidirectional Serial Data, UART format w/ASCII characters, (1 start, 8 data, 1 stop, no parity), 300 - 9600 baud (user selectable)	
Serial Input	Accepts RS232 levels or 0 to +5V logic levels	
Serial Output	Same as RS232 except for 0 to +5V logic levels; (Logic "0" = +5V) 10K Ohm - min. load	
Inv. Serial Output	Same as Serial Output except logic "1" = +5V	
NMEA 0183	NMEA 0183 compatible bidirectional data/levels/formats	
Synchronous	Strobe Input:	Ground momentarily to obtain data output
	Clock Output:	0 to +5V sq.wave @ 10KHz rate
	Data Output:	0 to +5V levels
	Data Format:	4 digit BCD, 16 bit binary or 16 bit serial gray code – user selectable through serial port 10K Ohm minimum load on output signals

Analog Outputs

Sine/Cosine	Output Voltage: +1.5V ±1.0V Reference Voltage: +1.5VDC; 20K Ohm minimum load capability
Linear Voltage	0.1 to 1.9VDC into 20K Ohm minimum load

[3] *Digital outputs can be user configured to provide strobed or free running data at up to a 10Hz message rate.*



Optional SE-10 coil with additional mechanical gimbal may replace SE-25 coil

KVH Industries, Inc.
Beijing FCWY Tech. Co., China
Phone: 010 68745180 fax: 010 68745811
E-Mail: info@GPS8848.com
Internet: http://www.GPS8848.com

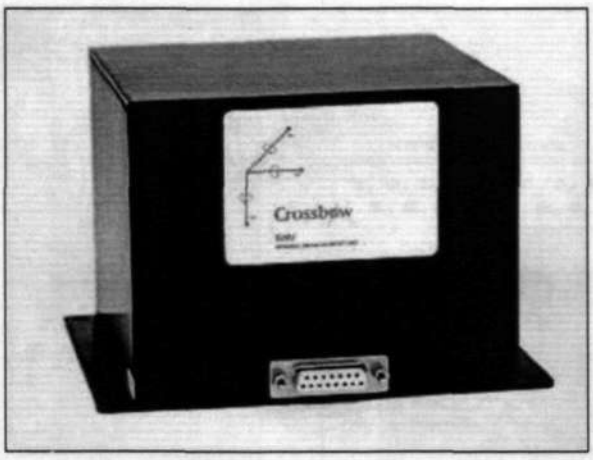
IMU

FIBER OPTIC GYRO BASED IMU

- High Bandwidth Angular Rate and Linear Acceleration Measurement
- Fiber Optic Gyro Stability
- Analog and Digital Outputs
- No Calibration Required

Applications

- Navigation and Control
- Avionics



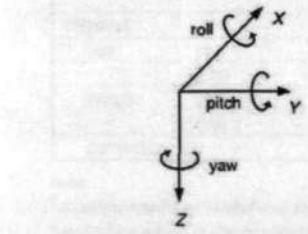
IMU600CA (DMU-FOG-6)

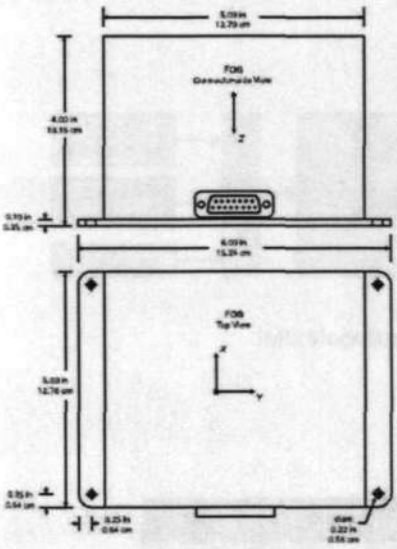
The IMU600CA is a six-degree-of-freedom (6DOF) Inertial Measurement Unit that provides accurate monitoring of linear acceleration and angular rate. The IMU600CA uses the latest in fiber optic rate gyro technology resulting in superior performance, reliability, and stability over time. Example applications include GPS/INS navigation, dynamic positioning and avionics.

The IMU600CA offers wide bandwidth measurement of acceleration and rotation rate about three orthogonal axes. The IMU600CA employs on board digital processing to sample data, compensate for deterministic errors, and format digital and analog outputs in engineering units.

The IMU600CA sensing elements are solid-state devices that have no moving parts. The three fiber optic rate gyros employ the sagnac effect to measure angular rate independently of acceleration. The three accelerometers are silicon MEMS devices that use differential capacitance to sense acceleration. The IMU600CA has three output options (one analog and two digital modes) to allow for easy integration into measurement and control systems.

Each Inertial System comes with a User's Manual offering helpful hints on programming, installation, and product information. In addition, Crossbow's GYRO-VIEW software is included to assist you in system development and evaluation, and allows you to perform data acquisition.





Specifications		IMU600CA-200	IMU600CA-201	Remarks
Performance				
Update Rate (Hz)		> 100	>100	Continuous Update Mode
Start-up Time Valid Data (sec)		< 1	< 1	
Angular Rate[1]				
Range Roll, Pitch, Yaw (°/sec)		± 200	± 200	
Bias: Roll, Pitch, Yaw (°/sec)		< ± 0.03	<± 0.03	
Scale Factor Accuracy (%)		< 1	< 1	
Non-Linearity (% FS)		< 0.2	< 0.2	
Resolution (°/sec)		< 0.025	< 0.025	
Bandwidth (Hz)		> 100	> 100	-3 dB point
Random Walk (°/hr$^{1/2}$)		< 1.25	< 1.25	
Acceleration				
Range X/Y/Z (g)		± 2	± 10	
Bias: X/Y/Z (mg)		<± 8.5	<± 12	
Scale Factor Accuracy (%)		<± 1	<± 1	
Non-Linearity (% FS)		<± 1	<± 1	
Resolution (mg)		< 0.25	< 1.25	
Bandwidth (Hz)		> 100	> 100	-3 dB point
Random Walk (m/s/hr$^{1/2}$)		< 0.1	< 0.5	
Environment				
Operating Temperature (°C)		-25 to +71	-25 to +71	
Non-Operating Temperature (°C)		-55 to +85	-55 to +85	
Non-Operating Vibration (g rms)		6	6	20 Hz - 2 KHz random
Non-Operating Shock (g)		1000	1000	1 ms half sine wave
Electrical				
Input Voltage (VDC)		15 to 30	15 to 30	
Input Current (A)		< 1	<1	
Power Consumption (W)		< 15	< 15	at 15 VDC
Digital Output Format		RS-232	RS-232	See "Digital Data Format"
Analog Range (VDC)		± 4.096	± 4.096	Pins 8, 9, 10, 12, 13, 14
		0 to 5.0	0 to 5.0	Pins 5, 6, 7
Physical				
Size	(in)	5.0 x 6.0 x 4.0	5.0 x 6.0 x 4.0	
	(cm)	12.70x15.24x10.16	12.70x15.24x10.16	
Weight	(lbs)	< 3.5	< 3.5	
	(kg)	< 1.6	< 1.6	
Connector		15 pin sub-miniature "D" female		

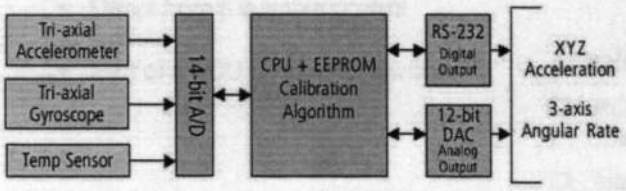
Notes:
[1] All analog outputs are fully buffered and are designed to interface directly to data acquisition equipment.
Specifications subject to change without notice

15 Pin "D" Connector Socket Pinout

Pin	Function
1	RS-232 Transmit Data
2	RS-232 Receive Data
3	Input Power
4	Ground
5	X-axis accel voltage[1]
6	Y-axis accel voltage[1]
7	Z-axis accel voltage[1]
8	Roll-axis angular rate[2]
9	Pitch-axis angular rate[2]
10	Yaw-axis angular rate[2]
11	NC – Factory use only
12	Roll angle/X-axis acceleration[3]
13	Pitch angle/Y-axis acceleration[3]
14	Not used/Z-axis acceleration[3]
15	NC – Factory use only

Notes
1 The accelerometer voltage outputs are taken directly from the accelerometers without compensation or scaling.
2 The angular rate analog outputs are scaled to represent degrees/second. Outputs are created by a D/A converter.
3 Actual output depends on IMU measurement mode.

Pin Diagram



IMU Block Diagram

Ordering Information

Model	Previous Model	Description	Gyro (°/sec)	Accel (g)
IMU600CA-200	DMU-FOG-6	Fiber Optic IMU	± 200	± 2
IMU600CA-201	DMU-FOG-6	Fiber Optic IMU	± 200	± 10

CALL FACTORY FOR OTHER CONFIGURATIONS

phone: 408.965.3300 ▾ fax: 408.324.4840 ▾ e-mail: info@xbow.com ▾ web: www.xbow.com

Tritech International Limited

Digital Precision Altimeters

PA 200 and PA500

Tritech PA200 and **PA500 Digital Precision Altimeters** are based on the same high performance electronics found in the well-proven, industry standard **SeaKing** sonar and profiler systems.

The range of compact **Tritech PA200** and **PA500 Digital Precision Altimeters** provide exceptionally accurate height off seabed and subsea distance measurements.

Full digital synthesis of transmit and receive frequencies, together with greatly improved input dynamic range, offer unsurpassed levels of performance from a compact unit.

The units may be supplied with simultaneous analogue and digital outputs allowing them to be interfaced to a wide range of PC devices, data loggers, ROV telemetry systems and multiplexers.

OEM configurations including low magnetic signature, rigid polyurethane housings or right-angled heads are available on request.

Control of the altimeters may be performed in many ways including :

- Direct from a PC running DOS
- Direct from a PC running Windows
- Direct from a suitable control system
- Part of a SCU-3 RS485 network

Features

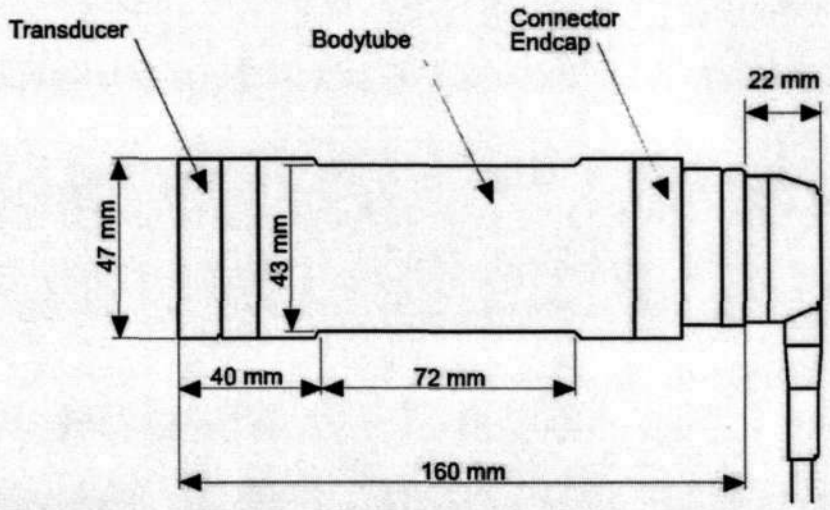
- *Analogue only outputs*
- *Digital only output*
- *Simultaneous analogue and digital outputs*
- *Noise immunity*
- *Can be field reconfigured for different applications*
- *Free-running outputs*
- *Interrogated outputs*
- *Low operating voltage options*
- *4000 metre standard depth rating*

Applications

- *ROV and AUV auto-altitude and under-ice measurement*
- *Integration to CTD & oceanographic sensor packages*
- *Low-cost hydrographic surveying of rivers and canals*
- *Integration to bathymetric packages*
- *Touchdown monitoring of subsea structures*
- *Wave height measurement*
- *Monitoring of scouring around bridge supports*

Tritech PA200 and PA500 Digital Precision Altimeters

Specification



all measurements in mm (inches)

	PA200-20	PA500-6
Operating frequency	200 kHz	500 kHz
Beamwidth	20° Conical	6° Conical
Operating range	1 to 100 metres	0.3 to 50 metres
	0.7 to 50 metres	0.1 to 10 metres

Standard power supply	24 VDC @ 80 mA, or 12 VDC @ 160 mA
Standard analogue output	0 to 10 VDC (24 VDC units only), or 0 to 5 VDC
Digital resolution	1mm on all ranges
Analogue resolution	0.025% of range
Data communication options	Serial RS232, or Serial RS485
Output modes	Free running, or Interrogated, or Tritech SCU-3 network
Serial output format	ASCII, 9600, 1, 8, n, 1
Length	160mm
Diameter	47mm
Standard depth rating (stainless steel body)	4,000 m
Optional depth ratings	700m 2,000m and 6,800m
Weight in air	1.1 kg
Weight in water	0.8 kg

All specifications are subject to change in line with Tritech's policy of continual product development. *Ref: altimeter.pm65 Issue 10 Feb 02*

Tritech International Limited
Peregrine Road
Westhill Business Park
Aberdeen AB32 6JL
United Kingdom
Tel. : ++ 44 1224 744111
Fax : ++ 441224 741771
Email : sales@tritech.co.uk
http://www.tritech.co.uk

Marketed by:

SCOTTISH OFFSHORE ACHIEVEMENT AWARDS