5-2007

# Software Development for Mechatronics Systems: Simulation of Multiple UAV Navigation and Device Driver Interface for the Q4 Data Acquisition Card

Ninad Pradhan
*Clemson University*, npradha@clemson.edu

# SOFTWARE DEVELOPMENT FOR MECHATRONICS SYSTEMS: SIMULATION OF MULTIPLE UAV NAVIGATION AND DEVICE DRIVER INTERFACE FOR THE Q4 DATA ACQUISITION CARD

---

A Thesis
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Electrical Engineering

---

by
Ninad Pradhan
May 2007

---

Accepted by:
Dr. Darren Dawson, Committee Chair
Dr. Ian Walker
Dr. Timothy Burg

ABSTRACT

Mechatronics systems are often a close interleaving of different aspects of hardware, software, control and mechanical design. Two such aspects are delved into in greater detail for this thesis, both related to software development.

Simulations are used to know the practical conditions under which a control design is effective. The system simulated in this case was a multiple vehicle system, where the vehicles navigated from source to a destination configuration avoiding two different kinds of obstacles.

Post simulation, a system is implemented using hardware and tested using an experimental setup. Data Acquisition Boards such as the Quanser Q4 are used to read the system state. A device driver interface containing a wide variety of functions was developed and extensively tested for the Q4 PCI bus card.

Both these projects were software development efforts towards contributing to different aspects of Robotics and Mechatronics projects in the Controls and Robotics Group.

# DEDICATION

I dedicate this work to my parents, Nilkanth and Aparna Pradhan, for it was their patience, belief and encouragement that made it possible.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Darren Dawson, for his guidance and emphasis on persistence and independence with work. I would also like to thank Dr. Timothy Burg for serving on my committee. I would like to thank Dr. Ian Walker for his good counsel and empathy, whenever it was sought.

I would like to thank Vilas for his help and guidance with the device driver project, and Jian, whose research Erhun and I simulated during our first year.

I would like to acknowledge and thank my lab partner, Erhun Iyasere. I learnt a lot from his technical insight and creativity at work, all laced with diligence and good humor. I would also like to thank Vikram Iyengar, for his support and his infectious enthusiasm for all things, including research and engineering. I would also like to thank another group mate and friend, Nitendra Nath, for his candid and constructive opinions whenever I was looking for them. I would like to thank my friend Rahul Saxena, for being there as always and bearing the brunt of my questions during my learning curve on the device driver project.

I would like to offer special thanks to Kaveri for her motivation and presence at all times, which made much of this possible. I would also like to thank all my friends from VG 110 and 127, especially Radhika, Neha and Utpal, who have been my support system for the past two years, and more.

Finally, I would like to thank my parents, my sister Meeta and aunt Kunda for their love and unconditional support and guidance. None of my endeavors would have fructified without it.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF FIGURES

# CHAPTER ONE

# INTRODUCTION

The field of Mechatronics brings together, under its umbrella, diverse areas of engineering theory and practice. The projects in this thesis provided an opportunity to explore problems in two such areas.

The first was a simulation of a cooperative control algorithm which used a navigation function to guide a group of mobile robots from a source position to a goal configuration. The navigation function built on the theory of potential fields, which repel robots from known obstacle locations while attracting them towards the goal position. This navigation function has been designed to avoid getting stuck at local minima when realizing the goal configuration. The robots were assumed to be unmanned aerial vehicles (UAVs) and their region of operation a combat zone. The task for the UAVs was to move to the target configuration in the combat zone while avoiding collision with stationary obstacles, and avoiding being detected by enemy assets (EAs). The EAs were each assumed to have a detection zone, and problem was formulated with certain assumptions regarding the detection range and capabilities of the mobile agents in question.

Simulations were carried out for cases where the EAs were assumed to be stationary, and mobile. Conditions for the simulation were varied, such as the area of the stationary obstacles, the sensing and detection zones of mobile agents (both UAVs and EAs), and the initial and final configuration of the UAV group. For each change in the system configuration, system gains were tuned to achieve desired performance. The sensitivity of the system to these variations provided feedback on the performance of the centralized navigation algorithm proposed in the publication.

In contrast to the theoretical background to the first project, and its subsequent simulation, is the second project, which is the design and implementation of a device driver for a Data Acquisition and Control Card.

The device driver made it possible for the PCI based Q4 Data Acquisition and Control Card, developed by Quanser Consulting Inc., to be used on a QNX system. The Q4 is a versatile system which contains a number of ADC, DAC and Encoder Channels, Digital Input/Output Pins, and External Signal and Interrupt Lines. Prior to the development of this software, the Q4 could only be run on the Windows platform.

Migration to the QNX platform would be beneficial to the Control and Robotics Group, since QNX based motor control software has been extensively used in and developed for experiments. The Q4 would provide the Data Acquisition functionality to compliment this control software; this functionality being currently supplied by the ServoToGo Board. The Q4 not only provides all the functionalities of the ServoToGo, and more, but also uses the PCI bus, unlike the ServoToGo's use of the increasingly obsolete ISA architecture.

There were numerous challenging sub-problems which needed addressing before the Q4 Device Driver Interface could be implemented. Initial hurdles were faced in communicating with the card. QNX system commands were adapted for Interrupt processing, to deal with 'events' which would be the means for the card to satisfy requirements for speed in performance, vital to most control based applications. Once interrupts could be serviced, it was necessary to set the card up in a way that the user application would be able to write to and read from card Registers, which were the means of configuring a mode of operation for the Q4.

The next stage was the development of a user friendly interface to the functions that the card had to offer. A number of functions were written so that the user could exercise the maximum possible degree of control over card configuration, and make changes to the configuration using intuitive function calls and arguments. At each stage during the software development process, numerous test programs were written, and all code was carefully commented so as to simplify future modifications to the software. The final driver interface was also exhaustively commented and documentation was generated so that a user manual was available to supplement the software.

It has been an endeavor to produce a thesis manuscript that provides adequate information on both projects, with a view towards not only outlining what has been done during the execution of these by the author, but also assist, in whatever way possible, future simulations and software development in this research group.

## Thesis Organization

This thesis discusses two software development projects for the Controls and Robotics Group. Chapter One introduces both projects and discusses significant aspects of their objectives and implementation.

Chapter Two delves into the development of a control law for the Cooperative Control of Multiple Vehicles with Finite Sensor Range. The navigation algorithm is developed for a case where the vehicles or UAVs encounter stationary obstacles and enemy assets, and is extended for a case where the enemy assets themselves are mobile. Chapter Three discusses the simulation of the system for the staionary Enemy Assets case, and provides figures for the UAV trajectories from source to their convergence to the destination configuration.

Chapter Four gives the design objectives for the device driver interface, and introduces QNX and Q4 features which form the basis of the implementation. The actual implementation of the driver Class, and testing of the software using standalone programs and QMotor is detailed in Chapter Five.

A detailed listing for the driver interface, an example program and derivation of a formula used in the Class are included in the Appendices, as is the layout for the multiple UAV simulation.

# CHAPTER TWO

# COOPERATIVE CONTROL OF MULTIPLE VEHICLES WITH LIMITED SENSING

A navigation function based cooperative control is developed in this paper for the navigation of multiple Unmanned Aerial Vehicles (UAVs) in the presence of known stationary obstacles and unknown enemy assets (EAs). Specifically, the motion of UAVs are planned in a centralized fashion. The standard navigation function approach is extended to a multiple navigation strategy with an analytical switch among different cases due to the limited sensing zone of the UAVs. A differentiable controller is proposed based on this navigation function that yields asymptotic convergence. A discussion for avoiding moving EAs is presented. Simulation results are provided to illustrate the performance of the proposed control strategy.

## Introduction

Numerous researchers have proposed algorithms to address the motion control problem associated with robotic task execution in an obstacle cluttered environment. A comprehensive summary of techniques that address the classic geometric problem of constructing a collision-free path and traditional path planning algorithms is provided in Section 9, ``Literature Landmarks", of Chapter 1 of [16]. Since the pioneering work by Khatib in [11], it is clear that the construction and use of potential functions has continued to be one of the mainstream approaches to robotic task execution in the presence of obstacles. In short, potential functions produce a repulsive potential field around the robot workspace boundary and obstacles and an attractive potential field at the goal configuration. A comprehensive overview of research directed at potential functions is provided in [16]. One of the criticisms of the potential function approach is that local minima can occur that can cause the robot to ``get stuck" without reaching the goal position. Several researchers have proposed approaches to address the local minima issue (e.g., see [1], [2], [6], [12], [27]). One approach toaddress

the local minima issue was provided by Koditschek in [13] for holonomic systems (see also [14] and [20]) that is based on a special kind of potential function, coined a navigation function, that has a refined mathematical structure which guarantees a unique minimum exists. For other related work that has focused on the development of potential function-based approaches, the reader is referred to [3], [9], [10], [15], [17], [20], [25], and [26].

By leveraging previous results directed at classic (single robot) systems, more recent research has focused on the development of potential function-based approaches for the more challenging multiple agent system. For example, Loizou et al. [18] extended the navigation function methodology, established for single robot navigation, to the case of multiple robots. [19] presented an extension to the navigation function methodology to the case where unmodelled obstacles are introduced in the workspace. The approach of [19] constructs a potential field that models all known environment features and combines it with an efficient control scheme that handles additional unknown features. In [4], gyroscopic forces and scalar potentials were utilized to create swarming behaviors for multiple agent systems. As stated in [4], two main assumptions, the kinetic energy of the vehicle is bounded and only one obstacle is present in the detection shell, limit the application of the approach. A decentralized motion control approach was proposed in [8] in which the agent reacts with the agents in its neighborhood. No obstacles were considered in [8]. In [24], the flocking motion remains stable as long as the graph that describes the neighboring relations among the agents in the group is always connected. For local agent interaction, a nonsmooth potential function was applied between two agents under a local interaction regime. Additionally, demonstration of the nonsmooth control law in [23] showed the use of local sensing (limited sensing distance) to affect motion but not affect the stability properties of the group. This type of system falls into the category of differential equations with discontinuous right hand sides, and hence, the stability tools are generally based on nonsmooth analysis. More recently, Olfati-Saber et al. presented a theoretical framework for the design and analysis of distributed flocking algorithms in the presence of multiple obstacles [21].

The goal of this paper is to develop a cooperative control algorithm for the multiple UAV

problem to ensure that all of the UAVs will: i) remain in the combat zone, ii) avoid collisions with other UAVs, iii) avoid collisions with predetermined stationary objects, iv) avoid being detected by EAs, and v) ensure each UAV moves to its desired location in the combat zone. The unique aspect of this approach is that the UAVs only act with respect to the EAs within its sensing zone. That is, the navigation function is analytically switched when EAs enter the sensing zone of UAVs without corrupting the stability and convergence properties of the navigation function. The proposed approach can be applied to numerous applications[1] such as mobile sensor network deployment, pre-flocking (gathering the UAVs to a specific formation in a certain location), and suppression of enemy air defense using a cooperative group of UAVs.

This paper is organized in the following manner. In Section 2.1, the problem is formulated for the multiple UAVs navigation. In Section 2.2, a smooth bump function, a boundary function, obstacle functions, EA detection zone functions, and UAV collision functions are developed. In Section 2.3, a multiple UAV navigation function is designed, and a cooperative control algorithm is developed. A discussion for avoiding moving EAs is presented in Section 2.4. Simulation results illustrating the performance of the cooperative control algorithm are given in Chapter 3, and concluding remarks are given following the simulation results.

[1]Although the vehicle model in this manuscript is the UAV, it can be applied to any multiple agent system which has various applications and has attracted significant attention as stated in [24].

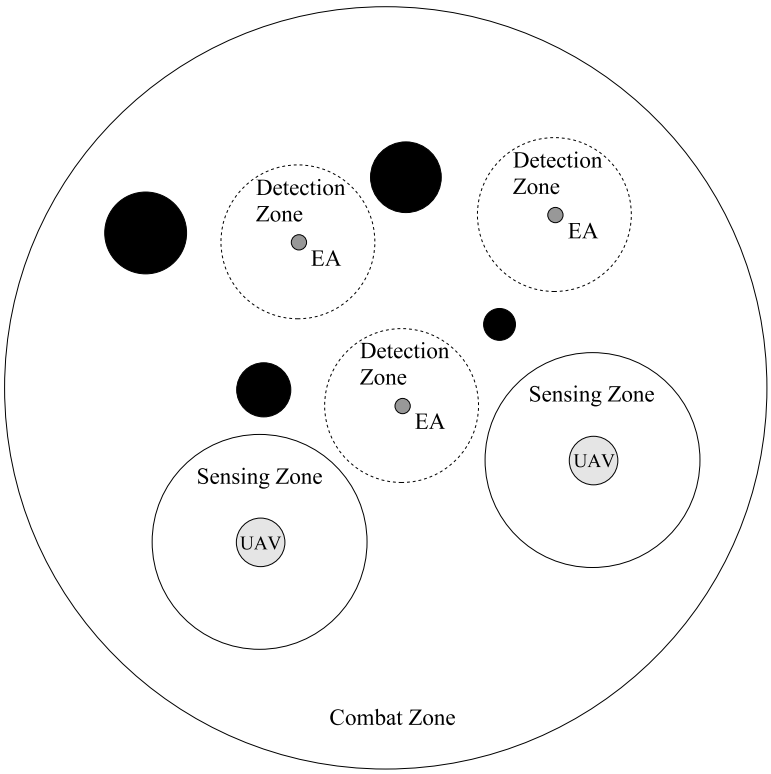


**Figure 2.1 Multiple UAVs in Combat Zone**

Problem Formulation

As illustrated by Figure (2.1), all UAVs are assumed to be initially in the combat zone (i.e., the large circle), which is a configuration space with a boundary. In the combat zone, there are some stationary obstacles and EAs. All UAVs have an onboard sensor, which can only sense objects in its neighborhood denoted as the sensing zone (i.e., due to the physical limitation of the onboard sensing capability of each UAV). With a limited detection range, the EAs can also only detect objects in its neighborhood denoted as the detection zone. The cooperative control objectives for the multiple UAVs are to ensure that all of the UAVs will:

- Objective 1: Remain in the combat zone.
- Objective 2: Avoid collisions with other UAVs.
- Objective 3: Avoid collisions with the predetermined stationary objects.

- Objective 4: Avoid being detected by EAs.
- Objective 5: Ensure that each UAV moves to its desired location in the combat zone.

To address the multiple UAV cooperative control problem in this paper, we use the following assumptions:

- Assumption 1: The physical dimensions of the combat zone are assumed to be known by all of the UAVs.
- Assumption 2: The location and the physical dimensions of the stationary obstacles are known within the combat zone by all of UAVs.
- Assumption 3: The UAV sensing zone and the EA detection zone are much greater than the size of the EA; therefore, the EAs can be treated as points.
- Assumption 4: The detection zone of the EA is known by all of UAVs.
- Assumption 5: The sensing zone of the UAVs is greater than the detection zone of the EAs (otherwise it is impossible for the UAVs to avoid being detected by the EAs).
- Assumption 6: Within the finite combat zone, all UAVs have access to all of the other UAVs' position.
- Assumption 7: Initially, all of the UAVs are in a non-contact configuration in the combat zone and do not contact the EAs, stationary obstacles, and the boundary of the combat zone.

## Model Development

To explain the approach, we first transform the physical geometry of each UAV into a sphere with $q_i(t) \in \mathbb{R}^3$ and $r_{zi} \in \mathbb{R}, i = 1, 2, \cdots, n,$ denoting the time-varying center and the constant radius of the $i$ th UAV sphere, respectively, and $n$ denotes the number of the UAVs. The composite configuration for the UAV $q(t) \in \mathbb{R}^{3n}$ is defined as

$$q(t) = \begin{bmatrix} q_1^T(t) & q_2^T(t) & \cdots & q_n^T(t) \end{bmatrix}^T . \qquad (1)$$

We will assume the multiple UAV system can be described by the following kinematic model:

$$\dot{q} = u \tag{1}$$

where $u(t) \in \mathbb{R}^{3n}$ is the cooperative control input. The goal configuration of the UAVs is denoted as $q^* \in \mathbb{R}^{3n}$ and defined by

$$q^* = \begin{bmatrix} q_1^{*T} & q_2^{*T} & \cdots & q_n^{*T} \end{bmatrix}^T \tag{2}$$

where $q_i^* \in \mathbb{R}^3, i = 1, 2, \cdots, n,$ denotes the desired constant location of the $i$ th UAV.

Smooth Bump Function

The multiple UAV navigation problem is complicated by the fact that the sensing zone of the UAVs is finite. To deal with this problem, a smooth bump function is introduced to analytically switch the repulsive force generated by the EAs and the boundary of the combat zone from zero to one. Indeed, it is this type of analytical smoothing approach that allows the boundary functions and EA detection functions developed subsequently to be appropriately smoothed out to facilitate the gradient-like differentiation that is utilized to construct the cooperative control algorithm.

Definition 1 *The smooth bump function, denoted by* $\rho_h(\cdot): \mathbb{R}_+ \to [0,1]$*, is a scalar function satisfying* [22]

$$\rho_h(x) = \begin{cases} 1 & 0 \leqslant x \leqslant h \\ 0 & x \geqslant 1 \\ \in (0,1) & h < x < 1 \end{cases} \tag{3}$$

where $\mathbb{R}_+$ denotes non-negative real number, and $h \in (0,1)$ is a positive constant parameter of the bump function.

The smooth property of the bump function defined in ((3)) is illustrated in Figure (2.2).

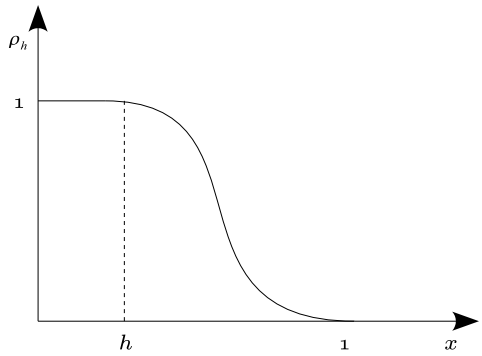


**Figure 2.2 Smooth Bump Function**

Based on the definition of the bump function in ((3)), one possible choice is the following bump function introduced in [22]

$$\rho_h(x) = \begin{cases} 1 & x \in [0, h] \\ \frac{1}{2}\left[1 + \cos\left(\pi \frac{x-h}{1-h}\right)\right] & x \in (h, 1) \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Boundary Function

To ensure that the UAVs remain within the predefined combat zone, a scalar boundary function is introduced to determine the relationship between the boundary of the combat zone and the position of the UAVs. The objective in selecting a boundary function is to prevent any UAV from contacting any part of the combat zone boundary.

Definition 2 *The boundary function, denoted by* $\beta_0(\cdot) \in \mathbb{R}$*, is a function satisfying*

$$\beta_0 = \begin{cases} 0 & \text{At least one UAV contacts the boundary of the combat zone} \\ 1 & \text{All UAV are inside the combat zone and the boundary of the combat zone is outside the sensing zone of UAVs} \\ \in (0,1) & \text{Otherwise.} \end{cases} \tag{5}$$

As indicated by ((5)), the function $\beta_0(\cdot)$ approaches zero if any of the UAVs approach the boundary of the combat zone, and hence, this type of design for $\beta_0(\cdot)$ spawns a gradient-like term to ensure that the UAV moves away from the boundary of the combat zone. Based on ((3)) and ((5)), the boundary function can be designed as follows

$$\beta_0 = \prod_{i=1}^{n} \rho_{h_{0i}} \left( \frac{1}{r_{o0} - r_{zi}} \| q_i - q_{o0} \| \right) \tag{6}$$

where $q_{o0} \in \mathbb{R}^3$ denotes the position of the center of the combat zone, and $r_{o0} \in \mathbb{R}$ denotes the radius of the combat zone. From ((5)) and ((6)), it is clear that: i) $\beta_0 = 0$ when for any $i$, $\| q_i - q_{o0} \| \geqslant r_{o0} - r_{zi}$, and ii) $\beta_0 = 1$ only when for all $i$, $\| q_i - q_{o0} \| \leqslant h_{0i} (r_{o0} - r_{zi})$. Based on ((3)), ((5)), and ((6)), the bump function parameter $h_{0i}$ in ((6)) can be designed as follows

$$h_{0i} = \frac{r_{o0} - r_{si}}{r_{o0} - r_{zi}} \tag{7}$$

where $r_{si} \in \mathbb{R}$ denotes the radius of the UAV sensing zone.

Obstacle Function

To ensure that no UAV contacts the stationary obstacles that have been predetected in the combat zone, a scalar obstacle function is introduced to determine the relationship between the UAVs and the obstacles.

Definition 3 *The $i$ th obstacle function $\beta_i(\cdot) \in \mathbb{R},\ i = 1, 2, \cdots, n_o,$ is a function satisfying*

$$\beta_i = \begin{cases} 0 & \text{At least one UAV contacts} \\ & \text{the } i\text{th obstacle} \\ > 0 & \text{Otherwise} \end{cases} \tag{8}$$

where $n_o \in \mathbb{R}$ is the number of the obstacles.

As indicated by ((8)), if any of the UAVs approach the boundary of the $i$ th obstacle, the $i$ th obstacle function $\beta_i(\cdot)$ approaches zero, and hence, this type of design for $\beta_i(\cdot)$ spawns a gradient-like term to ensure that the UAV moves away from the $i$ th obstacle. Based on ((8)), the $i$ th obstacle function $\beta_i(\cdot)$ can be designed as follows

$$\beta_i = \prod_{j=1}^{n} \left[ \left\| q_j - q_{oi} \right\|^2 - \left( r_{zj} + r_{oi} \right)^2 \right] \tag{9}$$

where $q_{oi} \in \mathbb{R}^3, i = 1, 2, \cdots, n_o,$ denotes the position of $i$ th obstacle, and $r_{oi} \in \mathbb{R}, i = 1, 2, \cdots, n_o,$ denotes the radius of the $i$ th obstacle.

EA Detection Zone Function

To avoid being detected by the EAs, a scalar EA detection zone function is introduced to determine the relationship between the UAVs and the EAs.

Definition 4 *The EA detection zone function* $g_i(\cdot) \in \mathbb{R},\ i = 1, 2, \cdots, n_e,$ *is a function satisfying*

$$g_i = \begin{cases} 0 & \begin{array}{l}\text{At least one UAV reaches}\\ \text{the detection zone boundary}\\ \text{of the } i\text{th EA}\end{array} \\ 1 & \begin{array}{l}\text{The } i\text{th EA is outside of the}\\ \text{sensing zone of all UAVs}\end{array} \\ \in (0,1) & \text{Otherwise} \end{cases} \tag{10}$$

where $n_e \in \mathbb{R}$ is the number of the EAs.

As the boundary of the sensing zone of any of the UAVs approaches the boundary of the detection zone of any of the EAs, the value of the EA detection zone function $g_i(\cdot)$ approaches zero. Based on the definition of the bump function in ((3)) and the EA detection zone function in ((10)), a specific EA detection zone function $g_i(\cdot)$ can be designed as follows

$$g_i = \prod_{j=1}^{n} \left[ 1 - \rho_{h_{ij}} \left( \frac{h_{ij}}{r_{di} + r_{zj}} \left\| q_j - q_{ei} \right\| \right) \right] \tag{11}$$

where $q_{ei} \in \mathbb{R}^3, i = 1, 2, \cdots, n_e,$ denotes the position of $i$ th EA, and $r_{di} \in \mathbb{R}^3, i = 1, 2, \cdots, n_e,$ denotes the radius of the $i$ th EA detection zone. From ((10)) and ((11)), it is clear that: i) $g_i = 0$ when for any $j = 1, 2, \cdots, n,$ $\left\| q_j - q_{ei} \right\| \leqslant r_{di} + r_{zj},$ and ii) $g_i = 1$ when for all $j = 1, 2, \cdots, n,$ $\left\| q_j - q_{ei} \right\| \geqslant r_{sj}.$ Based on ((3)), ((10)), and ((11)), the bump function parameter $h_{ij}, i = 1, 2, \cdots, n_e, j = 1, 2, \cdots, n,$ in ((11)) can be designed as follows

$$h_{ij} = \left( r_{di} + r_{zj} \right) / r_{sj}. \tag{12}$$

Remark 1 *Because* $g_i(\cdot)$ *is a user-defined function, and* $r_{di}$ *may be somewhat uncertain, a best guess for* $r_{di}$ *can be used instead of the actual size of the EA detection zone as long as the best guest of* $r_{di}$ *is greater than the actual values.*

UAV Collision Function

To avoid collisions between UAVs, we quantify the distance between the $i$ th UAV and the $j$ th UAV by introducing the UAV collision function.

Definition 5 *The UAV collision function* $b_{ij}(\cdot) \in \mathbb{R},\ i, j = 1, 2, \cdots, n$ *and* $i \neq j,$ *is a function satisfying*

$$b_{ij} = \begin{cases} 0 & \text{The } i\text{th UAV contacts the } j\text{th UAV} \\ > 0 & \text{Otherwise.} \end{cases} \tag{13}$$

From ((13)), it is clear that when the $i$ th UAV and the $j$ th UAV do not contact each other, $b_{ij}\left(q_i(t), q_j(t)\right) > 0$. As indicated by ((13)), the function $b_{ij}(\cdot)$ approaches zero if the $i$ th UAV approaches the $j$ th UAV, and hence, this type of design for $b_{ij}(\cdot)$ spawns a gradient-like term to ensure that there is no contact between the $i$ th UAV and the $j$ th UAV. Based on the definition of UAV relationship function in ((13)), a specific UAV collision function can be designed as follows [28]

$$b_{ij} = \left\| q_j - q_i \right\|^2 - \left( r_{zi} + r_{zj} \right)^2. \tag{14}$$

Multiple UAV Cooperative Control

Multiple UAV Navigation Function

The primary objective is to navigate the multiple UAVs from an initial configuration to a constant goal configuration while remaining in the combat zone and avoiding stationary obstacles and EAs. To avoid collision with obstacles and EAs, the UAV configuration $q(t)$ should remain in

a free configuration space denoted by $\mathcal{D} \subset \mathbb{R}^{3n}$ which is a subset of the whole combat zone with all configurations removed that involve collision with obstacles and EAs. The constant initial and goal configuration are assumed to be in the interior of $\mathcal{D}$. To generate $q(t) \in \mathcal{D}$, the special artificial potential function coined a navigation function in [13], can be used. Specifically, the navigation function used in this paper is defined as follows [20].

Definition 6 *Let $\mathcal{D}$ be a compact connected analytic manifold with a boundary, and let $q^*$ be a goal point in the interior of $\mathcal{D}$. A mapping $\varphi(q): \mathcal{D} \to$ [0, 1], is a navigation function if:*

- It is analytic on $\mathcal{D}$ (at least the first and second partial derivatives exist and are bounded on $\mathcal{D}$);
- It has a unique minimum at $q^*$;
- It obtains a maximum value on the boundary of $\mathcal{D}$ (i.e., admissible on $\mathcal{D}$);
- It is a Morse function (i.e., the matrix of second partial derivatives, the Hessian, evaluated at its critical points is non-singular).

After defining the obstacle functions, the UAV collision functions, the boundary functions, and the EA detection zone function, we now introduce the navigation function that is utilized to derive the cooperative control algorithm. The navigation function $\varphi(q) \in \mathbb{R}$ for multiple UAVs is constructed as follows

$$\varphi(q) = \frac{\left\| q - q^* \right\|^2}{\left[ \left\| q - q^* \right\|^{2\kappa} + G(q) \right]^{1/\kappa}} \tag{15}$$

where $\kappa \in \mathbb{R}_+$ is a positive constant parameter, $G(q) \triangleq G_1 G_2 G_3 \in \mathbb{R}$, and the scalar functions $G_1(\cdot), G_2(\cdot), G_3(\cdot) \in \mathbb{R}$ are defined as follows

$$\begin{aligned} G_1 &= \prod_{i=0}^{n_0} \beta_i(q) \quad G_2 = \prod_{i=1}^{n_e} g_i(q) \\ G_3 &= \sqrt{\prod_{i=1}^{n} \prod_{j \neq i} b_{ij}(q)} \end{aligned} \tag{16}$$

where $\beta_i(\cdot)$, $g_i(\cdot)$, and $b_{ij}(\cdot)$ were defined in ((6)), ((9)), ((11)), and ((14)). Based on ((16)) and the definition of $G(\cdot)$, we can easily show that $G = 0$ will occur:

- if $G_1 = 0$ (when any UAV contacts the boundary of the combat zone or the obstacles)
- if $G_2 = 0$ (when any UAV contacts the detection zone boundary of a EA)
- if $G_3 = 0$ (when two or more UAVs contact).

Remark 2 *Indeed, for a typical obstacle avoidance problem, it does not seem possible to construct* $\varphi(q(t))$*, coined a navigation function, such that* $\frac{\partial}{\partial q}\varphi(q(t)) = 0$ *only at* $q(t) = q^*$*. Koditschek* [14] *has shown that strict global navigation is not possible. That is, as discussed in* [20]*, the appearance of interior saddle points (i.e., unstable equilibria) seems to be unavoidable; however, these unstable equilibria do not really cause any difficulty in practice. As proved in* [14]*, the function* $\varphi(\cdot)$ *is a navigation function, so long as the parameter* $\kappa$ *exceeds a certain function of the geometric data. That is,* $\varphi(q(t))$ *can be constructed such that the attraction domain of the unstable equilibria is a set of measure zero.*

Remark 3 *The boundary of* $\mathcal{D}$ *includes the boundary of the combat zone, the boundary of the stationary obstacles, and the detection zone boundary of the EAs.*

Remark 4 *The main difference between the navigation function in (*(15)*) and the standard navigation function in* [20] *is that for the navigation function in (*(15)*), the boundary of the combat zone and the EAs have only local interaction with the UAVs. When the boundary of the combat zone and EAs are out of the sensing zone of the UAVs, the corresponding boundary function and EA detection zone function become unity. When the boundary of the combat zone contacts at least one UAV and the detection zone boundary of the EA contacts at least one UAV, the boundary function and corresponding EA detection zone function become zero. It is the analytical smoothing bump function that allows the functions given by (*(6)*) and (*(11)*) to be appropriately smoothed out among all cases to facilitate the gradient-like differentiation for multiple UAV navigation. With a similar structure as compared to the standard navigation function given in* [14]*, it is not difficult to follow the proof in* [14] *to show the multiple UAV*

*navigation function defined in (*(15)*) satisfies the navigation function properties defined in* [14], [20].

## Cooperative Control Design

The cooperative control in this paper is specified as all UAVs act based on all other UAVs' position to avoid collision with other UAVs, obstacles, and EAs. Based on the above definitions, the input $u(t)$ in ((1)) is designed as follows

$$u = -K\left(\frac{\partial \varphi}{\partial q}\right)^T \tag{17}$$

where $K \in \mathbb{R}^{3n\times 3n}$ is a constant positive definite symmetric control gain matrix and $\frac{\partial \varphi}{\partial q}(q) \in \mathbb{R}^{1\times 3n}$ is the partial derivative of $\varphi(\cdot)$ from ((15)), along $q(t)$. The control algorithm designed in ((17)) illustrates this cooperative control concept since the multiple UAV navigation function $\varphi(q)$ is a function of all of the UAVs' position.

Remark 5 *The control* $u(t)$ *in (*(17)*) has been based on the kinematic model in (*(1)*). This control can be extended using standard backstepping techniques to include second-order dynamic robot models.*

Remark 6 *With the definition of the EA zone detection function, the navigation function* $\varphi(q)$ *is a function of* $i$ *th EA position only when the* $i$ *th EA is within the sensing zone of UAVs. Therefore, the control designed in (*(17)*) depends on the position of the* $i$ *th EA only when the* $i$ *th EA is within the sensing zone of UAVs in which case the position of the* $i$ *th EA is known. This is one of the novelties of the proposed control algorithm.*

## Stability Analysis

Theorem 1 *Provided* $q(0) \in \mathcal{D}$*, the cooperative control designed in (*(17)*) along with the navigation function* $\varphi(q)$ *designed in (*(15)*) ensures that* $q(t) \in \mathcal{D}$ *and asymptotic navigation in the sense that*

$$q(t) \to q^* \text{ as } t \to \infty. \tag{18}$$

Proof: Let $V(q): \mathcal{D} \to \mathbb{R}$ denote the following non-negative function

$$V(q) = \varphi(q). \tag{19}$$

After taking the time derivative of ((19)), the following expression can be obtained

$$\dot{V} = \frac{\partial \varphi}{\partial q}\dot{q}. \tag{20}$$

After substituting ((17)) into ((20)), the following expression is obtained

$$\dot{V} = -f(t) \tag{21}$$

where $f(t) \in \mathbb{R}$ denotes the following non-negative function

$$f(t) \triangleq \frac{\partial \varphi}{\partial q} K \left( \frac{\partial \varphi}{\partial q} \right)^T. \tag{22}$$

Based on ((21)) and ((22)), it is clear that $V(q(t))$ is a non-increasing function in the sense that

$$V(q(t)) \leq V(q(0)). \tag{23}$$

From ((23)), it is clear that for any initial condition $q(0) \in \mathcal{D}$, that $q(t) \in \mathcal{D} \quad \forall t > 0$; therefore, $\frac{\partial \varphi}{\partial q}, \frac{\partial^2 \varphi}{\partial q^2} \in \mathcal{L}_\infty$ based on the Property 1 of Definition 6. Then, it is clear from ((1)) and ((22)) that $\dot{f}(t) \in \mathcal{L}_\infty$. Based on ((19)), ((21)), ((22)), and the fact that $\dot{f}(t) \in \mathcal{L}_\infty$ on $\mathcal{D}$, then Lemma A.6 of [7] can be invoked to prove that

$$\left\| \frac{\partial \varphi}{\partial q} \right\| \to 0 \tag{24}$$

in the region $\mathcal{D}$. Based on Definition 6 and Remark 2, it can be determined that if $\frac{\partial \varphi}{\partial q}(q(t)) \to 0$ then $q(t) \to q^*$.

Remark 7 *Based on Assumption 7 and ((9)), ((11)), ((14)), and ((16)),* $G_1(q(0))$, $G_2(q(0))$, $G_3(q(0)) > 0$. *Therefore* $G(q(0)) > 0$. *Since* $G(q(0)) > 0$ *and* $V(q(t)) \le V(q(0))$ *for all time, then it is clear from ((15)) that* $G(q(t)) > 0$. *Then* $G_1(q(t)) \neq 0, G_2(q(t)) \neq 0, G_3(q(t)) \neq 0$. *Therefore* $G_1(q(t)), G_2(q(t)), G_3(q(t)) > 0$. *We can now see that if* $G(q(t)) > 0$ *for all time and* $q(t) \to q^*$ *as* $t \to \infty$, *then all the objectives given in the Problem Formulation Section will be achieved.*

Moving EA Extension

As explained in the previous sections, the above navigation problem involves moving multiple UAVs from an initial configuration to a constant goal configuration while avoiding stationary obstacles and EAs. To address the moving EA problem, we now discuss, in a heuristic manner, how the approach can still be expected to provide reasonable performance with regard to cooperative control.

A composite configuration of the moving EAs, denoted by $q_e(t) \in \mathbb{R}^{3n_e}$, is defined as follows

$$q_e(t) = \begin{bmatrix} q_{e1}^T(t) & q_{e2}^T(t) & \cdots & q_{en_e}^T(t) \end{bmatrix}^T \tag{25}$$

Similar to the navigation function $\varphi(q)$ designed in ((15)), a navigation-like function $\varphi_1(q, q_e): \mathcal{D} \times \mathcal{D} \to$ [0, 1] is constructed as follows

$$\varphi_1(q, q_e) = \frac{\|q - q^*\|^2}{\left[\|q - q^*\|^{2\kappa} + G(q, q_e)\right]^{1/\kappa}} \tag{26}$$

where $G(q, q_e) \triangleq G_1 G_2 G_3 \in \mathbb{R}$, and the scalar functions $G_1(\cdot), G_3(\cdot) \in \mathbb{R}$ were defined in ((16)), and $G_2(\cdot) \in \mathbb{R}$ are defined as follows

$$G_2 = \prod_{i=1}^{n_e} g_i(q, q_e) \tag{27}$$

In ((27)), $g_i(\cdot)$ were defined in ((11)). To facilitate the subsequent discussion, the following assumption must be satisfied:

Assumption 8 The velocity of the EA must satisfy $\left\| \frac{\partial \varphi_1}{\partial q_e} \dot{q}_e \right\| \leqslant \varepsilon$ where $\varepsilon \in \mathbb{R}$ is a positive constant.

Similar to the input $u(t)$ designed in ((17)) for stationary EA case, the input $u(t)$ in ((1)) for moving EA case is designed as follows

$$u = -K_1 \left( \frac{\partial \varphi_1}{\partial q} \right)^T \tag{28}$$

where $K_1 \in \mathbb{R}^{3n \times 3n}$ is a constant positive definite symmetric control gain matrix.

To discuss the stability and convergence of multiple UAVs system for the moving EA case, a non-negative function $V_1(q, q_e) : \mathcal{D} \times \mathcal{D} \to [0, 1]$ is defined as follows

$$V_1(q, q_e) = \varphi_1(q, q_e). \tag{29}$$

After taking the time derivative of ((29)), the following expression can be obtained

$$\dot{V}_1 = \frac{\partial \varphi_1}{\partial q} \dot{q} + \frac{\partial \varphi_1}{\partial q_e} \dot{q}_e. \tag{30}$$

Based on ((1)), ((28)), and Assumption 8, $\dot{V}_1(t)$ in ((30)) can be upper bounded as follows

$$\dot{V}_1 \leqslant -k \left\| \frac{\partial \varphi_1}{\partial q} \right\|^2 + \varepsilon. \tag{31}$$

where $k \triangleq \lambda_{\min}(K_1)$, and $\lambda_{\min}(\cdot)$ denotes the minimum eigenvalue. We now discuss two cases.

Case 1 $q(t)$ is near the boundary of $\mathcal{D}$.

Since we are near the boundary, it seems reasonable to assume that $\left\| \frac{\partial \varphi_1}{\partial q} \right\| \geqslant \varepsilon$, where $\varepsilon \in \mathbb{R}$ is a positive constant. Hence, based on ((31)), $k$ can be selected to ensure that $\dot{V}_1(t)$ be non-positive when $q(t)$ is near the boundary of $\mathcal{D}$. Similar to Property 3 in Definition 6, it is clear from ((26)) that the navigation-like function $\varphi_1(q, q_e)$ obtains the maximum value on the boundary of $\mathcal{D}$. Since $\varphi_1(t)$ is a non-increasing function, and $\varphi_1(t) < 1$ for all time when $q(t)$ is near the boundary of $\mathcal{D}$, the UAVs will never contact the boundary of $\mathcal{D}$. It is also clear that $G_i(t) > 0,$ for $i = 1, 2, 3$ when $q(t)$ is near the boundary of $\mathcal{D}$. Hence, Objective 1 - Objective 4 will be achieved for this case.

Case 2: $q(t)$ is far away from the boundary of $\mathcal{D}$.

For this case, the second term in ((30)) will vanish based on the property of the bump function in ((5) ). Hence, an argument identical to the proof of Theorem 1 can now be applied to show that Objective 1 - Objective 5 will be achieved.

# CHAPTER THREE

# SIMULATION OF MULTIPLE UAV SYSTEM

Simulations were carried out on both the stationary EA system and the moving EA extension. The UAVs were configured in an arbitrary initial configuration from where they navigated to the final configuration while fulfiling all objectives of the navigation function. None of the UAVs touched the others, stationary obstacles were avoided, the UAVs stayed out of the detection zones of the EAs, and the UAVs stayed within the combat zone at all times.

The system was found to be sensitive to the values of the tunable parameters, namely $\kappa$ ( from (16)) and K (from (18)). Successful convergence to the goal configuration was highly dependant on the gains assigned to the system using tunable parameters. The gains required re-tuning when EAs or stationary obstacles were added to the combat zone, and when their properties such as physical radius or radius of motion were changed.

## Overview of Simulation Model

The system was simulated using MATLAB Simulink. Simulations were in the Simulink 6 format. For the final version of the simulations, the moving EA extension simulation template was modified to create the stationary EA simulation. The reason for this is apparent when the moving EA simulation is examined.

A circle was used as the trajectory for EA motion. The radius of the circle was varied for each EA. A S-Function was written to incorporate EA motion into the program. The S-Function generated position values for each of the EAs and they were passed to the S-Function responsible for calculating UAV positions based on the navigation function. For the stationary EA case, the EAs were set to retain their original position by setting their trajectory to a circle of radius zero.

Layout of the Simulation model

The main model template can be seen in Figure (D.1). In the model, four kinds of blocks can be seen : Subsystems, S-Functions, Data Files and Displays. They have been represented by orange, blue, gray and green blocks respectively, in the figure.

There are three multiplexers in the model. The multiplexers may be viewed as demarcation points, which divide the simulation can be divided into three major subsystems. EA Subsystems define the location, motion and properties of the EAs in the combat zone. Obstacle Subsystems combine the EA Subsystem with stationary, non-EA obstacles from the environment. Navigation Subsystems use information from Obstacle Subsystems, current position of the UAVs and desired Goal Configuration in the Navigation Function.

Each of the major subsystems mentioned above comprises of smaller subsystem blocks, which are seen in the model. Their nomenclature in the model explains their relevance to the algorithm or the nature of the data provided by them.

Desired Goal Configuration, Current EA Position and Current UAV position are written to separate data files. These data files log these positions in a matrix for each time instant in the simulation. This data can be used to create a movie file which animates the simulation. It is also possible to view the position values being logged by attaching a Displays to appropriate points. Displays were otherwise used as a diagnostic tool, for observing the variation in certain values which reflected the performance of the system.

All the algorithmic input was used by a single S-Function, called fiveBotsFourObs.m. This function performed all the mathematical calculations necessary to obtain the control input $\boldsymbol{u}$, which was the derivative of the UAV position. This was integrated and fed back to the same function. The control loop also included other, non-feedback inputs such as Goal Configuration, EA Position and value of gains from the Tunable Parameters. There were a number of long differentiation operations required for the math in the simulation. All of these operations were performed using Maple and these Maple outputs were used in the S-function .

## Simulation Results

To illustrate the performance of the proposed cooperative control algorithm, we set up a 2-dimensional simulation to navigate five UAVs from an initial configuration to the goal configuration while avoiding four obstacles and avoiding being detected by three EAs. Whenever there are multiple obstacles along the trajectory between the initial and final points, which in this case consists of stationary EAs and obstacles, the UAVs cannot pass through the obstacles. The chosen initial configurations constitute non-trivial setups since the straight paths connecting the initial and final positions are obstructed by the stationary EAs and obstacles. As a result, the UAVs might split along the straight line connecting the two points in two or smaller groups while avoiding the stationary EAs and obstacles.

The initial configuration of the five UAVs was selected as follows:

$$q_1(0) \triangleq [0 \quad -20]^T \quad q_2(0) \triangleq [3 \quad -20]^T \quad q_3(0) \triangleq [-3 \quad -20]^T$$
$$q_4(0) \triangleq [-3 \quad -17]^T \quad q_5(0) \triangleq [3 \quad -17]^T .$$

Their destination configuration was selected as follows:

$$q_1^* \triangleq [0 \quad 30]^T \quad q_2^* \triangleq [3 \quad 27]^T \quad q_3^* \triangleq [-3 \quad 27]^T$$
$$q_4^* \triangleq [-3 \quad 22]^T \quad q_5^* \triangleq [3 \quad 22]^T .$$

The stationary obstacles were positioned in the combat zone at

$$q_{o1} \triangleq [-10 \quad -5]^T \quad q_{o2} \triangleq [2 \quad 12]^T$$
$$q_{o3} \triangleq [-20 \quad 10]^T \quad q_{o4} \triangleq [10 \quad 0]^T .$$

Three stationary EAs were located at

$$q_{e1} \triangleq [-7 \quad 2.5]^T \quad q_{e2} \triangleq [15 \quad 13]^T \quad q_{e3} \triangleq [0 \quad -10]^T .$$

The radii of the UAVs were selected as follows

$$r_{z1} = 1 \quad r_{z2} = 1 \quad r_{z3} = 1 \quad r_{z4} = 1 \quad r_{z5} = 1.$$

The radii of the combat zone and the obstacles were selected as follows

$$r_{o0} = 35 \quad r_{o1} = 2 \quad r_{o2} = 3 \quad r_{o3} = 3 \quad r_{o4} = 1.$$

The radii of the UAV sensing zones were selected as follows

$$r_{s1} = 8 \quad r_{s2} = 8 \quad r_{s3} = 8 \quad r_{s4} = 8 \quad r_{s5} = 8.$$

The radii of the EA detection zone were selected as follows

$$r_{d1} = 4 \quad r_{d2} = 4 \quad r_{d3} = 4.$$

Figure (3.1) - Figure (3.5) show several stages of multiple UAVs from the initial configuration to the goal configuration. The gray circles illustrate the detection zone of the EAs, the black circles illustrate the obstacles, and the UAVs are denoted by the circles with the cross in the center. The five crosses at the top part of the figure illustrate the goal configuration of the UAVs. From the figures, it is clear that the UAVs moved to their destination while avoiding the stationary obstacles and avoiding being detected by the EAs along their paths.

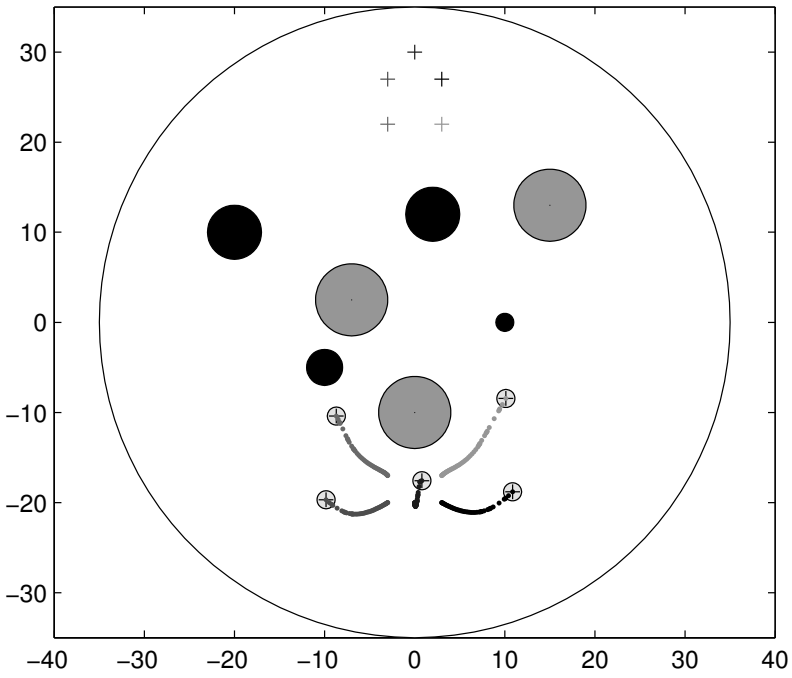


Figure 3.1 UAV Navigation - First Stage

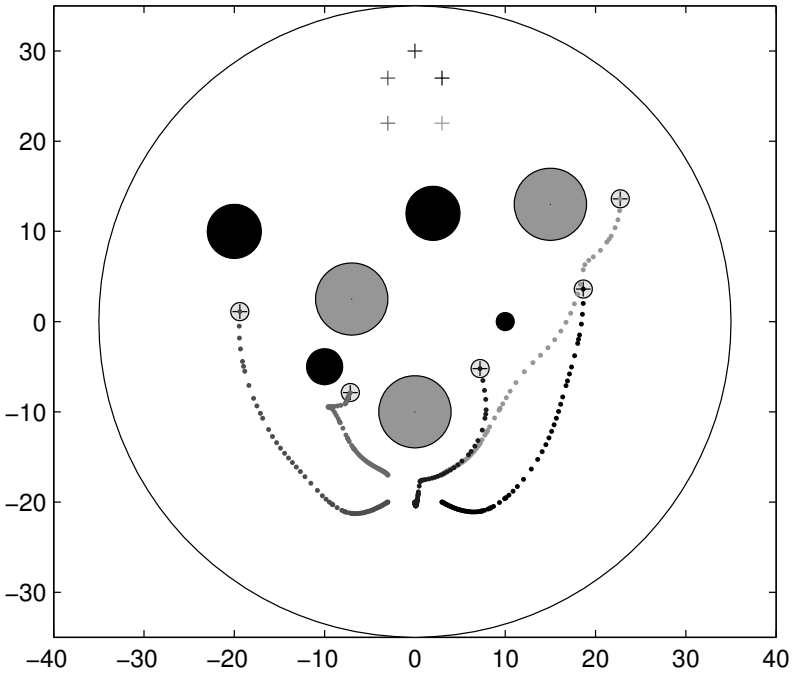


Figure 3.2 UAV Navigation - Second Stage

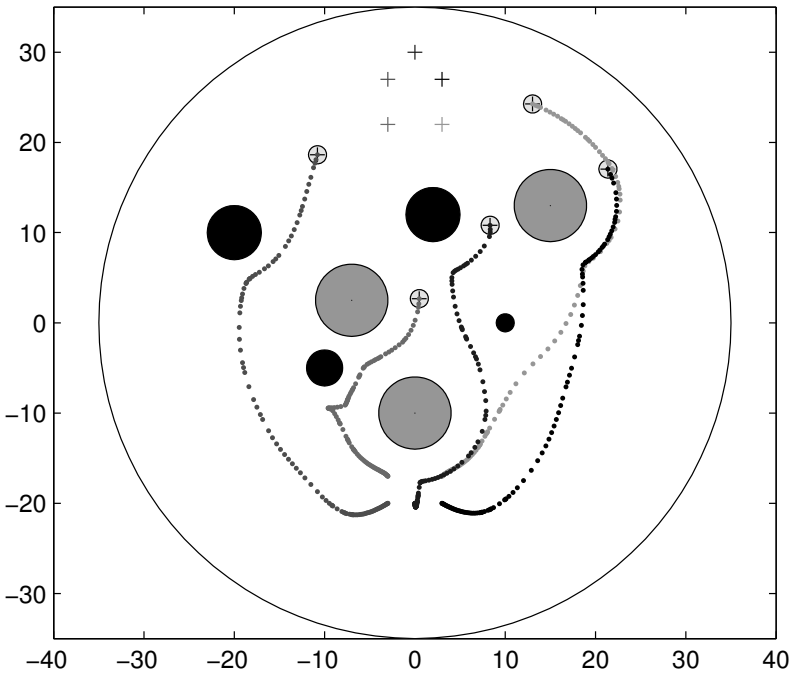


Figure 3.3 UAV Navigation - Third Stage

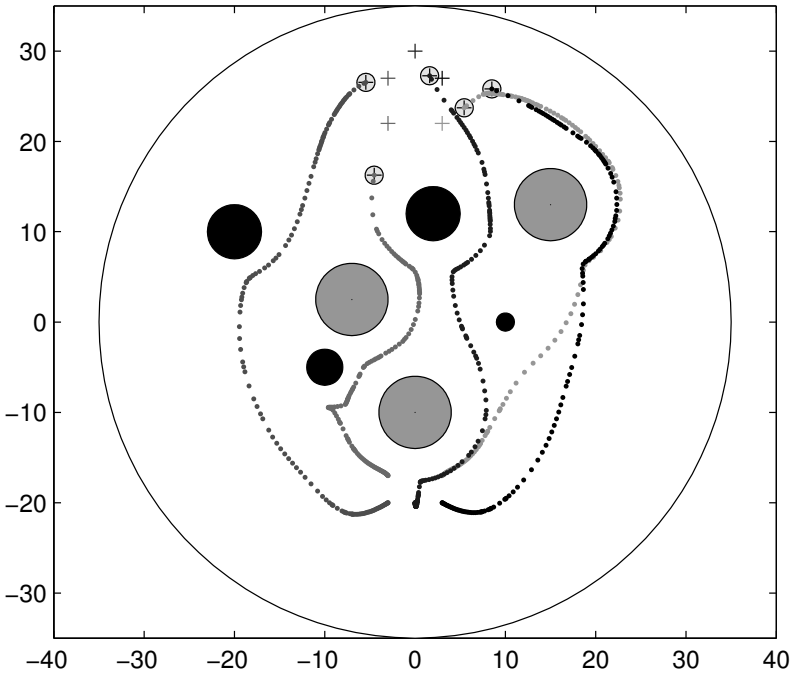


Figure 3.4 UAV Navigation - Fourth Stage

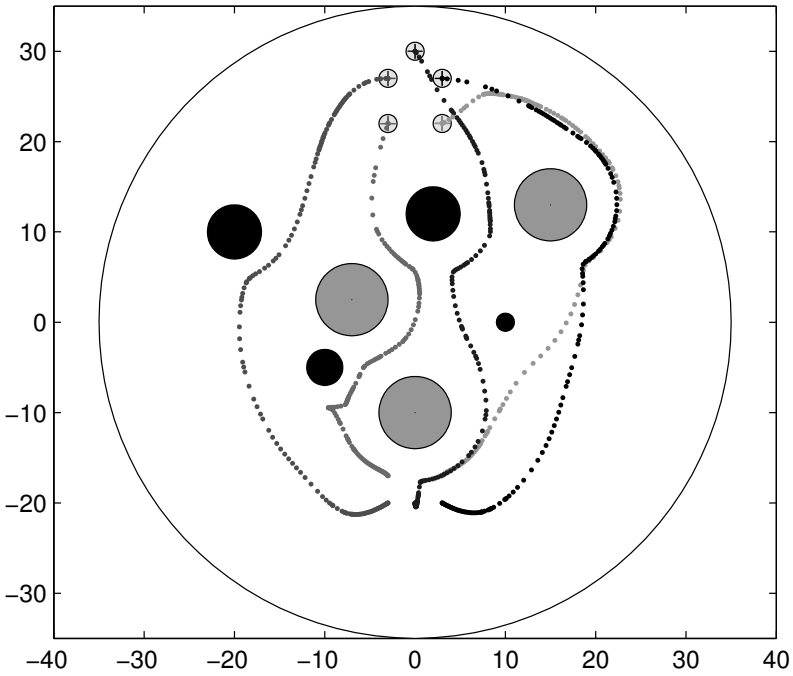


Figure 3.5 UAV Navigation - Final Configuration

## Generating a Simulation Movie File

The simulation data collected by .mat files was used in generation of .avi files which animated the simulation results. Movie generation in MATLAB is essentially a plot generating program which includes additional movie conversion commands. While the UAVs and EAs were drawn using standard MATLAB plot commands, their positions were taken from the data files from the simulation. The plot program, and hence the movie, left a trace of the path of each UAV as it moved from source to destination. MATLAB provides a combination of movie generation functions which create an animation based on information from data files.

The downloadable video clips for multiple UAV navigation for both stationary EAs and moving EAs are provided in [5].

## Conclusion

A navigation function based cooperative control is developed in this paper to navigate multiple UAVs in an obstacle and enemy asset cluttered environment. The standard navigation function approach was extended to a multiple navigation strategy with analytical switching among different cases due to the limited sensing zone of the UAVs. A differentiable cooperative control law is proposed based on this navigation function that yields asymptotic convergence. The navigation strategy of avoiding the moving EAs is also discussed. Simulation results are provided to demonstrate the performance of the proposed control strategy.

# CHAPTER FOUR

# DEVELOPMENT OF DEVICE DRIVER INTERFACE FOR Q4 DATA ACQUISITION CARD

## Project Summary

### Development Objectives

The objective of this project was the development of a device driver for the Quanser Q4 Measurement and Control Board for the QNX Operating System. The Quanser Q4 offers an interface to a wide variety of devices such as Analog and Digital sensors and Encoders. The development of this low level software was necessitated by two primary motivators

- The need to migrate to a PCI device for measurement and data acquisition. The data acquisition card currently in use is the ServoToGo Board, which is an ISA Board. This required even recently purchased motherboards to come equipped with ISA slots. The ISA architecture being increasingly obsolete, a switch to a measurement and control system running on the commonly used PCI bus was desired. The Q4 is such a device.
- The need for a QNX driver for the Q4, QNX being the operating system of choice in the Control and Robotics Group. The Quanser Q4 had been successfully tested on the Windows platform using standard drivers and software developed by Quanser Consulting Inc®.

### Design Considerations

It was desired to have a driver which

- Provided access to all features of the Quanser Q4.
- Was economical in its use of system resources
- Provided an easy interface for user to configure the operating mode of the Quanser Q4.

Support for achieving the above objectives would be provided by

- The functionalities provided by the operating system for a software developer. These would determine the fluidity possible during the development process.
- The technical support provided in the Q4 Manual in terms of device details necessary for implementing a driver.

It was observed that QNX offers a wide range of functions related to PCI Device communication and programming [31]. Further, the microkernel architecture of QNX allows the programmer to work at the user level even when developing low level software such as a device driver. This made it possible to debug and test the functioning of the device while still employing high level debugging techniques such as printing out information on the Terminal window. QNX Help provided exhaustive information on each of the QNX system calls and other relevant functions.

The Q4 itself has been well documented in the Q4 Manual. A section on the Quanser Q4 Registers has been included in the manual for the benefit of programmers working towards device driver development. All of the interaction with the Q4 internal to the computer can be defined in terms of operations on Q4 Registers, and Interrupt Servicing.

Working successfully with the card depended on being able to perform operations on card registers, and servicing interrupt requests coming from the card. Register level operations were assisted by provisions in QNX for device memory mapping. Interrupt Servicing was made possible by a number of interrupt related commands and system calls provided by the QNX OS.

Decisions on the software development process were made depending on matching the design objectives with the available tools for fulfilling those objectives. C was used as the preferred programming language for software testing in the initial phase. Later, it became possible to modularize the code using C++. The Class implemented using C++ was the culmination of the programming effort.

## Overview of the Project Implementation

The project could be considered to have three significant phases of development

- Achieving Interrupt Service and PCI Communication
- Testing Card Features using standalone C Programs
- Implementation and Testing of the Driver Class

In the first phase lay the testing of operations related to communicating with the card, which was to form the basis of all future development.

Real Time Clock Interrupt programs were used for understanding how interrupt driven software is written for QNX. The importance of the Real Time Clock interrupt was that it was a system interrupt assigned to a fixed interrupt line for x86 systems, and that the RTC was a periodic interrupt source. Failure to acknowledge or service the RTC Interrupt was thus a sure indicator of a flaw in either the logic or the implementation of the interrupt service mechanism. PCI commands in QNX were used to talk to the device and memory map the device to provide access to Registers. Memory mapping is used to provide user programs with direct access to device memory. The result of the mapping is an access to the Read/Write Registers of the Q4. The completion of Phase I meant that the card could be communicated with, successfully. Card Features were tested in the next phase following methods validated by Phase I.

Given the versatility of the Q4, testing each feature meant writing values to a different set of Registers, or writing different values to the same set of Registers. There were a number of such features to be tested, and they demanded slightly or vastly different testing methods. To simplify this process, each significant feature was tested using a different program. Register configurations were hard-coded into the program, and results were checked using either software methods or by using digital multimeters and wires.

Once the card features had been tested in Phase II, it was necessary to modularize the software to allow the user to define the configuration of the card. This was the phase in which the Driver Class was implemented.

A look at the Quanser Q4 Register Map [30] reveals that the operations that can be employed on the card can be divided into six major categories; Interrupt Service, ADC, DAC, Counter/Timer, Digital IO and Encoder functions. To complete the interface, one adds the Constructor and Destructor which set up the Card and renders it ready for use for one of the above set of functions. In addition, it was observed that some of the functions in a category needed to be called in a particular sequence to correctly set up the system. Moreover, functions across categories could be used in combination to realize a feature of the Q4. Finally, it was essential to let the user customize the setup and keep very little hard-coded control over the state of the system.

All of these factors were considered in the final implementation for this project, which was the class QuanserQ4. The outline of this project implementation is given in Figure (4.1).

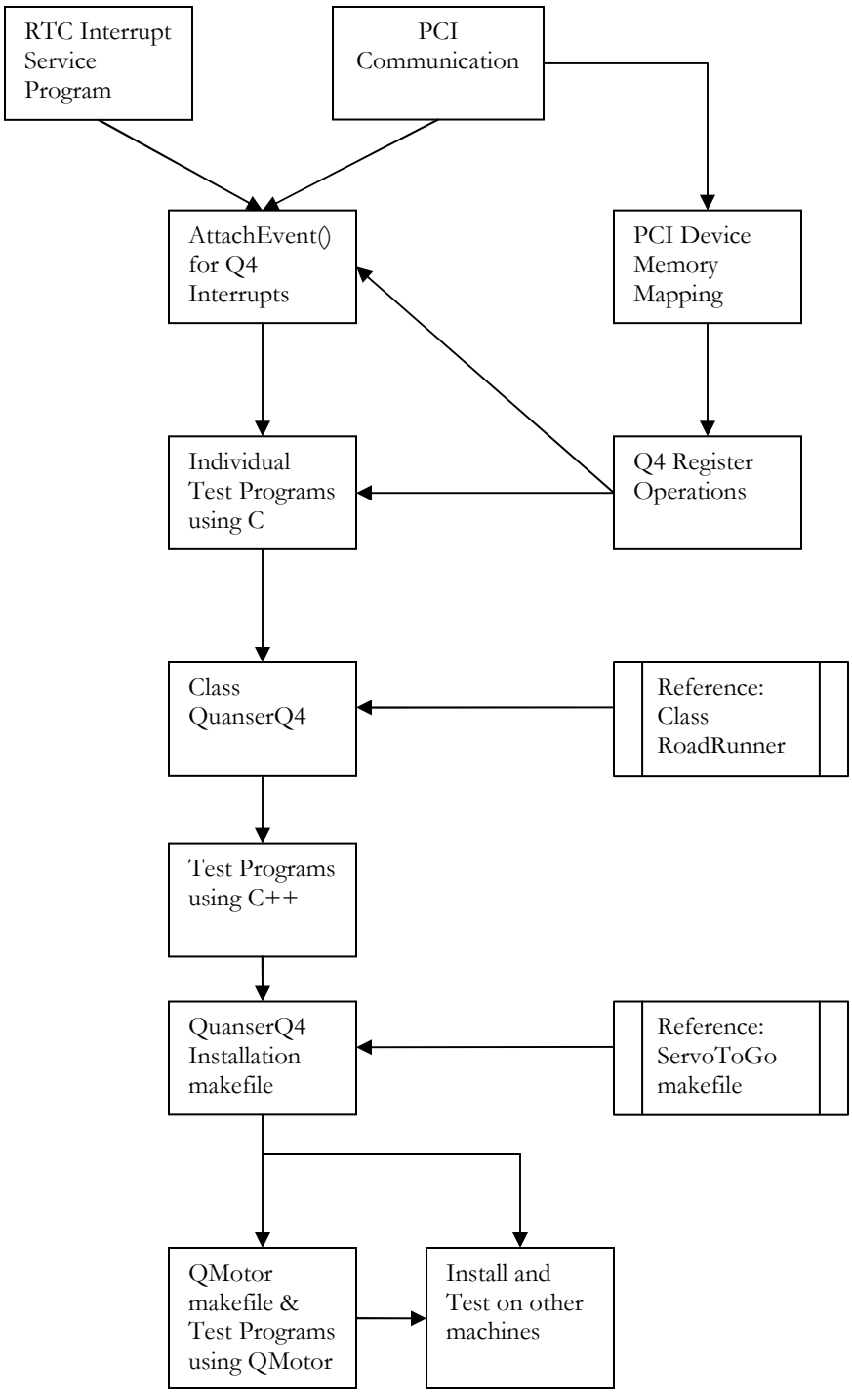


Figure 4.1 Device Driver Project - Implementation Flowchart

# PCI Communication in QNX

## About Device Initialization and Memory Mapping

QNX provides a set of functions which simplify communicating with a PCI device. They can broadly be classified into Device Initialization and Memory Mapping functions.

A card on the PCI bus needs to be initialized before any operations are performed on it. The following is done during Device Initialization

- Check the presence and operation of the PCI Bus,
- Locate the Card on the PCI Bus, and
- Associate a QNX structure called pci_dev_info, which stores PCI device information, with the card.

Associating the software structure with the device allocates an Interrupt Request line to the Card, and Interrupt Servicing methods can be used from that point onwards. Interrupt Servicing is necessary for complete testing for Card Features. To setup the device for Interrupt Servicing and memory read/write operations, device memory needs to be mapped to the system. The procedure for memory mapping is

- Obtain the device Base Address from the initialized pci_dev_info structure
- Verify that the Address is a Memory Address
- Use the Base Address to map device memory to system memory

All features of the Q4 can be used in the Q4 Driver Class by reading from, and writing to, the Q4 Card Registers. Once the device has been initialized and memory mapping has taken place, the features of the Q4 are accessible to the application.

## Algorithm for Implementing PCI Communication

The following sequence of steps are taken to communicate with the Q4 PCI device on the system

- Attach the PCI server

  ```
  pci_attach( )
  ```

- Check for the presence of the PCI BIOS

  ```
  pci_present( )
  ```

- Find the PCI device

  ```
  pci-v
  ```

  ```
  pci_find_device( )
  ```

- Attach a driver to the PCI device

  ```
  pci_attach_device( )
  ```

  ```
  pci_dev_info
  ```

- Map PCI device memory

  Extract CPU Base Address

  ```
  Q4info.CpuBaseAddress[ ]
  ```

  ```
  PCI_IS_MEM( )
  ```

  Map device memory

  ```
  mmap_device_memory( )
  ```

  Use of memory map to address registers (Memory map + offsets)

## Details of the Algorithm

### *Attach the PCI server*

The pci_attach( ) function connects to the Peripheral Component Interconnect (PCI) server. pci_attach( ) has to be called before calling any of the other PCI functions.

pci_attach( ) is used in the programs as follows

```
if (pci_attach(0) != -1 ) ...    //  If  no  error,  program
continues
```

*Check for the presence of the PCI BIOS*

The pci_present( ) function has been used to determine whether or not the PCI BIOS interface function set is present.

When no flags are defined, the function checks only for the presence of the PCI BIOS. This function has been used in Q4 Programs as follows

```
if(pci_present(NULL,NULL,NULL)==PCI_SUCCESS) ...
// Program continues
```

*Find the PCI device*

The QNX System can search for a PCI device when given identification numbers, called Device ID and Vendor ID. The function used for this search is pci_find_device( ). It returns the location of the nth PCI device that has the specified Device ID and Vendor ID.

Device and Vendor ID information for the Q4 are necessary for this and other commands. These may be obtained by either using the Q4 Manual or the QNX Terminal window listing of all PCI devices (using pci -v) on the system's PCI bus. The Q4 Manual lists device information. Using the Manual, the Device and Vendor ID obtained is as follows

```
Q4_DEVICE_ID 0x0010
Q4_VENDOR_ID 0x11e3
```

The pci_find_device( ) command can be excluded from test programs for a single Q4 Card system without a loss of functionality. However, it can be used as a diagnostic to confirm the presence of a PCI Card when pci_attach_device( ) has not been not fully tested. Also, it can be used find multiple PCI Cards of the same Device and Vendor ID. This is done by making successive calls

to this function, starting with an index of 0, and incrementing it until PCI_DEVICE_NOT_FOUND is returned.

*Attach a driver to the PCI device*

```
pci_attach_device( )
pci_dev_info
memset( )
```

At this point, the pci server has been attached, the presence of the PCI BIOS has been verified, and the device has been found on the PCI system. The next major step is to attach a driver to the device.

The command pci_attach_device( ) is used for attaching a driver to the card. The command uses information from some user-definable members of the structure pci_dev_info to initialize other members of the same structure. This device information struct is called pci_dev_info.

pci_dev_info : The pci_dev_info structure has input and output members. Input Member values can be assigned to the structure by the user. One can call pci_attach_device( ) and specify which of the Output Members are to be initialized by the command. pci_attach_device( ) uses specified Input Members to identify the device and carry out this assignment.

Input members of the structure are Device ID (DeviceID), Vendor ID (VendorID) and Class Code (Class). Output members of interest are Interrupt Number (Irq), CPU Base Address (a six member array called CpuBaseAddress[ ]), and PCI ROM Address (PciRom).

pci_attach_device( ) : Typically drivers use this function to attach themselves to a PCI device, so that other drivers can't attach to the same device. If the PCI_SHARE flag is specified, then multiple drivers can attach to the same device.

The PCI server can scan based on a class code, vendor/device ID, or bus number and device/function number. To control the server scanning, appropriate fields of the info structure are initialized and appropriate flags are set.

When first attached to a non-initialized device, the PCI server assigns all the I/O ports, memory and IRQs required for the device. IRQ routing is also done. Once this has completed successfully, these values are filled to the pci_dev_info structure for use by the application.

The device can be detached by passing its handle to pci_detach_device( ). This frees any resources used by pci_attach_device( ). Usage of pci_dev_info and pci_attach_device( ) in test programs is as follows

```
attachDevice = pci_attach_device(NULL, PCI_SHARE |
PCI_SEARCH_VENDEV | PCI_INIT_IRQ | PCI_INIT_BASE0 |
PCI_INIT_BASE1 | PCI_INIT_BASE2, 0, &Q4info);
// PCI_SEARCH_VENDEV : Scan device based on Vendor ID and
Device ID
// PCI_INIT_IRQ, PCI_BASEx : Initialize IRQ, Base Address
// &Q4info : Pointer to the device pci_dev_info structure
```

Following the assignment of the Base Address values in the pci_dev_info structure, the device memory can be mapped, as detailed in the following section. The user software can then directly access and manipulate the PCI Card.

*Map PCI device memory*

CPU Base Address

```
Q4info.CpuBaseAddress[ ]
PCI_IS_MEM( )
PCI_IS_IO( )
```

Map device memory

```
mmap_device_memory( )
```

Use of memory map to address registers (Memory map + offsets)

Memory mapping is used to provide application programs with direct access to device memory [33]. The result of the mapping is access to the Read/Write Registers of the Q4, which give the user control over the Data Acquisition and Control features. This access, for a PCI Device, can be provided either by I/O commands, such as in( ) and out( ), or by regular memory operations employing pointers to gain access to device memory [34]. I/O access is achieved by mapping the device memory using a mmap_device_io( ) command, whereas, if the device lends itself to regular memory memory operations, mmap_device_memory( ) is used.

The application does not need to know, in advance, how a PCI device may be mapped. The pci_dev_info structure is used for extracting this information.

*CPU Base Address*: pci_attach_device( ) writes to the struct pci_dev_info, a member known as the CpuBaseAddress[ ]. A CPU Base Addresses is the first physical memory address for the device. The struct member CpuBaseAddress[ ] is a six element array of type uint64_t (64 bit unsigned integer). Since PCI Cards such as the Q4 have just one CPU Base Address, only CpuBaseAddress[0] is written to, the rest of the array elements retain their initialization value of zero. Other devices such as graphics cards and multi-head ethernet cards may use more than one Base Address.

To know whether IO or Memory mapping of device is to be carried out, the initialized, non zero Base Address (CpuBaseAddress[0]) is considered. It is used in macros called PCI_IS_MEM( ) and PCI_IS_IO( ). The PCI_IS_MEM( ) macro returns a TRUE value if the address given to it is a memory address. If this is the case, then mmap_device_memory( ) is used to map the device memory. PCI_IS_IO( ) returns TRUE if the address given to it is an IO address. mmap_device_io( ) is used in this case. For the Q4, PCI_IS_MEM(CpuBaseAddress[0]) holds true, and mmap_device_memory( ) is used.

*mmap_device_memory( )*: mmap_device_memory( ) maps user defined *n* bytes of a device's physical memory address into the caller's address space at the location returned by mmap_device_memory( ). When pci_dev_info is updated using the pci_attach_device( ) function, a member called BaseAddressSize[ ] is initialized for every valid CPU Base Address found by the

function. BaseAddressSize is the size of the Base Address aperture into the board. Since the Q4 contains one CPU Base Address, hence one BaseAddressSize, this Base Address Size is used as the mapping length.

The PROT_NOCACHE flag is specified for programs in this project. This ensures that the CPU won't defer or omit read/write cycles to the device's registers, nor will it deliver the reads or writes to the registers in a different order than in which they were issued by the driver [34].

Memory mapping is implemented in programs as follows

```
// Ascertain if this is a memory mappable device
if (PCI_IS_MEM(baseAddValue))
{
// Map device memory
memoryMap = mmap_device_memory(NULL,
Q4info.BaseAddressSize[0], PROT_READ | PROT_WRITE |
PROT_NOCACHE, 0, Q4info.CpuBaseAddress[0]);
}
```

The device's memory can now be accessed using the memoryMap pointer. By giving the pointer offsets into the memory, Register Address variables can be created and assigned. Another programming feature is then incorporated so that test programs can use the mapped memory and test card features using Register Read/Write operations.

This feature is the ability to deal with interrupts generated by the PCI device.

## Interrupt Servicing in QNX

As seen earlier, the pci_attach_device( ) function writes to members of the pci_dev_info structure, initialising them to values relevant for the Q4. One of the members of the structure is PciIrq, which the interrupt request number assigned to the device. Interrupts are used extensively in the test applications since they provide an efficient means of communicating with the device. Interrupt related functions are also written for the driver Class.

A large number of Q4 Card features have interrupts associated with them. External interrupts such as the Ext_Int and Fuse, can act as Master-Slave and Emergency interrupts respectively. Analog to Digital Conversion completion is notified using the ADC03_RDY interrupt line, and the Counter, Watchdog and Encoder Channels have interrupts associated with them.

This leads to an investigation of the available interrupt setup, signalling and processing methods on the QNX platform. Both the standalone test programs and driver Class use the InterruptAttachEvent( ) method in conjunction with the sigevent structure. The Real Time Clock interrupt was used to test both Interrupt Service methods, InterruptAttach( ) as well as InterruptAttachEvent( ).

### Use of the Real Time Clock Interrupt

The RTC is associated with the hardware interrupt line IRQ8. As a test interrupt source, the IRQ8 is both reliable and periodic. Also, the interrupt frequency of the IRQ8 interrupt can be controlled by writing to its Address and Data Registers. Moreover, since the IRQ8 is a commonly used interrupt source for designing system clock displays, literature related to it is easily accessible. All of this makes the RTC an ideal interrupt source for experimenting with interrupt servicing in QNX.

The Real Time Clock interrupt was used to test the use of Interrupt Attach methods and Interrupt Wait and Service mechanisms. The code for this was adapted from [35] and modified to suit application requirements.

Programs which processed interrupts coming from the Q4 Card were initially based on the inferences drawn from the experience with the RTC Interrupt programs. The use of the sigevent structure and use of both the InterruptAttachEvent( ) and InterruptAttach( ) methods was suggested by tests on the RTC.

## The sigevent Structure

The Q4 Card is assigned an interrupt ID when pci_attach_device( ) is called. Interrupt requests from the Q4 are routed through this interrupt ID to the system. The question of how these interrupt requests can be delivered to the application is addressed by the use of Event Notification in QNX.

An interrupt request is an example of an Event. A structure called sigevent allows the user to define what is to be done when an event occurs. Examples of reactions to events are: raise an interrupt (SIGEV_INTR), send a pulse to the process (SIGEV_PULSE), send a signal to the process (SIGEV_SIGNAL) and do nothing (SIGEV_NONE)

A sigevent structure is associated with a function such as InterruptAttachEvent( ). Prior to this, the event structure is initialized to define the reaction to an event. For an event to cause an interrupt, the SIGEV_INTR response is defined, using the appropriate initialization macro

```
// Define the sigevent structure variable
      struct sigevent  event;
// Initialize it to raise interrupt on events
      SIGEV_INTR_INIT ( &event )
// Associate with command like InterruptAttachEvent( )
```

As noted earlier, the sigevent structure can be delivered to the process after associating it with an interrupt service method.

## Requesting IO Privity

The thread using Interrupt related system calls needs to have I/O Privity, which gives it the ability to manipulate hardware interrupt sources and affect the processor Interrupt Enable Flag. IO Privity is given using the instruction

```
ThreadCtl(_NTO_TCTL_IO, 0)
```

Root permissions are needed to use this command. If a thread attempts to use these opcodes without successfully executing this call, the thread faults with a SIGSEGV when the opcode is attempted.

## InterruptAttach( ) and InterruptAttachEvent( )

### *InterruptAttach( ) and its Limitations*

Interrupts in QNX can be serviced using either the InterruptAttach( ), or InterruptAttachEvent( ) function. Threads inherited by the calling thread retain IO privity, as provided by the _NTO_TCTL_IO status. This feature is useful when Interrupt Service Routines (ISR), which are separate interrupt processing threads, are used to service Interrupts. ISR usage is made possible by the use of the InterruptAttach( ) method.

The software needs to tell the OS [Print Ref 5] that it needs to associate the ISR with an interrupt source. InterruptAttach( ) is used to specify which interrupt source is to be associated with the ISR.

The ISR operates at a higher priority than any other thread. This makes operations done using the ISR effective almost immediately after the interrupt is called. However, very few kernel calls are available inside the ISR. There is a clear tradeoff between the very low interrupt latency provided by the ISR, and the severe limit on its functionality at that fast access.

*Choice between InterruptAttach( ) and InterruptAttachEvent( )*

The choice between InterruptAttach( ) or InterruptAttachEvent( ) is made on the basis of the work that needs to be done in the ISR [36]. If there is some time critical functionality which needs to be dealt with in an ISR, then the InterruptAttach( ) function call is used. If the ISR needs to do nothing other than schedule a thread which later does all the work at user priority, then InterruptAttachEvent( ) can be used.

Though there is the consideration of the time delay between an interrupt request and process in InterruptAttachEvent( ), it has been preferred over InterruptAttach( ) in the Test Programs for the following reasons

- It has no ISR, hence nothing running in kernel space, hence there is no danger of crashing the system [Krten pg 231]
- Process level debugging can be used on the Interrupt Handler code [36]. The most commonly used debugging method in this project is the use of fprintf( ) statements.

Latency in accessing and servicing the interrupt was deemed to be allowable for the application at hand. Test programs did not present a case for the faster servicing route of the InterruptAttach( ) call.

Interrupt Servicing

```
InterruptWait( )
InterruptDisable( )
InterruptUnmask( )
```

The combination of InterruptAttachEvent( ) and sigevent allows processing of the interrupt at the thread level. The desired interrupts are enabled using the Q4 Interrupt Enable Register, which leads to the next stage; waiting for an Interrupt and post-interrupt Processing.

Waiting for an Interrupt is done using InterruptWait( ). This system call waits for a hardware interrupt. The call to InterruptWait( ) blocks waiting for an interrupt handler to return an event with

notification type SIGEV_INTR (i.e. a hardware interrupt). If the notification event occurs before InterruptWait( ) is called, a pending flag is set. When InterruptWait( ) is called, the flag is checked; if set, it's cleared and the call immediately returns with success.

To avoid early or spurious interrupt notifications, all Q4 interrupts are disabled before Interrupt Processing function calls like the InterruptAttachEvent( ) are employed. The interrupt wait function is called as

```
InterruptWait(0, NULL)
```

Once an event notification in the form of a SIGEV_INTR has occurred, other necessary function calls are used, to ensure that the interrupt is handled as expected. These are InterruptDisable( ) and InterruptUnmask( ).

InterruptDisable( ) disables hardware interrupts. Once ISR or interrupt service related operations have been carried out, InterruptEnable( ) is called. This re-enables hardware interrupts. This re-enabling is required to ensure that the system performance is not affected, since InterruptDisable( ) has disabled all hardware interrupts on the system. It has been stated in literature that the command InterruptLock( ) may be a better alternative to InterruptDisable( ). InterruptLock( ) is said to provide better performance for Symmetric MultiProcessor (SMP) systems [29]. Thus, this command could be used in future revisions of the driver.

InterruptAttachEvent( ) automatically masks the interrupt source when it delivers the event, implicitly using the InterruptMask( ) function call. Hence, the interrupt servicing section needs to unmask the interrupt source using InterruptUnmask( ).

This sequence of operations: setting up the interrupt mechanism, waiting for an interrupt and servicing the interrupt completely explain the interrupt processing mechanism used in the project.

Use of Interrupt calls in Test Programs

The complete sequence of Interrupt Service Commands, in the manner in which they are used in the Test Programs, is as follows

```
// Variable Declaration
      struct pci_dev_info Q4info;
      int Pci_Irq;
      struct sigevent  event;
      int interruptID;
// Assign Interrupt notification type to the sigevent
structure
      SIGEV_INTR_INIT(&event);
// Obtain the interrupt number from the pci_dev_info
structure
      Pci_Irq = (int) Q4info.Irq;
// Call InterruptAttachEvent( ) and get the interrrupt ID
      interruptID   =   InterruptAttachEvent(Pci_Irq,&event,
      _NTO_INTR_FLAGS_END);
// Clear all Q4 interrupt sources using its Interrupt
Status Register
// Wait for interrupt to occur
      InterruptWait(0, NULL);
// Post interrupt function calls
// Disable hardware interrupts
      InterruptDisable( );
// Disable Q4 Interrupts using the Interrupt Enable
Register
```

```
// Clear all Q4 interrupt sources using its Interrupt
Status Register
// Unmask interrupts
      InterruptUnmask(Pci_Irq, interruptID);
```

With the successful implementation of both communication with the Q4 Registers and ability to service interrupts, it is possible to start testing the Card. This begins with a survey of Card Registers, and the features which follow from Register programming flexibility. The Q4 Register Set is examined next.

## Q4 Registers

A detailed look at the Registers is essential for using and programming the Q4. They have been grouped in this Chapter depending on the Card features which they help explore. For more detailed information on the Registers, the reader is referred to [30].

After PCI Communication with the device has succeeded and a hardware interrupt has been associated with the Q4 card, card features can be tested. The Q4 can be programmed using Read/Write operations performed on the Card Registers. All register addresses on the Q4 are relative to the system assigned base address.

### Register Addressing

As was seen during Memory Mapping in [Chapter: PCI Comm] the base address is obtained following a memory mapping of the device address space. The mmap_device_memory( ) command returned this base address. [30] provides a Register Map of the Q4. It shows the Byte Offsets of the Register. Register Addresses is obtained by adding the offset to the available Base Address. An example of this address assignment is

```
d_interruptEnable  = (uint32_t*)(d_memoryMap+0x0000);
```

where d_memoryMap is the base address from memory mapping.

The Register address can then be used to write a value to the Register of choice; e.g.

```
*d_interruptEnable = 0x00800000; // Enable Ext_Int
```

All registers may be read or written to using a 32 or 64 bit access. Some of the registers are 8 or 16 bit accessible as well.

### Interrupt Sources on the Q4 Card

In the previous chapter, it was shown how the QNX system can be set up to acknowledge and respond to interrupt requests coming from various sources, including the Q4 Card.

The Q4 Card has a total of 31 possible interrupt sources, all of which are triggered on the rising edge of the source. The Interrupt Enable Register is used to activate one or more interrupt sources on the Q4. Other than Encoder Interrupts (bits 7-0), seven interrupts are seen in the Register layout. In this documentation, they have been associated with the Registers using which they are configured. Each of the four Encoder Channels has six possible interrupt sources, yielding twenty four Encoder Interrupts. Thus, the total number of Q4 Interrupt sources is 31.

Of the possible interrupt sources, two are available externally on the Q4 Terminal Board. These Interrupts are examined in the next section.

## Interrupt Registers

### *Q4 External Interrupts*

Ext_Int

FUSE

Ext_Int is a special digital input, which is used to generate an interrupt. It can be used as an Emergency Stop interrupt signal or an indicator of limit switch activation. It can be used in simple test routines to verify the Interrupt Service setup. Ext_Int can also be setup as a Watchdog Interrupt, which changes the state of the hardware so as to allow for software recovery.

A falling edge causes the interrupt to occur, since the internal Ext_Int signal is an inversion of the external input at that pin. Thus the Ext_Int pin is given to Ground when an interrupt is to be generated. Pulse on both these external interrupt lines should be at least 40ns wide for the interrupt to be generated.

FUSE detects either that the J1 Cable is not connected or that the Terminal Board fuse has blown by incorrect connection of the Encoder inputs. Pulling out the J1 Connector when the system is waiting for an interrupt is a test method for this interrupt.

There are two registers which are exclusively associated with Interrupt processing, the Interrupt Status and the Interrupt Enable Registers.

*Interrupt Related Registers*

Interrupt Status Register

Interrupt Enable Register

Status Register

The Interrupt Enable Register is used to enable one or more of the Interrupt Sources available on the Q4. Writing a 1 enables an Interrupt, whereas writing a 0 disables it. This 32 bit register is cleared when the PCI Bus is reset. The Interrupt Status Register needs to be cleared before enabling interrupts, otherwise pending interrupts from the Interrupt Status Register will cause an interrupt to be asserted immediately.

The Interrupt Status Register identifies the source of the interrupt. The bit associated with the interrupt source will be SET if the source requested an interrupt. Since more than one source can request an interrupt at a given time, multiple bits may be set in this register. Bit 31 is the INT_PEND bit, which is SET if one or more of the Enabled sources have interrupt requests pending.

To generate an interrupt, the Status and Enable Registers are logically ANDed. If the result of this operation is TRUE, then an interrupt is raised by the Q4. This set of operations is carried out by internal logic of the Q4; the user is responsible only for enabling the right interrupts and clearing the Interrupt Status Register before doing so.

*Status Register:* There is another register called the Status Register, which logs the status of actual signals on various lines, including interrupt sources. This Register can be used to poll for the status of a signal.

The next class of Registers to be discussed is related to Analog to Digital Conversions.

ADC Related Registers

Control Register

A/D Register

Associated Interrupts

*Control Register*

Though the Control Register has a majority of ADC related bits, it can also be used to activate the Ext_Int line as a Watchdog Interrupt [using the EXT_ACT and EXT_POL bits], and to Enable/Disable Encoder Index pulses [using the ENCx_IDX bits]. The Control Register is also used in Enabling/Disabling the DAC Transparent Mode [using the DAC03_TR bit], in which values written to the DAC Output Registers are directly updated at the outputs without the need for further notification commands.

When used in ADC or DAC programs, the Control Register can

- Select the Channels for Analog to Digital Conversion [using ADC_SLx bits]
- Choose between Manual and Automatic Start of conversions [using the ADC03_CT bit]
- Enable the Start Conversion bit when running in Manual mode. [using the ADC03_CV bit]
- Enable the Counter/ CNTR_EN (external Counter pin) based trigger to start conversions in Automatic mode [using the CTEN_CV bit]
- Put ADC Registers in low power Standby Mode to conserve power, a useful feature for embedded applications [using the ADC_STBY bit]
- Choose between Control and A/D Register for Channel selection [using the ADC03_HS bit]. If A/D Register is selected, then bit 9 is used as SCK03, which controls the selection between an Internal and Common conversion clock.

*A/D Register*

The A/D Register is primarily used as an Output Register for the results of Analog to Digital Conversions. It returns the ADC value from selected conversion Channels when Read. When more than one Channel is selected, then successive Read operations return values at Channels starting from the lowest numbered selected Channel to the highest.

However, the A/D can also be used at the Input stage to select Channels for ADC. Setting the lowest 4 bits of the A/D Register, each corresponding to a different Channel, selects Channels for ADC. Using the AD Register for selecting Channels allows a choice between the internal ADC clock and the common Q4 clock, both of which operate at different timing resolutions.

*Interrupts Associated with ADC*

ADC03_RDY

ADC03_OVR

ADC03_EOC

There are three Q4 interrupts associated with ADC- the ADC03_RDY, the ADC03_OVR and the ADC03_EOC. ADC03_RDY is asserted if all the selected Channels have been converted. The ADC03_EOC flag goes HIGH after each Channel conversion is complete. The A/D Register can then be read for the conversion result. The ADC03_OVR flag is asserted if another conversion is started before the previous result has been read from the A/D Register, which is the FIFO for the ADC system.

DAC Registers

D/A Output Register

Mode Register

Mode Update Register

D/A Update Register

The Q4 has four DAC Channels, each of which can be operated in one of three modes - the Bipolar 10V mode, the Bipolar 5V mode and the Unipolar 10V mode. The relationship between the mode and the output voltage can be found in [30].

The DAC Channels have been used in all ADC test programs as analog input sources. During testing, an analog value is generated as a DAC Channel Output, from where it is connected to an ADC input and read back.

### *D/A Output Registers*

There are four D/A Output Registers, one per DAC Channel on the Q4. A 12 bit DAC Ouput value is written to the Output Registers. There is a linear relationship between the DAC voltage and its 12 bit representation. For three points in each operating mode, the output voltage and the corresponding 12 bit representation have been listed in the Q4 Manual. When a voltage has to be written to the DAC Channel, input to the DAC Register can be calculated by a simple formula which uses the linear relationship [Appendix C Formula Calculation].

### *D/A Update Register*

When the DAC is operating in the Transparent Mode, the value written to the Output Registers is immediately reflected as an output voltage at the DAC Channels. However, when the Transparent Mode is disabled, it requires a write to the D/A Update Register for Channels to get updated. The value written to the Register is a random value; it merely serves as an update notification for the Register.

### *D/A Mode Register*

The D/A Mode Register has GAINx and MODEx bits for each DAC Channel. Combinations for these bits results in selection of the desired operational mode of the DAC.

*D/A Mode Update Register*

Similar in concept to the D/A Update Register, the Mode Update Register requires a random value written to it when the DAC is not working in Transparent Mode. In Transparent Mode, writing to the D/A Mode and the DAC Mode Output registers alone suffices in updating operating Mode.

The Transparent Mode is enabled by the Control Register DAC03_TR bit.

Counter Registers

Counter Control Register

Counter Preload Registers

Counter Register

Watchdog Register

Watchdog Preload Registers

*Counter Control Register*

The Q4 has two 32-bit counters, the Counter and the Watchdog, each with 30ns resolution. Of these, the Counter can also be used to trigger A/D conversions, and the Watchdog can also be used as a Watchdog Timer, the expiration of which causes the system to enter a safe state in the event of a software failure. In the normal count mode, the counters can generate either a square wave or a pulse-width modulated wave. The normal count for both counters and Watchdog timer settings are controlled by this register.

Two external pins, CNTR_EN and CNTR_OUT are controlled by this register. The active state of CNTR_EN and its trigger type can be changed using this Register. CNTR_OUT can be programmed to display the Counter output, instead of its default HIGH state.

The counter counts can be set using Preload Registers. There are two Preload Registers assigned to each of the counters. Either can be used to load the count. Selection of the Preload

Registers, including the manner in which the Preload Register are loaded and how count is written to them can also be done using the Counter Control Register.

### *Counter Preload Registers*

Counter Preload Register

Counter Preload High Register

Counter Preload Low Register

There are four counter Preload Registers, two sets of two Registers each. The Counter Preload Register is used to read one of these four Preload Registers. The Register read is determined by the Counter mode, Counter output and active set, and the active Preload Register of that set. All these variables are controlled by the Counter Control Register.

The Counter Preload High and Counter Preload Low are the two Registers which make up each of the two register sets. The Counter Preload Low Register is used to set the Counter period or to set the duration of the Low pulse when the PWM mode is being used. The Counter Preload High Register is used to set the Counter period for a square wave or as the High pulse duration when used in the PWM mode.

The Counter Preload High Register shares its memory location with the Counter Register.

### *Counter Register*

The Counter Register contains the current counter value. When read, it returns the current count. The Counter counts downward from the preload value, then toggles the output at zero and loads the next preload value. The CNTR_OUT external pin can be programmed to display the current state of the Counter using the Counter Control Register's CT_OUTEN bit.

Watchdog Register and Watchdog Preload Registers share a similar functionality with their counterparts in the Counter Registers section.

*Interrupts associated Counter and Watchdog*

CNTR_OUT

WATCHDOG

The Counter can be used to cause a periodic interrupt or to initiate periodic conversions. An interrupt is generated during each period of the Counter when the CNTR_OUT bit is enabled in the Interrupt Enable Register. With appropriate settings in the Control Register, the Counter can be used to trigger periodic A/D Conversions.

The Watchdog Counter has additional functionality as a Watchdog Timer, which uses the WATCHDOG Interrupt. Watchdog Timer is used to detect software failures and set the system to recover from such a failure. It is basically the Watchdog Counter being used with additional settings. When used as a Watchdog Timer, the Watchdog Counter, on expiration, resets the D/A outputs to zero and pulls all Digital outputs HIGH. The counter normally does not expire, unless the software fails and does not reset the Counter. The status of the Watchdog Counter may be monitored externally using the Q4 Watchdog' output. This is the inverse of the Watchdog status bit in the Interrupt Status Register, and can be used as an Emergency Stop signal.

Digital IO Registers

Digital Direction Register

Digital IO Register

The Q4 has 16 general purpose Digital IO pins. They are individually programmable as Input or Output using the Digital Direction Register. Writing to or reading from the Digital IO Register sets the output bit values or fetches the input pin values respectively.

## Encoder Registers

Encoder Data Registers

Encoder Control Registers

The Q4 has two Encoder chips for its four Encoder Channels. ENC01 handles Channels 0 and 1. ENC23 handles Channels 2 and 3. Channels 0 and 1 are mapped to the lowest bytes of Encoder (Control and Data) Registers A and B respectively. Channels 2 and 3 are mapped to the next highest byte of the same registers in the same order. Thus, Encoder Data and Encoder Control Registers A map the even numbered Encoder Channels, and Registers B map the odd numbered Channels.

It is possible to program Registers to deal with all Channels simultaneously.

### *Encoder Data Registers*

Encoder Data Register A

Encoder Data Register B

Encoder Data Registers store the Encoder Channel count value. They can be written to when preloading a register is required for a mode of operation of an Encoder Channel.

Each Encoder Channel is mapped to one byte of the Data Register with which it is associated. Each Encoder Channel has a 24 bit Counter associated with it. In order to deal with this 24 bit data using the 8 bit wide bus available in the Data Register, three consecutive Read or Write operations need to be performed. For a Read, for example, the first Read returns the LSB, the next returns the intermediate byte and the third Read operation returns the MSB of the 24 bit Data. A Byte Pointer keeps track of which byte of the 24 bit value is being accessed.

This Byte Pointer needs to be Reset before reading the first byte of the result. The mode of operation of the Encoder Channels can be controlled using the Encoder Control Registers A and B.

Encoder Control Register A is considered for the next section since both Control Registers are identical in every respect but for the Channels controlled by them.

*Encoder Control Registers*

Encoder Control Register A

Status Flag Register, Reset and Load Signal Decoder, Counter Mode Register, Input Output Control Register, Index Control Register

Encoder Control Register B

Encoder Control Register A is used to program the even numbered Encoder Channels. It has a byte for each individual Encoder Channel which is mapped to it. The Control Register can be set up to program both the even numbered Channel and the odd numbered Channel associated with the same Encoder Chip. This is done by setting the most significant bit for the Channel.

For example, Channel 2 is mapped to the second byte (bit 8-15) of Encoder Control Register A. Setting its most significant bit (bit 15) to value 1 applies all the bit settings on Encoder Channel 2 to Encoder Channel 3, since both Channels are controlled using the same chip, ENC23.

The four combinations of Bits 6 and 5 of the Encoder Channel byte give access to a different Register for Encoder Control. The rest of the bits (4-0) are then used to program the Channel using the options provided by the selected Register.

Reading from the Encoder Control byte returns a byte called the Status Flag Register. Hence, there are five Registers to which access is provided using the Encoder Control Registers. These Registers and their features are explained ahead.

*Status Flag Register*

The Status Flag Register is returned on reading the Encoder Control Register. It shows the status of various flags in the Encoder system and also comparison flags for the count.

*Reset and Load Signal Decoder*

Byte 6-5 : 00

Used to

- Transfer Encoder Counter value to Output latch, necessary before a count READ operation.
- Reset Counter or Flags
- Transfer Preload Register to Counter, required in Preload operations.
- Reset Byte Pointer, required prior to a Read/ Write operation.

*Counter Mode Register*

Byte 6-5 : 01

Used to

- Set Quadrature Mode.
- Select Counting mode.
- Select type of Counter (binary/ BCD)

In non-Quadrature Mode, the Encoder A and B inputs are taken as Count and Direction inputs. Otherwise, they are considered as quadrature inputs, and the Quadrature Mode selects the Quadrature Multiplying Factor.

The Counting Mode determines how an Encoder Counter behaves at count limits and how the upper limit of a count is determined. Modes of operation are Normal, Range Limit, Non Recycle and Module N. In the Normal Mode, the counter counts between 0 and 0x00FFFFFF, and wraps around at the limits, setting a CARRY or BORROW flag in the process. In Range Limit Mode, the counter freezes at the lower limit, zero, and the upper limit, until the count direction is changed. In the Non Recycle Mode, the counter freezes when either the upper or lower limits are reached, and

unfreezes only after a load or reset operation. The Module N mode is just like the Normal Mode, except that the upper limit is user definable. The upper limit is also user definable for the Range Limit Mode.

*Input Output Control Register*

Byte 6-5 : 10

Used to

- Configure action of the Index/Load (I/LD) input.
- Enable A/B inputs
- Set Encoder Flags outputs for Interrupt Registers.

Encoder output Flags can be selected in this Register. These flags may be read in the Status Register, or used as interrupt sources in the Interrupt Status Register. Four combinations of the FLG outputs are provided for the user to choose FLG interrupt sources.

*Index Control Register*

Byte 6-5 : 11

Used to

- Determine Index polarity
- Enable Index Mode.

*Interrupts Associated with Encoders*

ENCx_FLG1

ENCx_FLG2

Interrupts may be generated by the toggling of the FLG bits defined by the Input Output Control Register.

# CHAPTER FIVE

# IMPLEMENTATION AND TESTING OF THE INTERFACE

The testing of individual Card features, PCI Communication and Interrupt Service methods culminates in the implementation of the device driver. The layout of the driver has been influenced by a previous implementation of PCI device driver for the RoadRunner Board, though it differs significantly from this project in its implementation.

The Class implementation follows CRB coding conventions as formulated in [32].

## Outline of Driver Class QuanserQ4

The Quanser Q4 Driver is written as a C++ interface. It features forty nine functions, designed keeping in mind the need for the user to have maximum control over the operation of the Q4 PCI Board. In addition, a diagnostic choice and greater control is provided to the user in the form of inline function implementations which are used in directly issuing Register Read or Write commands. Input macros are defined such that function input arguments are self explanatory. Error or Warning Returns have been defined such that the cause of the error is made as clear as possible to the calling program.

## Introduction to Class QuanserQ4 Functions

The Quanser Q4 can function as a Data Acquisition System by way of its ADC Channels, Digital IO ports and Encoder Channels. Control related operations are performed by the use of the DAC Channels, external signal pins such as the Ext_Int and CNTR_EN, and Digital IO pins. A lot of operating modes require the use of Interrupts on the Q4, and Counters and Timers internal to the Q4.

In addition to Q4 specific features, the Constructor needs to be designed such that the card is properly initialized and memory is correctly mapped before more complex operations can be performed successfully.

Given the above requirements, the functions written for the Q4 Driver Class are grouped as follows

- Constructor/ Destructor
- Device Information
- ADC
- Interrupt Service
- DAC
- Digital IO
- Counter/ Timer
- Encoders
- Inline Implementations

Function descriptions are given ahead. A more detailed synopsis of functions can be found in the [Appendix].

Constructor/ Destructor

```
QuanserQ4( )
~QuanserQ4( )
```

The Q4 is initialized using the Constructor, which attaches PCI Server, finds PCI BIOS, initializes pci_dev_info, attaches the driver and memory maps the Q4 Card. After memory mapping the card, Register pointer variables are initialized so that they contain the correct Register Addresses. All non-static variables and arrays, which are expected to retain their values across function calls for the same object, are also initialized.

If an error occurs in any of the initialization functions, then an error variable is set such that all of the subsequent functions return an error before any variable changes or memory operations are performed.

The Destructor unmaps device memory and detaches the card. The handles for both are set in the Constructor.

### Device Information

```
printBoardInfo( )
```

These functions are meant to print certain information on the Terminal window. The amount of information to be printed is determined by a user input value called verbosity. Two levels of verbosity are allowed by the software, 0 and >0.

### ADC Functions

```
setupADCChannels( )

setupADCChannelsADReg( )

configCntrenADC( )

startADConversion( )

readADCChannels( )

enableADCStandby( ) ;  disableADCStandby( )
```

The Q4 contains four Channels for Analog to Digital Conversion. The Channels on which ADC needs to be performed are selected by the user by writing to the array selectADC[ ]. The selectADC[ ] array is a four element array, initialized to zero. There is equivalence between the index of the array and the ADC Channel number. To select a Channel for ADC, a non-zero value is written to the array element corresponding to the Channel. e.g. selectADC[2] = 25 selects ADC Channel 2.

Once the ADC Channels have been selected, they are set up using one of the two functions, setupADCChannels( ) and setupADCChannelsADReg( ). These functions allow the user to select if

the conversion is to be an Automatic or Manually initiated process. startADConversion( ) is used for Manual initiation of the AD Conversion. setupChannelsADReg( ) is used when it is desired to use the AD Register for Channel selection. Use of the AD Register allows the user to choose between the internal clock and the common clock on the Q4 for the conversion.

For Automatic initiations of the AD Conversion, either the Counter or the external CNTR_EN pin can be used. The Counter can be configured for ADC using appropriate Counter/Timer functions. configCntrenADC( ) is used to configure the use of the CNTR_EN pin to initiate conversions.

The end of conversions is marked using one of the ADC Interrupts; ADC03_RDY or ADC03_EOC. The application can wait for these interrupts to occur. When this happens, the four element array readADC[ ] has been written to by the driver, and conversion outputs can be read using this array with the index corresponding to the ADC Channel.

ADC Standby is a special power saving feature which is enabled and disabled using the functions assigned to it.

Interrupt Service Functions

```
enableExternalInterrupts();disableExternalInterrupts()
enableExtIntWatchdog( ) ; disableExtIntWatchdog( )
enableEncoderInterrupts(); disableEncoderInterrupts()
enableADCInterrupts( ) ;  disableADCInterrupts( )
enableCounterInterrupts(); disableCounterInterrupts()
enableAllInterrupts( ) ; disableAllInterrupts( )
waitForEnabledInterrupts( )
```

There are four types of Interrupt sources on the Q4: External, Encoder, ADC and Counter. Enabling or disabling each of these kinds of interrupts is done using the corresponding function from the list. It is recommended that all interrupts be disabled at the beginning of the program,

following which interrupts of interest are enabled. Further, it may be a safe programming practice to always enable one or both of the External Interrupts (FUSE and EXT_INT) since they can both serve as failure indicators for the system.

The function waitForEnabledInterrupts( ) allows the setting of a wait window for an enabled interrupt to occur. QNX Timer Timeouts are employed as the mechanism to terminate an InterruptWait( ) operation. The wait period has a least count of 1ms for this implementation, but this may be reduced to allow for up to a 1ns resolution. Latency from the Q4 in signaling an interrupt may place a lower bound on the wait period. If the timed wait is not desired, the wait period can be set to infinity using INF as the argument in waitForEnabledInterrupts( ).

It has to be noted that waitForEnabledInterrupts( ) clears all bits of the Interrupt Enable Register in its post interrupt phase of operation. Some interrupts may need to be re-enabled after the function is called, if so required. The InterruptAttachEvent( ) method is used for interrupt servicing by this function.

DAC Functions

```
enableDACTransparent( ) ; disableDACTransparent( )
writeToDACChannel( )
updateDACChannels( )
```

The Q4 has four DAC Channels. They are all capable of operating in three modes of operation, the Unipolar 10V Mode, the Bipolar 5V Mode and the Bipolar 10V Mode. The mode of operation, Channel selection, and selection of the desired voltage is done using writeToDACChannels( ). Out of range desired voltages are automatically corrected to the range limit they are closest to, for that mode of operation.

The DAC can be operated in the Transparent Mode. In this mode, the operating mode and output value are immediately updated for all Channels. This is contrasted with the non Transparent mode, in which updateDACChannels( ) needs to be called for a output update.

updateDACChannels( ) writes a non zero value to both the Update and Mode Update Registers, which causes a change in the DAC output.

Intuitively, it follows that neither of the update Registers are to be used for DAC Transparent Mode. However, it has been noted that merely using writeToDACChannels( ) in Transparent Mode works in updating Channel outputs only in the Unipolar 10V Mode (which is the default Mode). When a different Mode is used as the input argument, an incorrect value is noted at the DAC Channel.

It is found that, though the Update Register does not need to be written to, a write operation to the Mode Update Register is required even when the DAC is operating in Transparent Mode. This may be a design glitch or a documentation error from Quanser regarding the properties of the Transparent Mode.

Digital IO Functions

```
configureIOAllPins( )
setDigIOPinDirection( )
writeDigIOPinValue( )
readDigIOPinValue( )
```

The Q4 has sixteen digital pins which can be individually configured either as input or output. configureIOAllPins( ) allows the user to configure all pins at the same time, by writing to the Digital Direction Register and the Digital IO Register.

The pins can be separately configured as well. setDigIOPinDirection( ) configures the digital IO pin as Input or Output. The functions writeDigIOPinValue( ) and readDigIOPinValue( ) are then employed, according to the chosen configuration of the pin.

Counter/Timer Functions

```
initCounters( ); setCounterMode( )
setSqWavePeriod( ) ; setPWMPeriod( )
setCounterRWReg( )
startCounter( );      stopCounter( )
setupWDTimer( ); startWDTimer( ) ;  disableWDTimer( )
```

The Q4 has two counters, called the Counter and the Watchdog. The Counter can be used as a trigger for AD Conversions, while the Watchdog can be used as a Timer which puts the system in a safe configuration if allowed to expire. These features are in addition to their potential use as normal 32-bit counters.

*Normal Counters*

The counters are initialized using initCounters( ), which sets all the Preload Registers to zero. Preload Registers store information about the count period. Setting all Preload Registers to zero is a precautionary procedure, and in no way necessary for the correct functioning of the Counters. Hence, the use of this function, while recommended, is not obligatory.

Either counter can be run in one of two Modes: Square and PWM. In the Square Wave mode, a free running square wave is generated. The period count is reloaded from the Preload Low Register each time. The PWM (Pulse Width Modulation) mode uses both Preload Registers, High and Low, to load its count. The mode of counter operation is selected using setCounterMode( ).

Once the counter mode has been set, its period is determined by the inputs to the functions, setSqWavePeriod( ) and setPWMPeriod( ). Either of these functions is used, depending on what mode has been selected previously.

Each counter has two Register Sets of each type, and setCounterRWReg( ) can determine which set is used for Read, and which one for Write operations. Specifying different Register Sets gives a smoother transition between duty cycles in the PWM mode. This is an additional feature

which could be of use only in applications which has a specified need for smoothness of state transition.

Counters are started using the startCounter( ) function, which also requires the user to specify whether the Counter is triggering an ADC. stopCounter( ) stops and disables the counter.

### *Watchdog Timer*

When the Watchdog is to be used as a Watchdog Timer, the function setupWDTimer( ) is called. This function sets up the Watchdog counter in square wave mode, activates the Watchdog capabilities of the counter and enables the Watchdog Interrupt in the Interrupt Enable Register. These settings are reversed when disableWDTimer( ) is called, whereas startWDTimer( ) initiates the Watchdog Timer. To use the Timer as a method of reacting to software failure, the timer is initiated, and restarted using startWDTimer( ) before it expires. In the event of a software failure, the timer will not be reset, and the count will reach zero, leading to the safe configuration being enforced for the Q4.

## Encoder Functions

```
readEncStatusByte( )
setupEncPreloadReg( )
setupEncChannels( ) ;  readEncChannels( )
setupEncSingleChannel( ) ;  readEncSingleChannel( )
enableEncIdxPulse( ) ; disableEncIdxPulse( )
```

The software architecture of the four Encoder Channels has been detailed in the previous Chapter. Encoder functions seek to realize the features offered by the Q4 Encoders.

Each Channel has a Status Flag Register, which may be read using the readEncStatusFlag( ) function. The status of flags such as Index, Carry/Borrow, Error is returned as a 8 bit unsigned integer.

Each of the operating modes of the Encoder Counter allow the use of Preload Registers which either set the upper range limit, or the initial Counter value depending on the mode of operation. setupEncPreloadReg( ) is used to load values into the Preload Register, and needs to be called separately for every Channel which is to be used. The function setupEncPreloadReg( ) has to be called before setupEncChannels( ).

setupEncChannels( ) allows the user to customize the Encoder Channel. Count Mode is determined to be one of Normal, Range Limit, Non Recycle and Module N. The manner is which A and B inputs to the Encoder are interpreted is set using the Quadrature mode. A combination of two Encoder Flag bits can be used to trigger an interrupt, and that combination is selected using this function. Index inputs may also be enabled or disabled here. The functions enableEncIdxPulse( ) and disableEncIdxPulse( ) make changes to the Control Register and enable or disable index pulses for one or more channels, as specified.

Once the setupEncChannels( ) function has been called, any movement of the Encoder Shaft connected to the channel (or channels) set up is recorded by the channel's Encoder Counter. Interrupts are triggered if the conditions are right. setupEncChannels( ) is thus effectively a setup and start function for the Encoder Channel to sense and react to shaft position information.

The count values are loaded into an array called encChannelVal[ ], a four element array whose index corresponds to the Encoder Channel number. To read the Encoder Channel, the function readEncChannels( ) is called. This loads the current Channel count into the encChannelVal[ ] array for selected Channels.

To work with a single Encoder Channel, the functions setupEncSingleChannel( ) and readEncSingleChannel( ) are used in place of setupEncChannels( ) and readEncChannels( ). Single and multiple Channel operations have been isolated because of the complexity associated with combining them into a single function. The Q4 Encoders lend themselves easily to working with combinations of Channels, such as Even, Odd and All Channels. Configuring and reading a single Channel involves a single byte read and write process, which differs from the 32 bit operations when

working with channel combinations. Separating these two types of Encoder Channel accesses simplified the programming at virtually no additional cost to user convenience.

## Inline Implementations

### All permitted Register Reads

### All permitted Register Writes

Inline implementations were added as a diagnostic tool for the user. All functions, by their nature, come with a certain degree of programming rigidity. Many of the Register operations are opaque to the application by design, since the driver software handles those operations in the background. However, it is feasible that there may be a need to troubleshoot a performance problem in the Card by first checking if the fault lies with the driver itself. In that case, a useful feature to have would be the ability to bypass the detailed functions and interact directly with the Card using Register Read/Write operations.

Inline implementations have been designed for this Class to provide this ability to interact directly with Card Registers. They allow Register Read/Writes; if the test application is interrupt based, however, the user will need to write his or her own code to service interrupts.

## Input/Output Macros

Input arguments for functions can be passed to it in the form of macros. These macros have names which are easy to remember abbreviations of the requested input choice. Function arguments which do not have macros associated with them include desired voltage output for a DAC Channel, and timeout period selection for an interrupt wait.

Output returns in this software are of three types

- Returns, which indicate success in a program and provide a positive return value which gives additional information about it

- Warnings, positive values which indicate that there may be an incorrect output because of certain flaws in inputs or because a function was called out of sequence, and
- Errors, which return negative values if an error has been detected in the setup or execution of a function call, the negative value itself being an indicator of what kind of error has occurred.

All such output returns are macros, with the sole exception of the function readEncStatusByte( ), which returns a unsigned integer byte which is an actual Register value.

## Installation and Documentation

### Installation using Makefile

The Q4 Driver is easily installed on another system using a simple makefile template and manually adding the generated header and library files to appropriate directories. Once this is done, the driver is ready for use in the target system.

The Makefile for the driver is adapted from the makefile for the ServoToGo Board. The make result of this file is a library file called libQuanserQ4.a and an object file. The library file is to be manually copied to the QRTS Library folder (`/usr/qrts/lib`).

In addition to this, the QuanserQ4.hpp header file is manually copied to the QRTS Include folder (`/usr/qrts/include`).

To use the Class QuanserQ4 in QMotor programs, the Makefile from (`/usr/qrts/makefiles`) called qmotorinc has to be modified to include this additional line

```
-lQuanserQ4
```

This adds the Quanser Q4 library to the make path of the makefile automatically generated by the QMotor command

```
createqcp <Project Name>
```

and allows the driver to be used on the QMotor interface.

This completes driver installation. The body of the C++ program which intends to use the Quanser QNX driver is merely required to include the header file. This makes the interface to the Quanser Q4

operational and all the driver Class functions may then be used as per the user's requirement and discretion.

## Documentation

Class documentation was automatically generated using the program `doxygen`. The software `doxygen` creates documentation based on its own method of interpreting and identifying functions, variables, header file includes and macros. Additionally, `doxygen` tags were used in the Class interface to better present information about the driver.

Functions and variables can be conveniently grouped according to their functionality, and this grouping is documented on a separate html or document page by `doxygen`. Example programs can be added to the documentation. Function information can be displayed below the function name.

The fact that `doxygen` was intuitive and easy to use, and that it produced good documentation results made it an appealing choice for automatically generating a document for the driver Class.

A synopsis of the class QuanserQ4 functions, including their input arguments and returns, can be found in Appendix A.

## Software Testing

The driver Class has been written keeping the features of the Q4 as reference. The performance of the driver is tested by using the Class in small C++ Programs which test a single feature, or a combination of features. Function input arguments are varied for these test programs, so that most possibilities are covered.

The example program cited in the doxygen documentation is QuanserQ4.t.cpp. However, many more example programs have been written during the testing phase of driver development.

These programs emulate the objectives of standalone C test programs written before the Class QuanserQ4 was constructed.

The driver interface would typically be used with control software such as QMotor. Representative C++ standalone programs for the driver interface are tested for portability to QMotor, and it is found that they can be easily adapted to run in conjunction with QMotor. This test of portability is essential, since experiments in the research group are conducted with the stable environment of the QMotor being the one of choice.

## C Test Programs

Standalone test programs were implemented before software development entered the phase where the driver Class was constructed. A sequence of Register Read/Write operations built these test programs. C test programs sought to verify whether the logic for programming the PCI device, in terms of Register bit settings and load operations, was correct.

A typical C test program followed this algorithm

In the main program

- Initiate a thread for device interaction using `pthread_create( )`
- `sleep( )` for n seconds while the above thread is performing card related operations.

In the thread created in main( )

Initialization

- Initialize PCI device
- Map device memory
- Initialize Card Register pointers

Operations

- Write to Card Registers to setup the mode of operation.

- Enable interrupts.
- Wait for interrupts.
- Service interrupts.
- Perform post interrupt Read or measurement operations if applicable.

Termination

- Unmap device memory
- Detach PCI device.

## Transition to C++ tests

Driver Class functions were written such that all Initialization is performed by the Constructor and all Termination by the Destructor. For Operations, as seen in the previous chapter, function names and argument names were as self-explanatory as possible. C++ test programs did away with the need for a sleep( ) function call in main( ), since the device driver thread was part of the main( ) program itself.

A typical C++ test program followed this algorithm

Initialization

- Create an object of the class QuanserQ4.

Operations

- Call appropriate functions to setup the Card.
- Call appropriate interrupt functions to enable desired interrupts
- Call waitForEnabledInterrupts( ) to wait for a specified period of time for an interrupt.
- Read data stored in appropriate arrays, or measure signal if applicable.

Termination

- (Class Destructor)

The C++ test program size is reduced to about a tenth of the C test program size because of the modularity. For example, the body of a program which waits for the external interrupt Ext_Int to occur looks like this

```
// Create object
      QuanserQ4 first;
// Disable all Interrupts
      first.disableAllInterrupts( );
// Enable Ext_Int
      first.enableExtInterrupts(EXT_INT);
// Wait 10ms for Ext_Int to occur
      first.waitForEnabledInterrupts(10);
```

This compared with the original C test program, the body of which was in excess of 500 lines of code. This example illustrates the level of ease achieved by the implementation of appropriate functions for interacting with the Card.

Test programs, as described in the introduction to this Chapter, covered almost all verifiable aspects of card performance. The same kinds of tests were conducted using both C and C++ as programming languages. The former for the use of its direct Register programming in verifying the algorithm, and the latter for its ability to test what was to ultimately become the Q4 driver.

A brief summary of all test programs is given ahead.

Summary of Test Programs

extInterrupt.cpp

Tests external interrupts EXT_INT and FUSE.

Equivalent C program: Q4_ISR_AttachEvent.c

digitalIO.cpp

Tests settings for digital input/output pins. Pins can be programmed all the same time or separately.

Equivalent C program: Q4_DigitalIO.c

adcManualMode.cpp

Tests ADC in Manual Mode, performing conversions on one or more Channels. DAC output is used as the input source for ADC Channel.

Equivalent C program: Q4_ISR_AttachEventADC.c ; Q4_ISR_ADCmanyChannel.c

dacModeReg.cpp

ADC is used in Manual Mode as in adcManualMode. DAC is not used in transparent mode, mode and channel updates are used for updating DAC output value.

Equivalent C program: Q4_ISR_DACusingModeReg.c

adcAdReg.cpp

ADC Channel selection performed using AD Register. This also allows selection of the ADC clock.

Equivalent C program: Q4_ISR_ADCusingADReg.c

adcCtr.cpp

Counter triggered AD Conversions are tested. The Counter is configured in Square Wave mode.

Equivalent C program: Q4_ISR_ADCwithCounter.c

adcCtrLoop.cpp

Periodic AD Conversions are performed. Counter works in Square Wave mode and is configured to trigger conversions.

This program exhibits a performance issue, in that out of 10 sample conversions, about three give erroneous results regardless of the Counter period.

Equivalent C program: none

adcCntren.cpp

AD Conversions are triggered by a change in the state of the external input pin, CNTR_EN.

Equivalent C program: Q4_ISR_ADCwithCNTREN.c

adcStby.cpp

Test characteristics of the ADC when operating in Standby Mode. Namely, ADC conversions should not be initiated if in this mode, but conversion results may be read.

Equivalent C program: Q4_ISR_ADCStbyCNTREN.c

cntrAll.cpp

Counter, or Watchdog, or both, are configured as Square Wave or PWM generating Channels and their outputs are checked using a Digital Multimeter using their respective external output pins.

Equivalent C program: none

wdTimer.cpp

The Watchdog counter is used as a Watchdog Timer with a specified period. When it times out, DAC Channels are set to output zero, and Digital IO pins all read HIGH, this until the timer is disabled and DAC and Digital IO reprogrammed.

While the .cpp was successfully executed, there were problems with the C implementation of this test. The cause of this problem was identified while writing the C++ test program.

Equivalent C program: Q4_ISR_WDTimer.c

wdExtInt.cpp

The external interrupt Ext_Int is configured as a Watchdog Interrupt. This, in master-slave controllers, can be an input pin for the slave to recognise that the master has asked for a safe configuration to be realised immediately.

Equivalent C program: Q4_ISR_WDExtInt.c

encNormalMode.cpp

Encoder Channels are operated in Normal Mode. Register presets are used to set an initial count for the Channels. Encoder Interrupts on CARRY and BORROW are also tested.

Equivalent C program: Q4_Encoder_Channels_2.c

encRangeLimit.cpp

Encoder Channels are operated in Range Limit mode. The upper limit of the range is set using Preload Registers.

This program did not freeze the count at the user defined upper limit. The Q4 was consistent in not allowing the user defined upper limits to act as limits which could freeze the counter, as expected in Range Limit mode, or trip a CARRY flag, as would be expected in Module N mode. It was verified several times that the algorithm for enforcing upper limits was correct.

Equivalent C program: Q4_Encoder_RangeLimit.c

encNonRecycle.cpp

Encoder Channels are operated in Non Recycle mode, employing presets to set initial counts.

Equivalent C program: Q4_Encoder_NonRecycle.c

encModuleN.cpp

Encoder Channels are operated in Module N mode.

The program gave upper limit response errors as noted in encRangeLimit.cpp

Equivalent C program: Q4_Encoder_ModuleN.cpp

C programs for testing individual Encoder Channel programming were not implemented in .cpp form. Also not implemented in C++ was a C program which tested the polling method for interrupt servicing. This was found to be too system expensive, and the possibility of polling as an interrupt service method was abandoned early, as QNX methods were found to be reliable and easy to use. A C program was also written to implement interrupt timeout differently, using a combination of InterruptWait( ) and the Counter interrupt. However, using this came at the price of not being

able to use the Counter, and thus this approach was dropped in favour of the QNX TimerTimeout( ) method.

This is the complete summary of the test programs implemented for this project. Except where mentioned, the programs performed just as expected. A sample test program has been given in Appendix B.

## Testing using QMotor

QMotor is versatile motor control software commonly used in the Controls and Robotics Group. The Q4 Card could be used most often on a QNX system running QMotor, hence it was essential to test the performance of the driver in the QMotor environment.

On creation of a project in QMotor, a framework for implementing the algorithm is created and made available to the user. The .cpp file created as part of this framework contains a control Class declaration, called Class <Project Name>, which can be modified to include additional, user defined Classes and their corresponding header files. The Class QuanserQ4 is added to the QMotor interface in this fashion, and the header file QuanserQ4.hpp is included; thus making all the provisions of the driver available for use with QMotor.

The only other change needed to make the driver functional in QMotor was to modify one of the QRTS makefiles, as has been described in the section 'Installation and Documentation' from this chapter.

Two significant observations were recorded in the testing using QMotor. The Card Index provision in the Q4 Constructor was unavailable when Class QuanserQ4 was initialized in the QMotor Class <Project Name>. Initialization of the Class QuanserQ4 using a Constructor argument was forbidden. This led to the removal of the card index as the Constructor argument. Secondly, InterruptAttachEvent( ) returned an error when used in the function startControl( ) of the project Class. However, the interrupt attach process worked in the function enterControl( ). This characteristic could not directly be attributed to a property of the function

waitForEnabledInterrupts( ) from the Class QuanserQ4. This behavior needs to be examined and rectified to allow complete utilization of the Q4 feature set.

The test program for the Q4 running on a QMotor project did the following

- Read an Encoder Channel and plot the Channel Count curve
- Send a Sine wave as DAC Channel Output and read back the same at the ADC Channel.

The Encoder was run in Normal Mode of operation, Index mode was disabled and its count was Preset to 123456 in decimal representation. During the execution of the program, the Encoder Channel 0 count was logged and plotted. This plot is seen in Figure 5.1. The count increased or decreased in number depending on the direction of rotation of the shaft, with clockwise rotation contributing an increase, whereas anticlockwise rotation reduced the count. No motor was connected to the Encoder shaft, and the perturbations were manually generated.

In the second program, sine wave value during control loop was calculated and assigned to DAC Channel 1. This Channel was setup to operate in the Bipolar 10V Mode, without the DAC Transparent update feature. Amplitude of the sine wave was set to a peak value of 8V. The voltage generated at the DAC Channel was fed to the AD Channel as input. The ADC Channel 1 operated in the Manual Conversion Mode. It was seen that the ADC readout traced the same wave shape as output by the DAC, albeit with a small time lag, of the order of about 0.1 sec. Such a time lag was not observed for AD Conversions in the standalone C++ Programs, and could be examined in future work on driver performance. The plot for this part of the test can be seen in Figure 5.2.

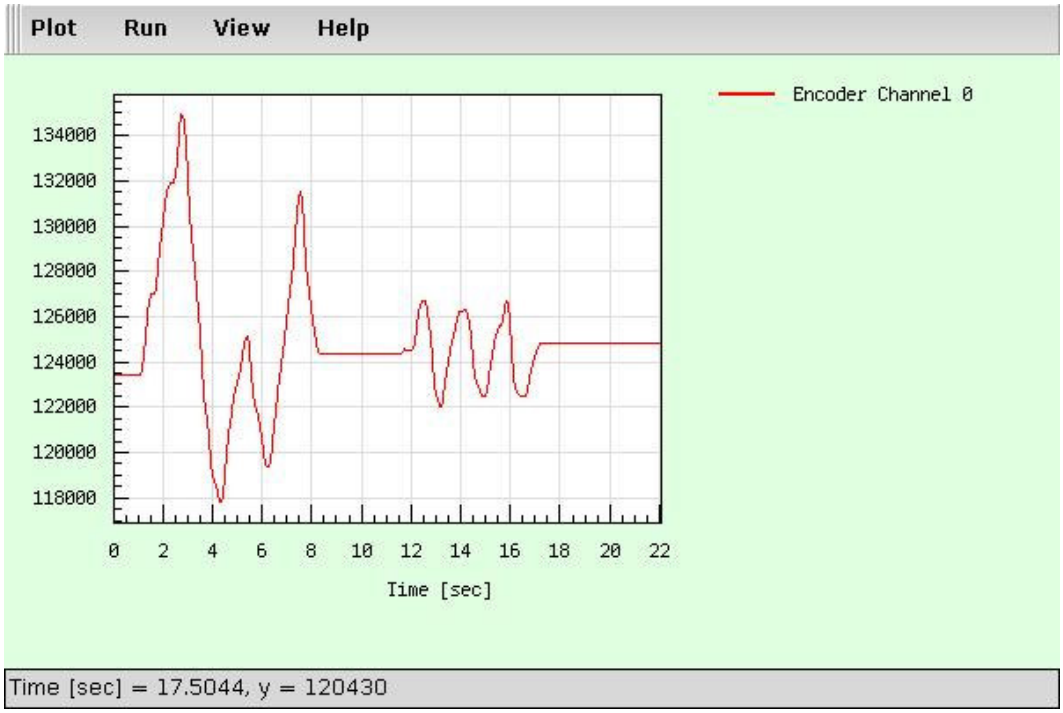


Figure 5.1 Plot of Encoder Channel Count

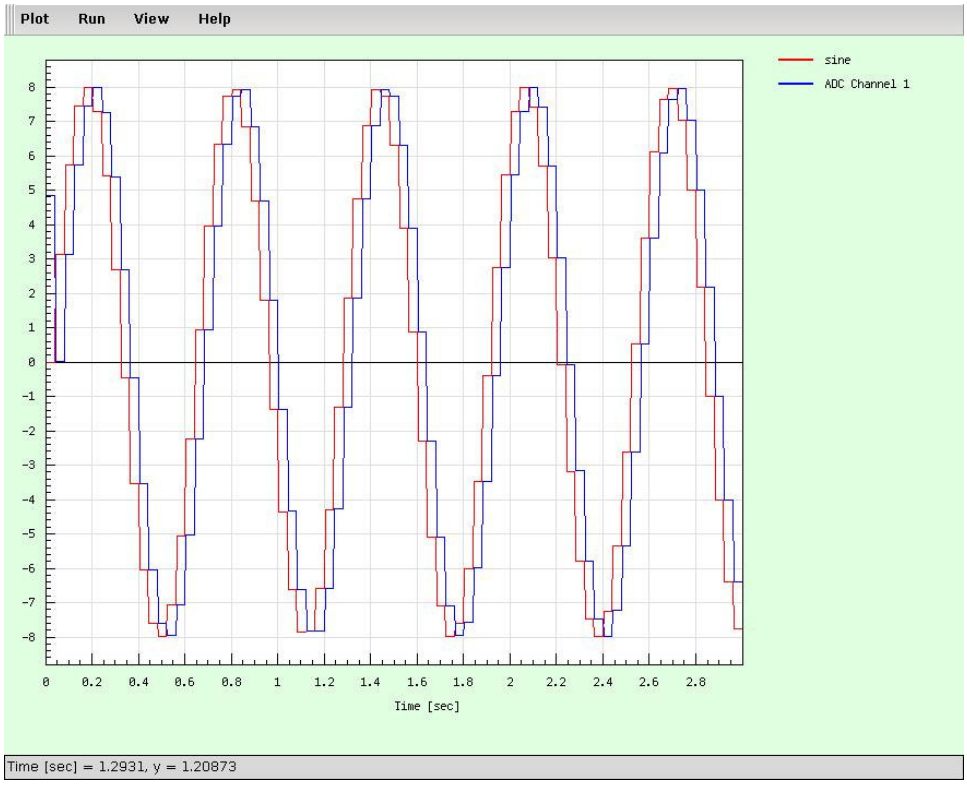


Figure 5.2 Plot of DAC Output and ADC Readout of DAC Output

*Conclusion*

The Quanser Q4 Driver has been written with the objective of providing a user friendly software interface to the Data Acquisition and Control Card while making the program as computationally efficient as possible. A significant amount of time during driver development was devoted to learning and implementing QNX system calls and Q4 functionalities, and integrating these features into one algorithm.

The software has been tested numerous times for its ability to carry out program objectives. Nearly all features of the card have been successfully tested. There remain a few recurring errors, which have yet to be eliminated. These have been recorded for the purpose of troubleshooting at a later stage. These include Encoder Range Limit and Module N mode upper limit setting errors, and performance issues when single Encoder Channels are configured.

These errors aside, the test programs deliver exactly as expected, even when all possible variations and operating limits are tested. The use of InterruptAttachEvent( ) for interrupt servicing ensures that the programs use very little system memory even when waiting for card response in the form of an interrupt. The latency associated with the use InterruptAttachEvent( ) does not seem to affect card performance. However, it is desirable that a future version of the driver be able to measure this interrupt service latency and provide statistical backing to this statement about card performance being unaffected by the use of the present interrupt service method.

A conscious effort has been made to comment the test and example programs so that the procedure followed is explained in the body of the program itself. For C test programs, testing methods have been detailed. These were deemed necessary since C test programs perform low level Register Read/Writes, and making changes to them would require a precise idea about how they work in the first place. All errors and possible pitfalls have been documented.

This will enable future work on the driver to be assisted to some extent by experiences from this project. It is hoped that this will accelerate work, when required, on improvements to the driver, and deliver software which has sustained practical use in the Control and Robotics Group.

# APPENDICES

# Appendix A

## Q4 Functions: Summary

CONSTRUCTOR/ DESTRUCTOR

QuanserQ4 ( int boardNumber );

Attaches PCI Server, finds PCI BIOS, initializes pci_dev_info and attaches driver, memory maps the Q4 Card.

Input Arguments

int boardNumber - The index of the Q4 which is to be found. If the first (or only) card in the system is to be located, 0 is entered as the argument.

Returns none

Warnings none

Errors Since Constructor does not return anything, all errors are indicated using fprintf( ) and the error returns are incorporated into all other function calls. See Note at end of this section.

~QuanserQ4 ( void );

Unmaps device memory and detaches the card.

Input Arguments none

Returns none

Warnings none

Errors Error messages in the form of fprintf( ) statements

*Note on Error Return common to all other functions*

*All other functions return* FATAL_INIT_ERROR *if a card Initialization error is encountered in the Constructor.*

DEVICE INFORMATION

int printQ4Info ( int Verbosity ) ;

Prints card information according to the requested verbosity level

Input Arguments

int Verbosity - 0 or 1.

Returns

For verbosity = 0

Q4 Device ID, Vendor ID, Subdevice ID and Subvendor ID

For verbosity >= 1

verbosity = 0 information and IRQ Number and CPU Base Address, in addition.

Warnings none

Errors none

ADC FUNCTIONS

int setupADCChannels( int conversionMode );

Enables Channels and the preferred conversion mode for Channels selected using selectADC[ ]. The Control Register is used for enabling Channels.

Input Arguments

int conversionMode

ADC_MANUAL : Manual Conversion

ADC_COUNTER : Counter- triggered conversions

ADC_CNTREN : CNTR_EN external line triggered conversions

Returns none

Warnings none

Errors

NO_CHANNEL_SELECTED : No Channels have been selected using the selectADC[ ] array

int setupADCChannelsADReg ( int conversionMode, int selectClock );

Similar to setupADCChannels( ), except that the AD Register is used for enabling Channels. This allows for additional selection between internal and common clock for conversions.

Input Arguments

int conversionMode

ADC_MANUAL : Manual Conversion

ADC_COUNTER : Counter- triggered conversions

ADC_CNTREN : CNTR_EN external line triggered conversions

int selectClock

INTERNAL : Internal clock in the AD chip. 150 ns tick period.

SCK03 : External or common clock. 210 ns tick period

Returns none

Warnings none

Errors

NO_CHANNEL_SELECTED : No Channels have been selected using the selectADC[ ] array

int configCntrenADC ( int selectEdge, int selectPolarity );

Configure triggering characteristics of the CNTR_EN external pin.

Input Arguments

int selectEdge

EDGE_FALL : Trigger on falling edge

EDGE_RISE : Trigger on rising edge

int selectPolarity

POL_HIGH : Active High

POL_LOW : Active Low

Returns none

Warnings none

Errors none

int enableADCStandby(void);

Enables the ADC Standby Mode.

Input Arguments none

Returns none

Warnings none

Errors none

int disableADCStandby(void);

Disables the ADC Standby Mode.

Input Arguments none

Returns none

Warnings none

Errors none

int startADConversion (void);

Start ADC Conversion for selected Channels. This function only has effect in ADC Manual Mode.

Input Arguments none

Returns none

Warnings none

Errors

NO_CHANNEL_SELECTED : No Channels have been selected using the selectADC[ ] array

ADC_STANDBY : ADC operating in Standby Mode. Cannot initiate conversions.

int readADCChannels (void);

Reads the selected and configured ADC Channels, results are stored in the array readADC[ ]

Input Arguments none

Returns none

Warnings none

Errors

NO_CHANNEL_SELECTED : No Channels have been selected using the selectADC[ ] array

INTERRUPT FUNCTIONS

int enableExternalInterrupts ( uint32_t extInterrupts );

Enable selected external interrupts

Input Arguments

uint32_t extInterrupts

Combination ( x|y ) of, or in isolation

EXT_INT : Ext_Int interrupt pin

FUSE : FUSE interrupt

Returns none

Warnings none

Errors none

int enableExtIntWatchdog (void);

Enable Ext_Int as a Watchdog interrupt

Input Arguments none

Returns none

Warnings none

Errors none

int enableEncoderInterrupts ( uint32_t encInterrupts );

Enable selected Encoder Flag Interrupts

Input Arguments

uint32_t encInterrupts

Combination ( x|y ) of, or in isolation

ENCaFLGb : a: Channel (0,1,2,3) ; b: Flag (1,2)

ENC_FLG_ALL : All Encoder Interrupts

Returns none

Warnings none

Errors none

int enableADCInterrupts ( uint32_t adcInterrupts );

Enable selected ADC Interrupts

Input Arguments

uint32_t encInterrupts

Combination ( x|y ) of, or in isolation

ADC_RDY : ADC Ready Interrupt

ADC_EOC : ADC End of Conversion Interrupt

ADC_OVR : ADC FIFO Overrun Interrupt

Returns none

Warnings none

Errors none

int enableCounterInterrupts ( uint32_t ctrInterrupts );

Enable counter Interrupts

Input Arguments

uint32_t ctrInterrupts

CNTR_OUT : Counter Interrupt

WD_INTR : Watchdog Interrupt

Returns none

Warnings none

Errors none

int enableAllInterrupts ( void );

Enable all Q4 Interrupts

Input Arguments none

Returns none

Warnings none

Errors none

int disableExternalInterrupts ( uint32_t extInterrupts );

Disable selected external interrupts

Input Arguments

uint32_t extInterrupts

Combination ( x|y ) of, or in isolation

EXT_INT : Ext_Int interrupt pin

FUSE : FUSE interrupt

Returns none

Warnings none

Errors none

int disableExtIntWatchdog (void);

Disable Ext_Int as a Watchdog interrupt. Ext_Int as an external interrupt is also disabled.

Input Arguments none

Returns none

Warnings none

Errors none

int disableEncoderInterrupts ( uint32_t encInterrupts );

Disable selected Encoder Flag Interrupts

Input Arguments

uint32_t encInterrupts

Combination ( x|y ) of, or in isolation

ENCa_FLGb : a: Channel (0,1,2,3) b: Flag (1,2)

ENC_FLG_ALL : All Encoder Interrupts

Returns none

Warnings none

Errors none

int disableADCInterrupts ( uint32_t adcInterrupts );

Disable selected ADC Interrupts

Input Arguments

uint32_t encInterrupts

Combination ( x|y ) of, or in isolation

ADC_RDY : ADC Ready Interrupt

ADC_EOC : ADC End of Conversion Interrupt

ADC_OVR : ADC FIFO Overrun Interrupt

Returns none

Warnings none

Errors none

int disableCounterInterrupts ( uint32_t ctrInterrupts );

Disable counter Interrupts

Input Arguments

uint32_t ctrInterrupts

CNTR_OUT : Counter Interrupt

WD_INTR : Watchdog Interrupt

Returns none

Warnings none

Errors none

int disableAllInterrupts ( void );

Disable all Q4 Interrupts.

Input Arguments none

Returns none

Warnings none

Errors none

int waitForEnabledInterrupts ( uint32_t msTimeoutPeriod );

Wait for enabled Interrupts for a specified timeout period.

Input Arguments

uint32_t msTimeoutPeriod

timeout in ms, or

INF : Do not timeout the interrupt wait.

Returns

The type of Interrupt which occurred, or <0 if error.

| | |
|---|---|
| EXT_INT_INTR | External interrupts |
| FUSE_INTR | |
| WATCHDOG _INTR | Watchdog |
| CNTR_OUT_INTR | Counter Interrupt |
| ADC_RDY_INTR | ADC Interrupts |
| ADC_OVR_INTR | |
| ADC_EOC_INTR | |
| ENC3_FLG2_INTR | Encoder Interrupts |
| ENC3_FLG1_INTR | |
| ENC2_FLG2_INTR | |
| ENC2_FLG1_INTR | |
| ENC1_FLG2_INTR | |
| ENC1_FLG1_INTR | |
| ENC0_FLG2_INTR | |

ENC0_FLG1_INTR

TIMED_OUT The Interrupt Wait was timed out

Warnings none

Errors

NO_INTERRUPTS_ENABLED : No interrupts have been enabled

NO_SUPERUSER_CAPABILITY : The thread cannot be assigned IO Privity

INT_ATTACH_EVENT_FAILED : The call to InterruptAttachEvent( ) failed

INTR_ERROR : The interrupt type could not be recognised

DAC FUNCTIONS

int enableDACTransparent ( void );

Enable the DAC Transparent Mode.

Input Arguments none

Returns none

Warnings none

Errors none

int disableDACTransparent ( void );

Disable the DAC Transparent Mode.

Input Arguments none

Returns none

Warnings none

Errors none

int writeToDACChannel ( int channelSelect, int modeSelect, float desiredVoltage );

Setup a DAC Channel Output Mode and output an Analog Voltage at that Channel. Out of range inputs are automatically corrected to the range limit closest to them.

Input Arguments

int channelSelect

DAC_CH0 DAC Channels 0-3

DAC_CH1

DAC_CH2

DAC_CH3

int modeSelect

UNI_TEN : Unipolar 10V. Range: 0 to 10 Vdc

BI_FIVE : Bipolar 5V. Range: -5 to 5 Vdc

BI_TEN : Bipolar 10V. Range: -10 to 10 Vdc

float desiredVoltage

Desired output voltage.

Returns none

Warnings none

Errors none

int updateDACChannels ( void );

Update DAC Channel outputs.

Input Arguments none

Returns

Warning, if applicable

Warnings

DAC_TRANS_WARNING : DAC is operating in the Transparent Mode; Channel update unnecessary

Errors none

DIGITAL IO FUNCTIONS

int configureIOAllPins ( uint32_t pinDirection, uint32_t outputValue );

Configure all Digital IO pins simultaneously.

Input Arguments

uint32_t pinDirection

ALL_INPUT : All pins configured as Inputs

ALL_OUTPUT : All pins configured as Outputs

bit mask : A 32 bit mask. The lowest 16 bits correspond to the 16 IO pins. 0 - Input, 1 - Output

uint32_t outputValue

Applies to Output pins. Input pins are unaffected by this value assignment.

ALL_HIGH : All output pins are HIGH

ALL_LOW : All output pins are LOW

bit mask : A 32 bit mask. The lowest 16 bits correspond to the 16 IO pins. 0 - LOW (0 Vdc), 1 - HIGH (5 Vdc)

Returns none

Warnings none

Errors none

int setDigIOPinDirection ( int selectPin, int selectIODirection );

Configure the selected Digital IO pin as Input or Output.

Input Arguments

int select Pin

0-15 : The selected pin number.

int selectDirection

INPUT

OUTPUT

Returns none

Warnings none

Errors

INVALID_PIN_NUMBER Pin number is out of range

int writeDigIOPinValue ( int selectPin, int selectValue );

Write a value to an Output Pin.

Input Arguments

int selectPin

0-15 : The selected pin number

int selectValue

LOW : 0 Vdc

HIGH : 5 Vdc

Returns

Warning, if applicable.

Warnings

INPUT_PIN_WARNING : Selected pin is an Input pin. Writing to it has no effect on pin value.

Errors

INVALID_PIN_NUMBER Pin number is out of range

int readDigIOPinValue ( int selectPin );

Read value at selected Input Pin

Input Arguments

int selectPin

0-15 : The selected pin number

Returns

HIGH : Pin value is 5 Vdc

LOW : Pin value is 0 Vdc

Warning, if applicable

Warnings

OUTPUT_PIN_WARNING : Selected pin is an output pin. Reading will return output value to pin.

Errors

INVALID_PIN_NUMBER Pin number is out of range

COUNTER/ TIMER FUNCTIONS

int initCounters ( int selectCounters );

Set selected counter Preload Registers to zero; both Write Sets are reset.

Input Arguments

int selectCounters

COUNTER : Select Counter

WATCHDOG : Select Watchdog

ALL_COUNTERS : Both counters are selected

Returns none

Warnings none

Errors none

int setCounterMode ( int selectCounterType, int selectCounterMode );

Select one or both of the 32 bit counters on the Q4 and program their mode of operation.

Input Arguments

int selectCounterType

COUNTER : Select Counter

WATCHDOG : Select Watchdog

ALL_COUNTERS : Both counters are selected

int selectCounterMode

SQUARE : Set Square Wave mode for counter

PWM : Set Pulse Width Modulation mode for counter

Returns none

Warnings none

Errors none

int setSqWavePeriod ( int selectCounterType, uint32_t sqPeriod, int selectPreloadReg );

Set count period for counter operating in square wave mode.

Input Arguments

int selectCounterType

COUNTER : Select Counter

WATCHDOG : Select Watchdog

ALL_COUNTERS : Both counters are selected

uint32_t sqPeriod

Square Wave period in ms.

int selectPreloadReg

PRE_LOW : Load count into Preload Low Register

PRE_HIGH : Load count into Preload High Register

Returns

Warning, if applicable

Warnings

COUNTER_PWM_WARNING : PWM Mode has been selected for the Counter

WATCHDOG_PWM_WARNING : PWM Mode has been selected for the Watchdog counter

Errors

PERIOD_OUT_OF_LIMITS : period is out of limits (has to be less than 0xFFFFFFFF)

int setPWMPeriod ( int selectCounterType, uint32_t pwmLowPeriod, uint32_t pwmHighPeriod );

Set count period for counter operating in the PWM mode

Input Arguments

int selectCounterType

COUNTER : Select Counter

WATCHDOG : Select Watchdog

ALL_COUNTERS : Both counters are selected

uint32_t pwmLowPeriod

Period for which PWM output is LOW, in ms.

uint32_t pwmHighPeriod

Period for which PWM output is HIGH, in ms.

Returns

Warning, if applicable

Warnings

COUNTER_SQ_WARNING : SQUARE Mode has been selected for the Counter

WATCHDOG_SQ_WARNING : SQUARE Mode has been selected for the Watchdog counter

Errors

PERIOD_OUT_OF_LIMITS : period is out of limits (has to be less than 0xFFFFFFFF)

int setCounterRWReg ( int selectCounterType, int selectReadSet, int selectWriteSet);

Select Read and Write Register sets for the counter.

Input Arguments

int selectCounterType

COUNTER : Select Counter

WATCHDOG : Select Watchdog counter

int selectReadSet

SET0

SET1

int selectWriteSet

SET0

SET1

Returns none

Warnings none

Errors none

int startCounter ( int selectCounterType, int outputPinEnable, int outputPinVal, int ADCtag );

Start the selected counter. The output can be seen at an external pin, if enabled.

Input Arguments

int selectCounterType

COUNTER : Select Counter

WATCHDOG : Select Watchdog

ALL_COUNTERS : Both counters are selected

int outputPinEnable

ENAB_OUT : Enable output at pin corresponding to the selected counter

0 : Do not enable output pin

int outputPinVal

LOW : Current output at output pin. Preload Low is used for first load.

HIGH : Current output at output pin. Preload High is used for first load.

int ADCtag

ADC_COUNTER : Counter is being used for ADC Conversions.

0 : Counter is not being used for ADC.

Returns none

Warnings none

Errors none

int stopCounter ( int selectCounterType, int ADCtag );

Stop the selected counter.

Input Arguments

int selectCounterType

COUNTER : Select Counter

WATCHDOG : Select Watchdog

ALL_COUNTERS : Both counters are selected

int ADCtag

ADC_COUNTER : Counter is being used for ADC Conversions.

0 : Counter is not being used for ADC.

Returns none

Warnings none

Errors none

int setupWDTimer ( uint32_t wdTimerPeriod );

Setup Watchdog counter for operation in the Watchdog Timer mode. Enable the WATCHDOG interrupt.

Input Arguments

int wdTimerPeriod

Timer period in ms.

Returns none

Warnings none

Errors

PERIOD_OUT_OF_LIMITS : period is out of limits (has to be less than 0xFFFFFFFF)

int startWDTimer ( void );

Start the WD Timer.

Input Arguments none

Returns none

Warnings none

Errors

WATCHDOG_NOT_ACTIVATED : Watchdog feature has not been activated

int disableWDTimer (void);

Disable the Watchdog Timer and reset the WATCHDOG interrupt.

Input Arguments none

Returns none

Warnings none

Errors none

ENCODER FUNCTIONS

uint8_t readEncStatusByte ( int selectEncChannel );

Read the 8 bit Status Byte for the selected Encoder Channel

Input Arguments

int selectEncChannel

ENC_CH0 Encoder Channels 0-3

ENC_CH1

ENC_CH2

ENC_CH3

Returns

8-bit Encoder Status Byte

Warnings none

Errors

FATAL_INIT_ERROR_READENC : Fatal Card Initialization error. Since function returns uint8_t, the value of the error is 128. For normal Encoder Status

Byte Reads, bit 7 is always 0; whereas here, 128 is returned, since it is impossible to get in normal Encoder Read operations.

int setupEncPreloadReg ( int selectEncChannel , uint32_t channelCount ) ;

Setup Preload or Limit count for a Encoder Channel's Preload Register. The count has to be 24 bit, or less than 1677216 in the decimal system.

Input Arguments

int selectEncChannel

ENC_CH0 Encoder Channels 0-3

ENC_CH1

ENC_CH2

ENC_CH3

uint32_t channelCount

The preload or upper limit value.

Returns none

Warnings none

Errors

COUNT_OUT_OF_RANGE : Count exceeds the 24 bit limit 0x00FFFFFF

int setupEncChannels ( int channelCombination, uint32_t countMode, uint32_t quadrature, uint32_t encFlags, uint32_t indexMode );

Configur e Encoder Channels for operation.

Input Arguments

int channelCombination

ENC_ALL : Program all Channels

ENC_ODD : Program Odd numbered Channels only

ENC_EVEN : Program Even numbered Channels only

uint32_t countMode

NORMAL : Encoder counters in Normal Mode

RANGE_LIMIT : Encoder counters in Range Limit Mode

NON_RECYCLE : Encoder counters in Non Recycle Mode

MODULE_N : Encoder counters in Module N Mode

uint32_t quadrature

NO_QUAD : Inputs A and B treated as CNT and DIR inputs

QUAD_1X : A and B as 1X quad

QUAD_2X : A and B as 2X quad

QUAD_4X : A and B as 4X quad

uint32_t encFlags

The Encoder FLG combination (FLG1_FLG2) for use in the Interrupt Enable Register

CARRY_BORROW : Carry and Borrow

COMP_BORROW : Compare and Borrow

CB_UPDOWN : Carry/Borrow and Up/Down

INDEX_ERROR : Index and Error

uint32_t indexMode

ENAB_INDEX : Enable Index input

DISAB_INDEX : Disable Index input

Returns none

Warnings none

Errors none

int readEncChannels ( void );

Read Channels selected using setupEncChannels( ). Read values are assigned to elements of encChannelVal[ ], the index of the array element corresponding to Channel number.

Input Arguments none

Returns none

Warnings none

Errors none

int setupEncSingleChannel ( int selectChannel, uint32_t countMode, uint32_t quadrature, uint32_t encFlags, uint32_t indexMode );

Configure a single Encoder Channel.

Input Arguments

int selectEncChannel

| | |
|---|---|
| ENC_CH0 | Encoder Channels 0-3 |
| ENC_CH1 | |
| ENC_CH2 | |
| ENC_CH3 | |

uint32_t countMode

| | | |
|---|---|---|
| NORMAL | : | Encoder counters in Normal Mode |
| RANGE_LIMIT | : | Encoder counters in Range Limit Mode |
| NON_RECYCLE | : | Encoder counters in Non Recycle Mode |
| MODULE_N | : | Encoder counters in Module N Mode |

uint32_t quadrature

| | | |
|---|---|---|
| NO_QUAD | : | Inputs A and B treated as CNT and DIR inputs |
| QUAD_1X | : | A and B as 1X quad |
| QUAD_2X | : | A and B as 2X quad |

QUAD_4X : A and B as 4X quad

uint32_t encFlags

The Encoder FLG combination (FLG2_FLG1) for use in the Interrupt Enable Register

CARRY_BORROW : Carry and Borrow

COMP_BORROW : Compare and Borrow

CB_UPDOWN : Carry/Borrow and Up/Down

INDEX_ERROR : Index and Error

uint32_t indexMode

ENAB_INDEX : Enable Index input

DISAB_INDEX : Disable Index input

Returns none

Warnings none

Errors none

int readEncSingleChannel ( void );

Read Channel selected using setupEncSingleChannel( ). Read value is assigned to an element of encChannelVal[ ], the index of the array element corresponding to Channel number.

Input Arguments none

Returns none

Warnings none

Errors none

int enableEncIdxPulse ( uint32_t selectChannels );

Enable Index pulse for selected Encoder Channels.

Input Arguments

uint32_t selectChannels

Combination ( x|y ) of, or in isolation

| | |
|---|---|
| ENC3_IDX | Encoder Channel Selection |
| ENC2_IDX | |
| ENC1_IDX | |
| ENC0_IDX | |
| ENC_IDX_ALL | Enable All Channels |

Returns none

Warnings none

Errors none

int disableEncIdxPulse ( uint32_t selectChannels );

Disable Index pulse for selected Encoder Channels.

Input Arguments

uint32_t selectChannels

Combination ( x|y ) of, or in isolation

| | |
|---|---|
| ENC3_IDX | Encoder Channel Selection |
| ENC2_IDX | |
| ENC1_IDX | |
| ENC0_IDX | |
| ENC_IDX_ALL | Disable All Channels |

Returns none

Warnings none

Errors none

## Appendix B

## Class QuanserQ4 Example Program

This program illustrates the use of the Q4 Driver Class QuanserQ4 in a standalone C++ Program. The operation selected for testing is setting DAC Output Value for one or more Channels and reading the same value from one or more ADC Channels. The DAC and ADC Channels are connected using RCA Connectors on the Q4 Terminal Board.

```
// Includes
#include "QuanserQ4.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;
int main()
{
// Set up the Q4 on the system for use
     QuanserQ4 first;
// Local Variables
     int interruptType;
     int chSelect;
     float channelValue;
     uint32_t valueIEReg;
//   Write Output Value to one or more DAC Channels
     first.enableDACTransparent();
     first.writeToDACChannel( DAC_CH0, UNI_TEN, 7.5);
     first.writeToDACChannel( DAC_CH1, BI_TEN, -2.5);
```

```
      first.updateDACChannels();
// Select Interrupts to be enabled
      first.disableAllInterrupts();
      first.enableADCInterrupts(ADC_RDY);
      first.enableExternalInterrupts(EXT_INT);
// Test an inline function by reading the Interrupt Enable
Register
      valueIEReg = first.readInterruptEnable();
      fprintf(stdout,"\n\n IE Register Value = %lx \n",
valueIEReg);
// Select ADC Channels for Conversion
      first.selectADC[0] = 10;   // Select ADC Channels
      first.selectADC[1] = 20;
      first.selectADC[2] = 0;
      first.selectADC[3] = 0;
// Setup ADC Channels and Start Conversion
      first.setupADCChannels (ADC_MANUAL);
      first.startADConversion();
// Wait for one of the Enabled Interrupts to occur
interruptType = first.waitForEnabledInterrupts(15000);
// Identify Interrupt Type
      if (interruptType == TIMED_OUT)    fprintf(stdout, "\n\n
      Interrupt Wait timed out! \n");
      else if (interruptType == ADC_RDY_INTR)     fprintf(stdout,
      "\n\n ADC_RDY \n");
```

```
      else if (interruptType == EXT_INT_INTR)     fprintf(stdout,
      "\n\n EXT_INT \n");
// Read ADC Channel Values into the readADC[] Array
      first.readADCChannels();
// Read back ADC Channel Values
      fprintf(stdout,"\n\n    Channel    0    value:    %f    \n",
first.readADC[0]);
      fprintf(stdout,"\n\n    Channel    1    value:    %f    \n",
first.readADC[1]);
      return(0);
}
```

# Appendix C

## DAC Channel Output Calculation Formula

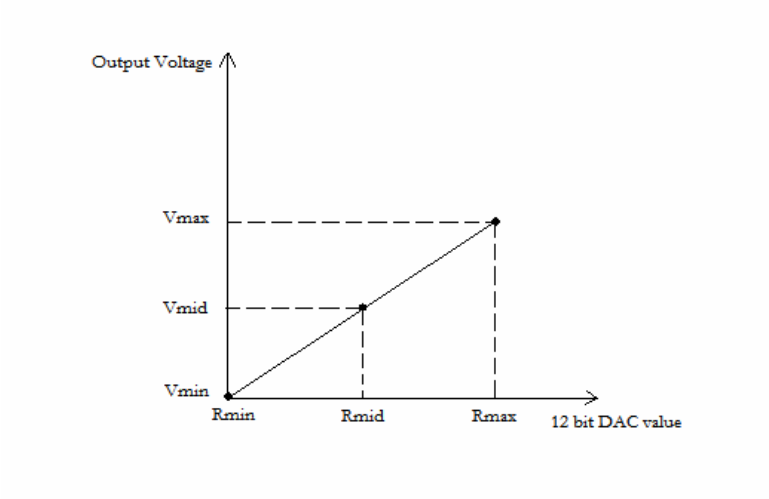


Figure C.1 Linear Relationship between Register Input and DAC Channel Output

It is desired to obtain a formula for the input to be given to a DAC Channel Output Register when a certain voltage is to appear at the Channel Output. Input-Output relationship has been tabulated in the Q4 User's Guide in the section on DAC Registers. This formula uses the fact that the input-output relationship is linear, and that three points on the line are known, which are sufficient for finding any other abscissa given an ordinate.

The Figure (C.1) shows the line representing the relationship, with Vmin, Vmid and Vmax being the output voltage values given minimum (Rmin), intermediate (Rmid) and maximum (Rmax) values for a 12 bit input.

Using general formula for a line

$$y = mx + c$$

From the figure

$$c = Vmin$$

Slope Calculation

$$m = (Vmax - Vmin) / (Rmax)$$

Given any desired Output Voltage (y), corresponding DAC input (x)

$$x = (y - Vmin) / m$$

# Appendix D

## Simulation Layout

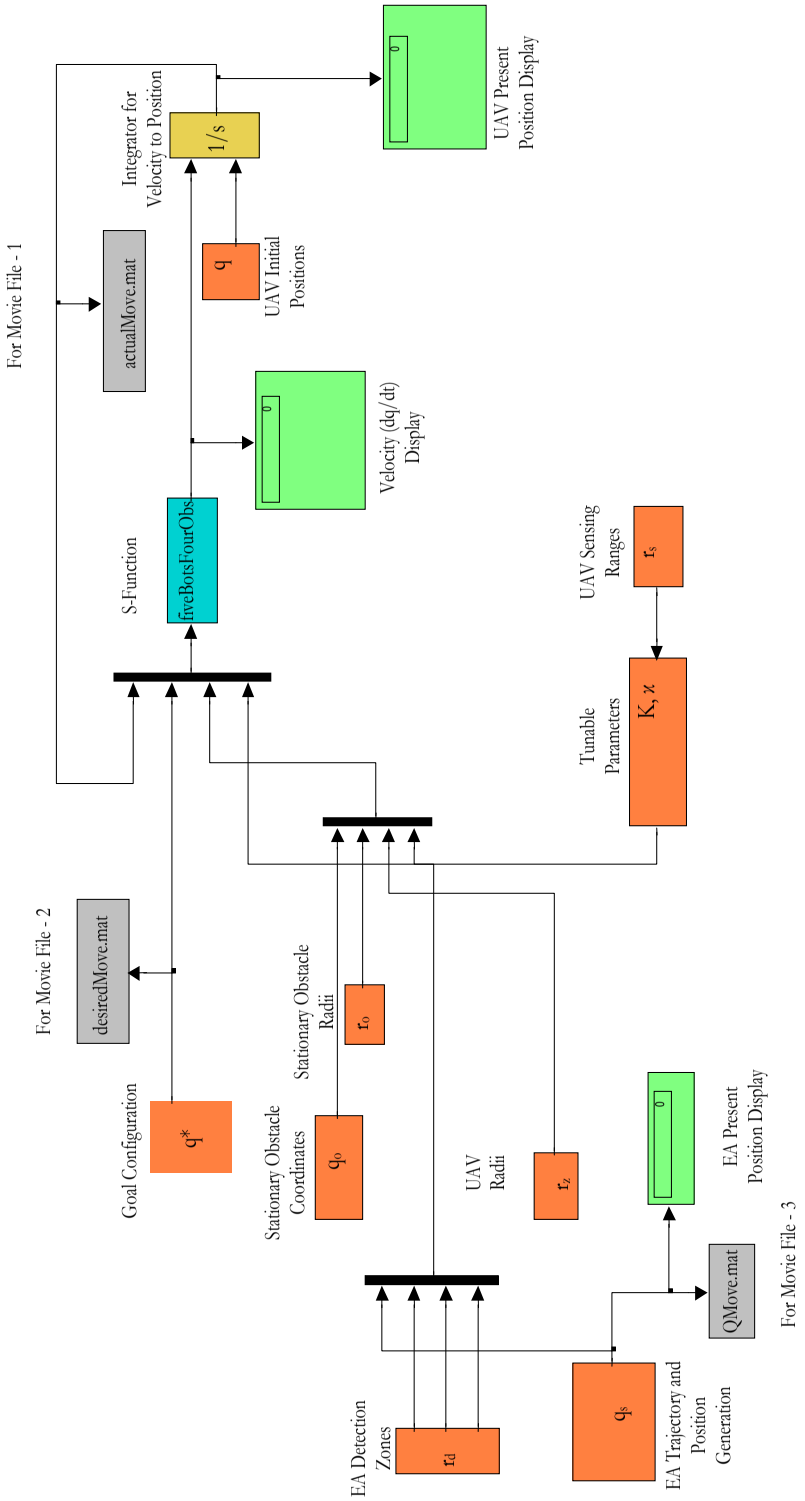


Figure D.1. Multiple UAV Simulation – Simulink Model Layout

REFERENCES


[1] J. Barraquand, B. Langlois, and J. C. Latombe, ``Numerical Potential Fields Techniques for Robot Path Planning," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 22, pp. 224-241, (1992).

[2] J. Barraquand and J. C. Latombe, ``A Monte-Carlo Algorithm for Path Planning with Many Degrees of Freedom," *Proceedings of the IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, May 1990, pp. 584-589.

[3] A. Bemporad, A. De Luca, and G. Oriolo, ``Local Incremental Planning for a Car-Like Robot Navigating Among Obstacles," *Proceedings of the IEEE International Conference on Robotics and Automation*, Minneapolis, Minnesota, April 1996, pp. 1205-1211.

[4] D. E. Chang, S. Shadden, J. Marsden, and R. Olfati-Saber, ``Collision Avoidance for Multiple Agent Systems," *Proc. of the 42th IEEE Conference on Decision and Control*, Maui, Hawaii USA, Dec.2003, pp. 539 - 543.

[5] J. Chen, D. M. Dawson, M. Salah, and N. Pradhan, ``Multiple UAV Navigation with Finite Sensor Range," Clemson University CRB Technical Report, CU/CRB/3/3/05/#2, http://www.ces.clemson.edu/ece/crb/publictn/tr.htm , March 2005.

[6] C. I. Connolly, J. B. Burns, and R. Weiss, ``Path Planning Using Laplace's Equation," *Proceedings of the IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, May 1990, pp. 2102-2106.

[7] M.S. de Queiroz, D.M. Dawson, S.P. Nagarkatti, and F. Zhang, *Lyapunov-Based Control of Mechanical Systems*, Birkhäuser, 1999.

[8] D. V. Dimarogonas, M. M. Zavlanos, S. G. Loizou, and K. J. Kyriakopoulos, ``Decentralized Motion Control of Multiple Holonomic Agents under Input Constraints," *Proc. of the 42th IEEE Conference on Decision and Control*, Maui, Hawaii USA, Dec.2003, pp. 3390 - 3395.

[9] S. S. Ge and Y. J. Cui, ``New Potential Functions for Mobile Robot Path Planning," *IEEE Transactions on Robotics and Automation*, Vol. 16, No. 5, pp. 615-620, (2000).

[10] J. Guldner, V. I. Utkin, H. Hashimoto, and F. Harashima, ``Tracking Gradients of Artificial Potential Field with Non-Holonomic Mobile Robots," *Proc. of the American Control Conference*, Seattle, Washington, June 1995, pp. 2803-2804.

[11] O. Khatib, *Commande dynamique dans l'espace opérational des robots manipulateurs en présence d'obstacles*, Ph.D. Dissertation, École Nationale Supérieure de l'Aéronatique et de l'Espace (ENSAE), France, 1980.

[12] O. Khatib, ``Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *International Journal of Robotics Research*, Vol. 5, No. 1, pp. 90-99, (1986).

[13] D. E. Koditschek, ``Exact Robot Navigation by Means of Potential Functions: Some Topological Considerations," *Proc. of the IEEE International Conference on Robotics and Automation*, Raleigh, North Carolina, May 1987, pp. 1-6.

[14] D. E. Koditschek and E. Rimon, ``Robot Navigation Functions on Manifolds with Boundary," *Adv. Appl. Math.*, Vol. 11, pp. 412-442, (1990).

[15] K. J. Kyriakopoulos, H. G. Tanner, and N. J. Krikelis, ``Navigation of Nonholonomic Vehicles in Complex Environments with Potential Fields and Tracking," *Int. J. Intell. Contr. Syst.*, Vol. 1, No. 4, pp. 487-495, (1996).

[16] J. C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers: Boston, Massachusetts, 1991.

[17] J. P. Laumond, P. E. Jacobs, M. Taix, and R. M. Murray, ``A Motion Planner for Nonholonomic Mobile Robots," *IEEE Transactions on Robotics and Automation*, Vol. 10, No. 5, pp. 577-593, (1994).

[18] S. G. Loizou, and K. J. Kyriakopoulos, ``Closed Loop Navigation for Multiple Holonomic Vehicles," *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, Lausanne, Switzerland, October 2002, pp. 2861-2866.

[19] S. G. Loizou, H. G. Tanner, V. Kumar, and K. J. Kyriakopoulos, ``Closed Loop Motion Planning and Control for Mobile Robots in Uncertain Environments," *Proceedings of the 42th Conference on Decision and Control*, Maui, Hawaii USA, Dec. 2003, pp. 2926-2931.

[20] E. Rimon and D. E. Koditschek, ``Exact Robot Navigation Using Artificial Potential Function," *IEEE Trans. on Robotics and Automation*, Vol. 8, No. 5, pp. 501-518, (1992).

[21] R. Olfati-Saber, ``Flocking for Multi-Agent Dynamic Systems: Algorithms and Theory," Submitted to the *IEEE Transactions On Automatic Control*, June, 2004 (Technical Report CIT-CDS 2004-005).

[22] R. O. Saber and R. M. Murray, ``Flocking with Obstacle Avoidance: Cooperation with Limited Communication in Mobile Networks," *Proc. of the 42$^{th}$ IEEE Conference on Decision and Control*, Maui, Hawaii USA, Dec. 2003, pp. 2022 - 2028.

[23] H. G. Tanner, ``Flocking with Obstacle Avoidance in Switching Networks of Interconnected Vehicles," *Proc. of the IEEE International Conference on Robotics and Automation*, New Orleans, LA, April 2004, pp. 3006-3011.

[24] H. G. Tanner, A. Jadbabaie, and G. J. Pappas, ``Stability of Flocking Motion," Technical Report No MS-CIS-03-03, Department of Computer and Information Science, University of Pennsylvania, January 2003.

[25] H. G. Tanner and K. J. Kyriakopoulos, ``Nonholonomic Motion Planning for Mobile Manipulators," *Proc. of the IEEE International Conference on Robotics and Automation*, San Francisco, California, April 2000, pp. 1233-1238.

[26] H. G. Tanner, S. G. Loizou, and K. J. Kyriakopoulos, ``Nonholonomic Navigation and Control of Cooperating Mobile Manipulators," *IEEE Transactions on Robotics and Automation*, Vol. 19, No. 1, pp. 53-64, (2003).

[27] R. Volpe and P. Khosla, ``Artificial Potential with Elliptical Isopotential Contours for Obstacle Avoidance," *Proc. of the IEEE Conference on Decision and Control*, Los Angeles, California, December 1987, pp. 180-185.

[28] L. L. Whitcomb and D. E. Koditschek, ``Automatic Assembly Planning and Control via Potential Functions," *Proc. of the IEEE/RSJ International Workshop on Intelligent Robots and Systems*, Osaka, Japan, November 1991, pp. 17-23.

[29] Rob Krten, *Getting Started with the QNX Neutrino*, PARSE Software Devices, 1999

[30] Quanser Consulting Inc., *Q4 Data Acquisition System: User's Guide Version 1.0*, 2003.

[31] QNX Developer Support, *QNX Neutrino OS*, QNX Software Systems Ltd., 2003. http://www.qnx.com/developers/docs/momentics621_docs/momentics/index.html

[32] Quality Real Time Systems, LLC, *QMotor 3.0; A PC Based Real-Time Multitasking Graphical Control Environment: A User's Guide*, October 13, 2000.

[32] CRB DoE Project Team, *Programming Guide for Robotics and Mechatronics Group at Clemson University*, 1998-2000.

[33] Alessandro Rubini & Jonathan Corbet, *Linux Device Drivers*, O'Reilly & Associates, Inc., 2001.

[34] Dave Donohoe, "*Talking to Hardware under the QNX Neutrino*", http://www.qnx.com/developers/articles/article_304_2.html

[35] Eduard Kromskoy, RTC Program, 2002 ftp://ftp.qnx.org.ru/pub/projects/ed1k/clock.tgz

[36] QNX Developer Support, "Writing an Interrupt Handler", http://www.qnx.com/developers/docs/momentics_nc_docs/neutrino/prog/inthandler.html