Electronic Theses and Dissertations   Theses, Dissertations, and Major Papers

2012

Design of an Engine Test Cell Control System

Anthony Joseph Fountaine
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Design of an Engine Test Cell Control System

by

Anthony Fountaine

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Mechanical, Automotive, and Materials Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

2011

© 2011 Anthony Fountaine

Design of an Engine Test Cell Control System

by

Anthony Fountaine

APPROVED BY:

_____

Dr. Majid Ahmadi, Outside Department Reader

Department of Electrical and Computer Engineering

_____

Dr. Jimi Tjong, Department Reader

Department of Mechanical, Automotive, and Materials Engineering

_____

Dr. Xiang Chen, Advisor

Department of Electrical and Computer Engineering

_____

Dr. Ming Zheng, Advisor

Department of Mechanical, Automotive, and Materials Engineering

_____

Dr. Ronald Barron, Chair of Defence

Department of Mechanical, Automotive, and Materials Engineering

November 30, 2011

LIST OF PUBLICATIONS / PAPERS / PROJECTS

1. Fountaine A., "Dynamic Servo Current Reduction In ABB S3 Gantry To Reduce End Effector Damage", Ford, TR1998-001, 1998.

2. Fountaine A., "Engine Block Conveyor RFID Tag Part Tracking System", Ford, TR1998-002, 1998.

3. Fountaine A., "Ultrasonic Engine Block Measurement for Part Identification", Ford, TR1999-001, 1999.

4. Fountaine A., "EDDI Logic PLC Conversion To Run On Siemens S5 95U For C4 Conveyor", Ford, TR1999-002, 1999.

5. Fountaine A., "Engine Block Riser Rough Dimensioning Using DVT Vision System", Ford, TR1999-003, 1999.

6. Fountaine A., "Engine Block Cubing Line Part Identification Using DVT Vision System", Ford, TR2000-001, 2000.

7. Fountaine A., "Body and Assembly Web Based Returns Tracking and Reporting System", Ford, TR2001-001, 2001.

8. Fountaine A., "LMS UPA NVH Processing Application", Ford, TR2002-001, 2002.

9. Fountaine A., "Encoder Based Engine Vibration Sampling And Variance Analysis Application", Ford, TR2002-002, 2002.

10. Fountaine A., "Camshaft Signal Resampling For Engine Vibration Variance Analysis Application", Ford, TR2003-001, 2003.

11. Fountaine A., "HP Standard Data File Format ATL COM Object", Ford, TR2003-002, 2003.

12. Fountaine A., "Using The NVH UPA COM Object", Ford, TR2004-057, 2004.

13. Fountaine A., "One Third Octave Viewer Utility", Ford, TR2004-056, 2004.

14. Fountaine A., "Scripting For Robot Control And Data Acquisition", AMTC, London Ontario, 2004.

15. Fountaine A., "Inline Test Station Application For Engine Air Intake Flutter Detection", MarkIV Automotive Inc., TR2004-001, 2004.

16. Fountaine A., "NVH Color Map Application For Engine Baseline Mapping", Ford, TR2005-028, 2005.

17. Fountaine A., "Realtime Cylinder Pressure Analysis Guide", University Of Windsor, 2005.

18. Fountaine A., "Realtime Engine Monitoring And Control Using LabVIEW FPGA", University Of Windsor, 2005.

19. Fountaine A., "Using DLLs With LabVIEW Real-Time Applications", Ford, TR2006-011, 2006.

20. Fountaine A., "Real Time Heat Release Analysis", University Of Windsor, Prepared Thesis, 2006.

21. Fountaine A., Kim S., "Embedded EGR PID Controller With CAN Bus Valve Position Output", Ford, TR2006-007, 2006.

22. Fountaine A., Fountaine S., "Web Based Ordering And Part Tracking System", Ford, TR2006-001, 2006.

23. Fountaine A., Kim S., "Embedded Urea Injector PWM Controller For Diesel After Treatment Testing", Ford, TR2007-001, 2007.

24. Fountaine A., Kim S., "Electronic Pedal Simulator Using Analog Devices ADUC7020", Ford, TR2007-002, 2007.

25. Fountaine A., Dilaudo N., "Hydraulic Proportional Valve Controller For Crankshaft Thrust Bearing Testing Using Delta Computer RCM75 PID Controller", Ford, TR2007-003, 2007.

26. Fountaine A., "ETAS INCA COM Automation For Engine Test Cell Integration", Ford, TR2007-004, 2007.

27. Fountaine A., "MDF Measurement Data File Reader", http://sourceforge.net/ projects/mdfdatafile, 2008.

28. Fountaine A., "Embedded Wind Speed And Direction Monitoring For Tracking Environmental Concerns", Ford, TR2008-001, 2008.

29. Fountaine A., "Dynamometer Bearing Monitoring Using ICP Tri-axial Accelerometer", Ford, TR2009-001, 2009.

30. Fountaine A., "Horiba Mexa 9100 HPIB Device Driver", Ford, TR2009-002, 2009.

31. Fountaine A., "California Analytic Gas Analyzer Ethernet Device Driver", Ford, TR2009-003, 2009.

32. Fountaine A., "Portable Throttle Actuator Replacement For Jordan Actuator", Ford, TR2010-001, 2010.

33. Fountaine A., "SQLite Database Time History Data Storage", Ford, TR2010-002, 2010.

34. Fountaine A., Kim S., "Conversion Of A New Hermes Engraver To A CNC Using Linux EMC2", Ford, TR2010-003, 2010.

35. Fountaine A., Kelly C., "Yokogawa WVF File Format COM Object", Ford, TR2011-001, 2011.

36. Fountaine A., "Time History Data Viewer For MDF, ADACS And SQLite File Formats", Ford, TR2011-002, 2011.

37. Fountaine A., "DynoDashBoard Engine Cell Status RIA Web Monitoring Application", Ford, TR2011-003, 2011.

38. Fountaine A., Kim S., "MODBUS TCP/IP Power Monitoring System", Ford, TR2011-004, 2011.

39. Fountaine A., "Horiba Mexa 7000 Gas Analyzer Ethernet Device Driver", Ford, TR2011-005, 2011.

40. Fountaine A., Kelly C., "CAS Phoenix Ethernet Device Driver", Ford, TR2011-006, 2011.

41. Fountaine A., Kelly C. "Cam And Crankshaft Synthesis Using An XMOS Processor", Ford, TR2011-007, 2011.

42. Fountaine A., "Design Of An Engine Test Cell Control System", University Of Windsor, Thesis, 2011.

ABSTRACT

The work contained in this thesis offers an alternative to the commercial solutions available for automatic engine testing. Both the software and hardware needed for the operation of an engine test cell were designed. The hardware used, employs a distributed architecture to move the acquisition devices as close to the sensors as possible. A test sequencer was developed that allows the creation of complex engine tests. The tests are executed on a real time operating system. This operates in conjunction with a graphical user interface, which runs on Windows. A custom data viewer was created to perform test analysis quickly, as well as indentify long term trending in test data. The automated system supports both simple and complex alarming for fault detection. The full solution was installed in three eddy current dynamometer test cells at Ford. It is successfully running twenty four hours a day, seven days a week.

DEDICATION

This work is dedicated to my lovely wife, Soula, and our two boys, Anthony and John. Thank you so much Soula for all your hard work helping me get this done and figuring out a way to take care of the boys at the same time.

Anthony and John, I love you guys. When you are old enough to read this, know that there is one section that your mother forced me to remove. I have put it in a safe place for you guys. Read page 27, it will lead you to the location. I also have a special message for both of you on page 5. You will need to decode it. When you have reached the answer, you will find an envelope. There is an important message in the envelope for both of you. There is also a set of keys in the envelope that will open a safe. Do not lose the contents of the safe and do not give it to anyone but your children. My parents gave this to me and I want you and your children to have it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

a	Acceleration due to gravity ≈ 9.81 at sea level [m/s^2]
AC	Alternating Current
ADACS	Automated Data Acquisition and Control System ©Horiba ATS
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
CPU	Central Processing Unit
DAQ	Data Acquisition
DC	Direct Current
e	Error
EFT	Engine Fatigue Test
F	Force [N]
FTP	File Transfer Protocol
GUI	Graphical User Interface
HMI	Human Machine Interface
HTTP	Hyper Text Transfer Protocol
I/O	Input Output
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
k	Sample Number
K_D	Derivative Gain [s]
K_I	Integral Gain [1/s]
K_P	Proportional Gain [-]

m	Mass [kg]
MDI	Multiple Document Interface
MXI	Multisystem Extension Interface
n	Sample Number
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCM	Powertrain Control Module
PERDC	Powertrain Engineering Research and Development Center
PID	Proportional Integral Derivative
PLC	Programmable Logic Controller
PXI	PCI Extensions for Instrumentation
PV	Process Variable
r	Length [m]
RIA	Rich Internet Application
RMS	Root Mean Squared
rpm	Revolutions Per Minute [1/min]
RTOS	Real Time Operating System
SDK	Software Development Kit
SP	Setpoint
SQL	Structured Query Language
TCP	Transmission Control Protocol
t	Time
τ	Torque [N·m]

T_S	Sample Time
u	PID Control Output Signal [%]
UDP	User Datagram Protocol
WOT	Wide Open Throttle

CHAPTER 1: INTRODUCTION

Automotive manufacturers are under pressure to reduce design and development cycle times for new vehicles. One of the major parts of the development cycle is component testing. Testing is performed to validate the engineering design. With the increased pressure of competition to produce vehicles faster and cheaper, a large portion of testing is often outsourced to an external test facility.

There are a number of reasons the outsourcing takes place. One of the reasons is a lack of internal testing expertise within the automotive companies. As the consumer market fluctuates, companies reduce head count and are slow to replace employees when the market recovers. Another reason is the cost of ownership for the state of the art testing facilities. Beyond the initial purchase cost of the test equipment, there is a high cost to maintain and staff the testing facilities and keep the facilities updated. There are very few companies in the business of supplying engine test control systems. Over the years, the small number of these companies has been reduced even further due to mergers and acquisitions. These handful of companies have built up a trust in their abilities to deliver quality test equipment and software in a timely manner. However, these services are typically associated with very high costs. As these companies merge, they are also under pressure to reduce cost and head count. These reductions put pressure on the suppliers to maintain their level of expertise as well.

The Powertrain Engineering Research and Development Center, or PERDC as it is called, is one such testing facility where engine and transmission research is conducted. The PERDC facility currently consists of nine dynamometer engine test cells. In an effort to keep current in the engine testing world, PERDC is in the process of expanding

its facilities and also retrofitting the existing engine test cells. One of the key costs of retrofitting the testing facility is the cost of the engine test cell control system.

Many vendors of engine test cell control systems use data acquisition hardware that was purchased off the shelf from another vendor. What distinguishes one vendor from another is therefore not the hardware, but the software part of the control system. Designing the software for a modern engine test cell seems to be a daunting task. Martyr and Plint went so far as to say that it was beyond the capabilities of any one person, regardless of their experience in engine testing [1]. This thesis will, in part, aim to challenge this statement by Martyr and Plint.

The basic concepts of what is required to develop an engine test cell control system can be easily applied. The integration of all the components into a complete working package is the goal of this research. The following objectives were set for this thesis:

1. Develop and implement an engine testing control system capable of executing arbitrary automatic test sequences in real time.

2. Provide a flexible and intuitive GUI for the control system.

3. Develop an engine test cell control system that will be cost effective compared to the commercially available systems.

4. Provide engine test data equivalent to the commercially available systems, including tools to analyze the data.

5. Develop a user friendly system that the technicians, engineers and other staff members of PERDC can easily use to perform engine testing.

CHAPTER 2: METHODOLOGY

2.1 Engine Test Cell Overview

An engine test cell is comprised of a large number of inter-related components. The two primary components are the engine and the dynamometer. Each of the components requires several key inputs to produce the desired outputs. The overall inputs and outputs of the test cell include physical matter such as fluids and air, energy sources such as fuel and electricity, and data inputs and outputs. The flow of the system inputs and outputs is shown pictorially in Figure 2.1.
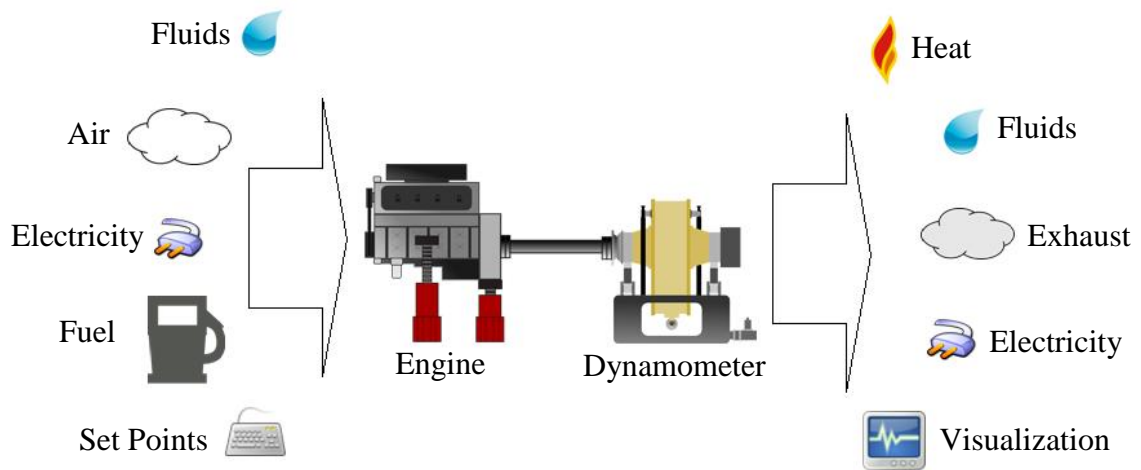
Figure 2.1: Test Cell Inputs and Outputs.

The test cells that were modified as part of this project, have an existing infrastructure to provide the needed inputs and dispose of the test by-products. There is a fuel panel, proper air handling and ventilation units, process cooling water, engine exhaust treatment, and a source of electricity. A full explanation of what these systems are and the recommended best practices are discussed in Martyr and Plint [1].

This thesis is primarily concerned with the details of controlling, monitoring, and acquiring data from the engine test cell. The engine being tested will typically be under the control of a production powertrain control module (PCM). The designed control system will provide a variable pedal position input to the engine. This enables control of the engine load. Attached to the engine, in the present configuration, is an eddy current dynamometer.

The eddy current dynamometer is a dynamic braking device that will be used to provide a load for the engine and to control the engine speed. The dynamometer is used to simulate the conditions an engine would undergo when installed in a vehicle. Eddy current dynamometers do not have motoring capability. The ability to motor an engine implies that a dynamometer can rotate an engine that is not powered. AC and DC dynamometers have this ability. An additional starter motor connected to the rear of the eddy current dynamometer is used during engine cranking. Heat exchangers and process control valves are used to maintain engine oil and coolant operating temperatures. This arrangement is depicted in Figure 2.2.
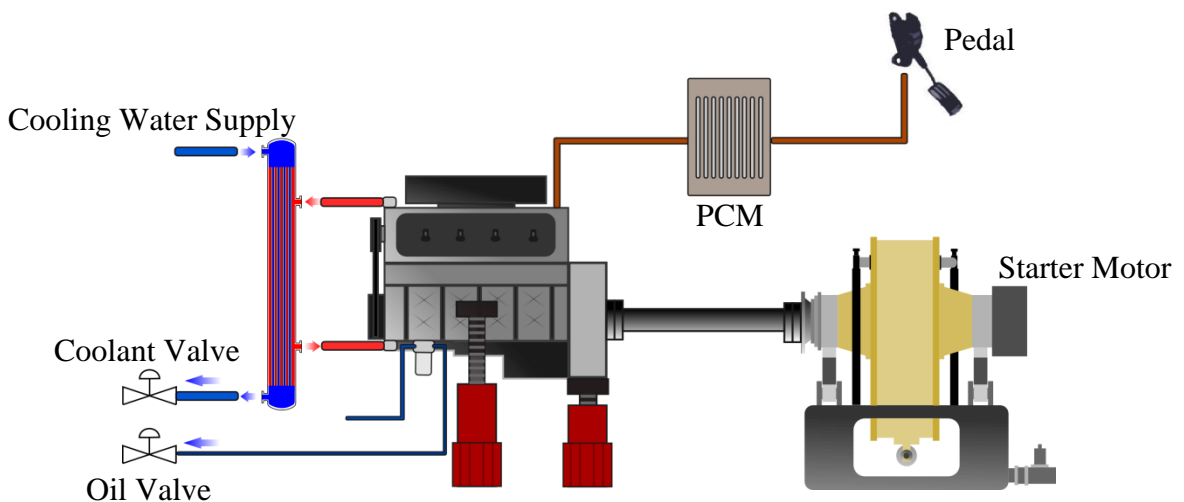
Figure 2.2: Controlled Devices.

2.1.1 Eddy Current Dynamometer

An eddy current dynamometer is used to apply a braking torque to the engine being tested and to control its speed. A braking torque is produced by the rotation of the dynamometer's rotor in the presence of a magnetic field [2]. The magnetic field is generated by a DC current flowing through the exciting coil of the dynamometer. An electrically conductive material, such as iron, copper or aluminum, is used for the rotor construction. As the rotor rotates through the magnetic field, eddy currents will be induced in it. The flow of eddy currents in the rotor, creates a magnetic field in a direction that opposes the rotational force and supplies the needed braking force. The eddy current dynamometer and controller signal flow diagram is shown in Figure 2.3.
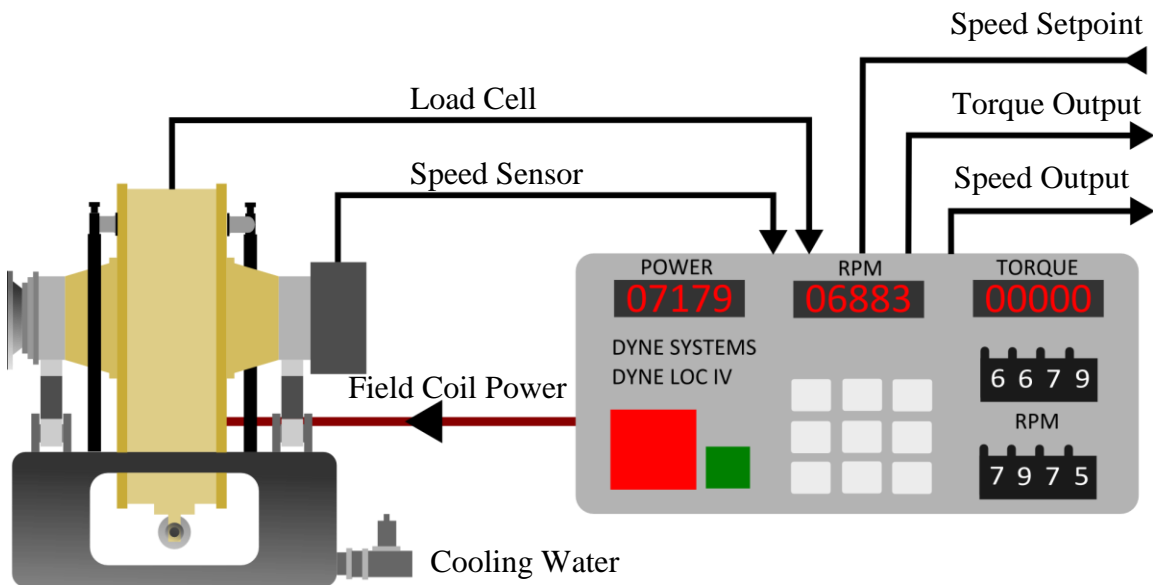
Figure 2.3: Dynamometer Connections.

Eddy current dynamometers convert the mechanical energy, created by the engine, into heat. A large source of cooling water is required to dissipate the heat

generated. There is both a pressure and flow switch to ensure that the cooling water is flowing at all times. If the cooling water is not flowing, even for a brief period of time, the rotor could overheat and deform. Both the pressure and the flow switch are monitored by the engine control system.

The eddy current dynamometer housing is mechanically anchored at a single point of contact. If the contact point is removed, the dynamometer housing would rotate freely. This contact point is made through a load cell. The load cell is able to measure the magnitude and the direction of the force or torque imposed on the dynamometer housing. This is the most crucial measurement in the engine test setup [1].

The load cell output is calibrated to measure torque using a pair of certified calibration levers and masses. This is shown in Figure 2.4 where two equal length levers and identical weight trays are installed on the dynamometer housing. This provides perfect balance and is used to calibrate the zero torque measurement.
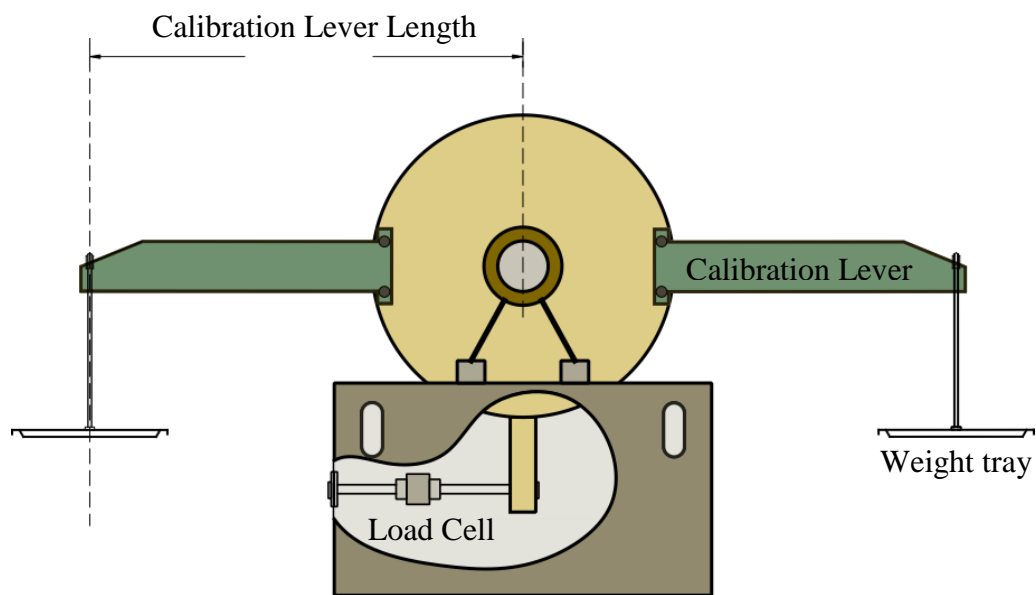
Figure 2.4: Eddy Current Dynamometer.

By applying a mass to one of the weight trays, the force applied to the load cell becomes unbalanced resulting in a net output. The force applied to the calibration lever is shown in equation (2.1).

$$F = ma \tag{2.1}$$

The net torque, τ, on the dynamometer, from the mass, is given in equation (2.2) where r is the calibration lever length [3].

$$\tau = r \times F \tag{2.2}$$

Using the calibrated masses and lever arms, a known value of torque can be applied to the dynamometer. The output value from the load cell is then calibrated to this known value of torque in the engine control system software. Typically, the length of the calibration lever arms are adjusted by the manufacturer so that the torque applied to the dynamometer is an integer multiple of the mass applied.

Internal to the load cell is a set of four strain gauges in a bridge configuration. The impedance of the bridge is typically 350 Ω. The load cell will normally have a rated output of mV/V. If the rating is 3mV/V, and 10V is used for the supply, the full scale output would be 30mV [4].
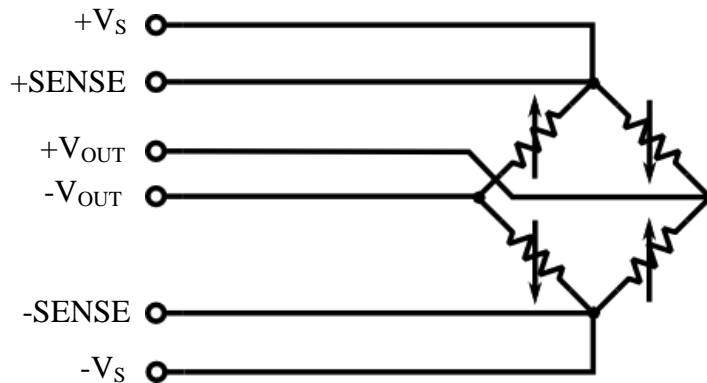
Figure 2.5: Load Cell Strain Gauges.

7

Six wires are normally connected to a quality load cell. The two extra wires in Figure 2.5, labelled "SENSE", are for the signal conditioner to measure the supply voltage applied to the sensor. This accounts for any drop in the supply voltage along the wires from the conditioner to the load cell. Either AC or DC supply voltages can be used.

In order to measure the output power of the engine, an additional sensor is required to measure speed. An example sensor would be a non contact magnetic pickup used in conjunction with a 60 toothed wheel [5]. This is shown in Figure 2.6.
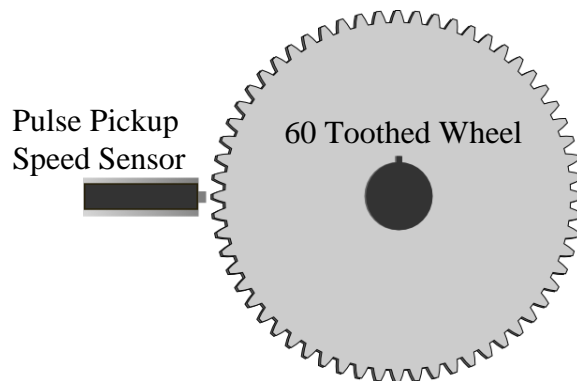
Figure 2.6: Dynamometer Speed Pickup.

The speed sensor detects the teeth on the wheel as it rotates. The output signal from the speed sensor is an input to the dynamometer controller. For every revolution, 60 teeth are counted by the dynamometer controller. The time elapsed while counting the 60 teeth is used to calculate the engine speed.

The specifications for the eddy current dynamometer used to collect data for this thesis are given in Table 2.1 and the torque diagram is shown in Figure 2.7. This information was reproduced from [2].

Table 2.1: Horiba WTS470 Eddy Current Dynamometer Specifications.

	Unit	Value
Power	kW	470
Rated Torque	N·m	2400
Minimum Speed for Rated Torque	rpm	1000
Maximum Speed	rpm	7000
Minimum Speed for Maximum Power	rpm	1870
Moment of Inertia	Kg·m^2	2.06
Maximum Excitation current	A	10
Weight	kg	1350
Measuring Accuracy of Speed	rpm	+-0.025% of Max
Measuring Accuracy of Torque	%	+- 0.2% of Max
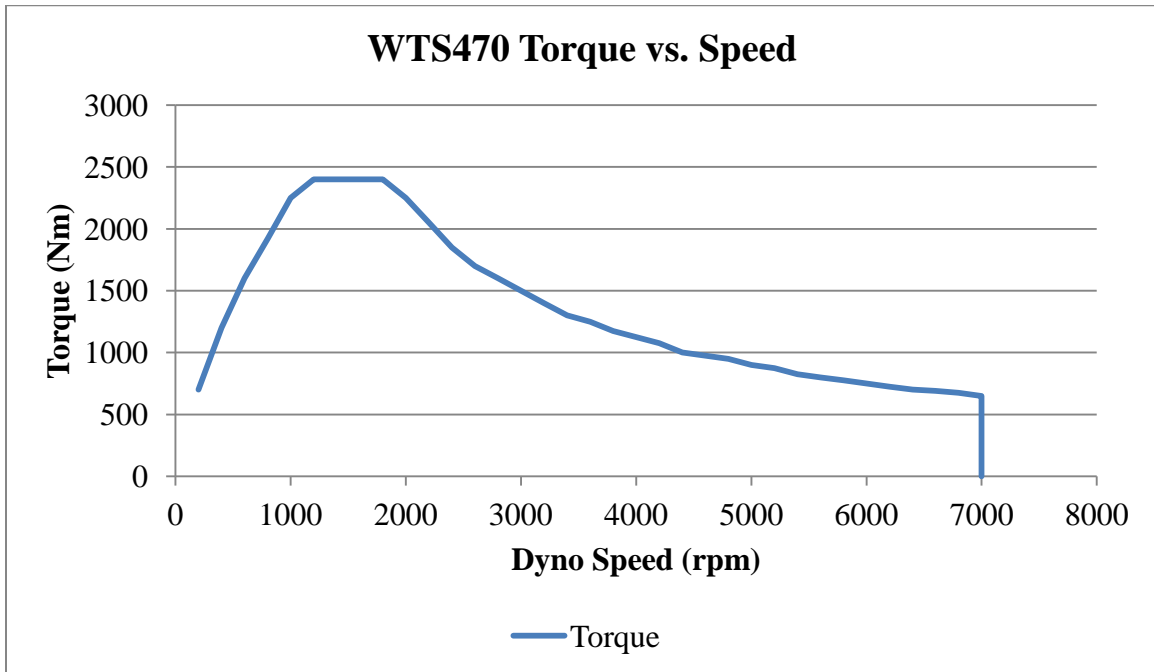
Figure 2.7: WTS470 Dynamometer Torque Specification.

9

2.1.2 Thermocouple, Pressure and Vibration Sensors

Thermocouples are widely used in engine testing for measuring temperature. They are durable, work over a large temperature range and are inexpensive.

A thermocouple is simply two dissimilar metal wire segments that are connected together at one end, as seen in Figure 2.8. If a temperature difference exists between the open (cold junction) and connected (hot junction) ends of the thermocouple, a voltage can be measured between the two wires on the open end. This is called the Seebeck or thermoelectric effect [6]. The voltage is proportional to the difference in temperature between the two ends. The relationship between temperature difference and voltage is nonlinear. There are lookup tables or polynomial equations that provide a method to convert the voltage measured to a temperature difference. For a K type thermocouple, the voltage ranges from 0V for 0 degrees to 54mV at 1370 degrees Celsius difference [6].
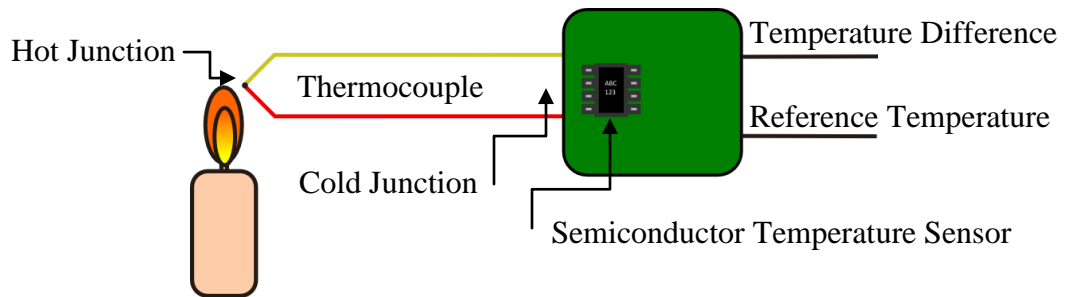
Figure 2.8: Thermocouple Measurement.

In order to convert the voltage to an absolute temperature, a cold junction temperature reference is used. This was previously done using an ice bath for the cold junction. Today, a semiconductor reference temperature sensor is typically used, as seen in Figure 2.8 [6]. This process is termed cold junction temperature compensation.

A thermocouple would normally be measured using a differential analog input. This allows for common mode voltage rejection. Since the thermoelectric voltage is at such a low level, thermocouples are susceptible to electrical noise. If the noise is common mode, it can be removed with a differential amplifier. In addition, a differential input amplifier circuit provides the ability to use a broken thermocouple detection circuit. One method to do this is shown in Figure 2.9 [7].
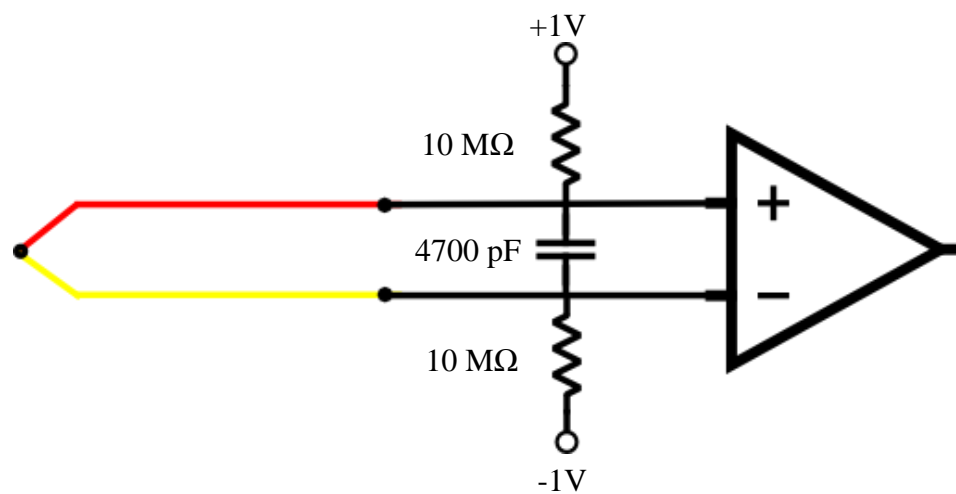
Figure 2.9: Open Circuit Thermocouple Detection.

The thermocouple, connected to the circuit in Figure 2.9, will short out the capacitor which results in a common mode voltage on both inputs of the amplifier. If one wire of the thermocouple breaks, the capacitor will charge and a large voltage will be present at the input of the differential amplifier [7]. Broken wire detection is important in engine testing. There is a significant amount of vibration during the testing of engines which causes thermocouples to break frequently. Without broken wire detection, the input would simply float at some unknown value.

The pressure sensor is equally as important as the thermocouple. It is often found that the two sensors are used together. For example, to monitor the engine coolant, both temperature and pressure sensors are used. Pressure sensors used for measuring static pressures are categorized into four types dependant on the reference pressure the measurement is being compared to as discussed by Wilson [4]:

1. Gauge pressure sensors measure pressure relative to atmospheric pressure.

2. Absolute pressure sensors measure a pressure value that is referenced to a perfect vacuum.

3. Differential pressure sensors measure the difference between two pressures.

4. Vacuum gauge pressure sensors measure vacuum pressures relative to atmospheric pressure.

Each of these pressure sensor types is shown pictorially in Figure 2.10.
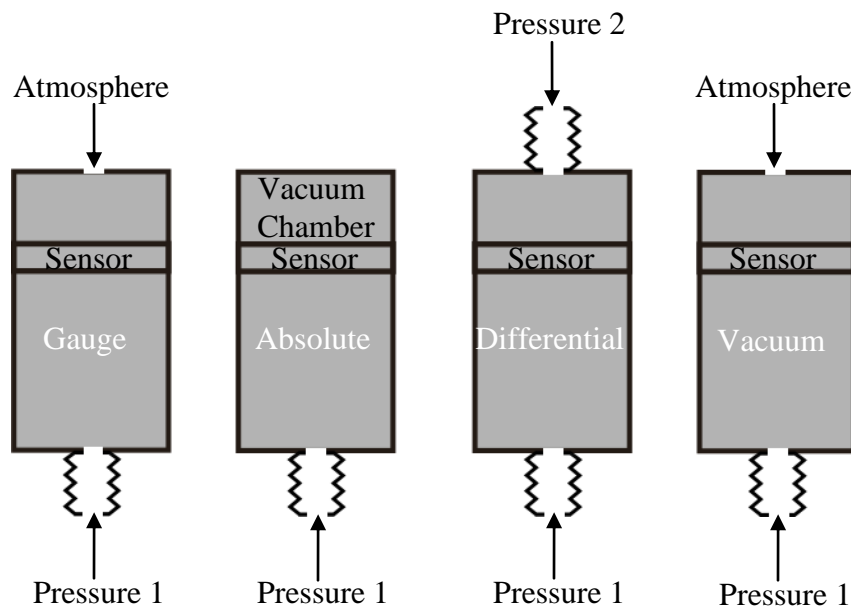
Figure 2.10: Pressure Sensors Types.

Gauge pressure sensors are the most common. If using psi units, absolute and differential pressures will have units of psia and psid, respectively. Gauge pressure sensors will have units of psig.

During the execution of an engine test, a number of issues could expose the dynamometer or engine to excessive vibration. Some of these could be a damaged or broken driveshaft, engine failure, unbalanced adapter plates or bearing wear on the dynamometer. When a new engine installation is complete, there may also be resonances in the mechanical system. Extended periods with a high level of vibration can result in costly damage to the dynamometer. An inexpensive vibration monitoring sensor is used to sense the vibration level. The sensor is an Ifm Efector VKV021 [8]. It outputs an analog signal proportional to the RMS vibration velocity, in accordance with the ISO 10816 standard [8]. A reference for acceptable vibration levels, based on ISO 10816, is shown in Table 2.2. The vibration sensor's output signal is monitored by the engine control system. The bearings of the dynamometer are also instrumented with thermocouples to monitor for bearing damage. Figure 2.11 shows the vibration sensor mounted to the dynamometer bearing housing.
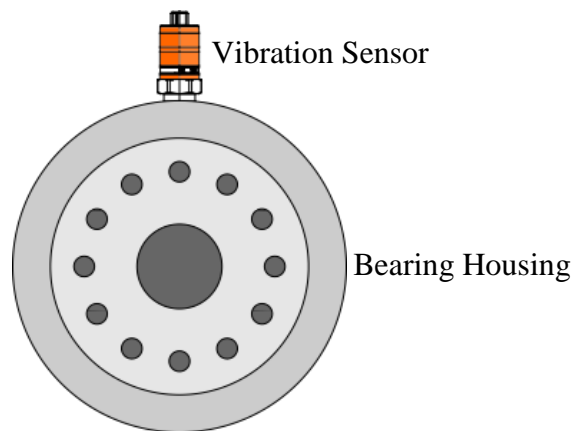
Figure 2.11: Dynamometer Vibration Sensor.

Table 2.2: ISO10816 Vibration Level Reference, Reproduced from Ifm Efector [8].

Vibration Level Reference Guide (based on ISO 10816)					
Machine Type		Class 1	Class 2	Class 3	Class 4
	mm/s	Small machines (20hp)	Medium machines (100hp)	Large machines rigid foundation	Large machines soft foundation
Vibration Velocity (RMS)	0.28	Good			
	0.45				
	0.71				
	1.12				
	1.80				
	2.80	Satisfactory			
	4.50				
	7.10	Warning			
	11.20				
	18.00	Unacceptable			
	25.00				

2.2 PC Based Distributed Control Strategy

In order to provide control of the engine and to gather data from the test cell, a computer system needs to be developed and utilized. This control system will make use of industrial grade personal computers. Using personal computers is the most common approach taken when implementing an engine test cell control system. Personal computers are inexpensive and can be easily upgraded when higher performance central processing units (CPUs) become available.

Two types of architectures were considered for this control system: the centralized control system and the distributed control system. In a centralized control system, all of the inputs and outputs are routed into one main control cabinet. Today, this type of system would typically incorporate a PXI chassis as the main measurement

device, with a short MXI cable connecting it to the main control computer. This architecture has been used in a recent upgrade of the ADACS engine test cells at PERDC. The PXI architecture provides the ability to perform large bandwidth, low latency data acquisition. In some applications, this benefit is more desirable than having the flexibility to move the data acquisition devices closer to the sensors.

Some centralized measurement systems require long cable runs back to the main control cabinet. The primary disadvantage of long cables is not the cost, but signal degradation. Signals can degrade from the sensor to the measurement device because of losses in the cable or noise contamination. Engine test cells typically have cable lengths of at least 15m when using a centralized control system.

Conversely, a distributed control system aims to distribute measurement devices closer to the sensors. The United States Tinker Air Force Base recently upgraded their jet engine test cells using a distributed architecture [9]. A distributed control system was chosen for this thesis as well. Distribution of the control system adds a small amount of complexity to the overall project. However, the added complexity is outweighed by other improvements and the ease of future expansion. Table 2.1 provides an overview of the positive and negative points associated with distributed control systems.

Table 2.3: Distributed Control Arguments.

Positive	Negative
Low cabling costs	Possible Latency in data acquisition
Easy expansion	Communication issues
Embedded computing	More points of failure
Less chance of noise	Poor environmental conditions for DAQ hardware

Centralized and distributed I/O configurations are depicted in Figures 2.12 and 2.13, respectively. The real time computer is located in the control cabinet for both systems. Only the measurement equipment changes location.
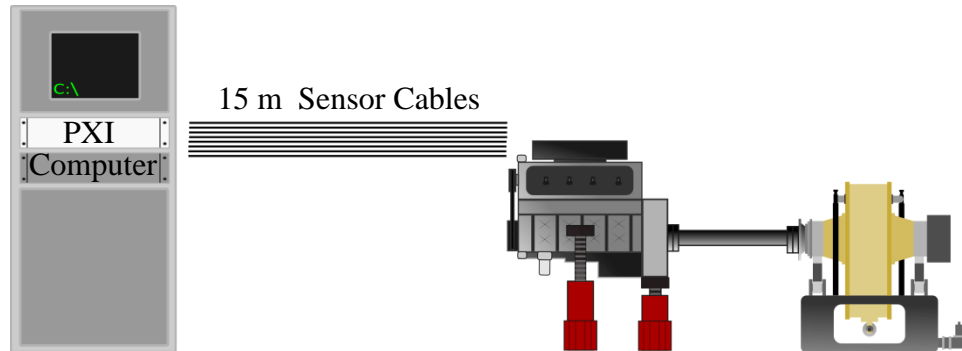
Figure 2.12: Centralized I/O.
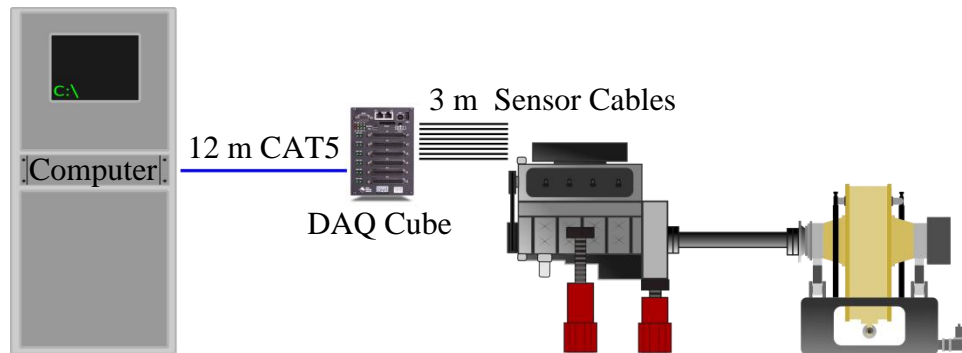
Figure 2.13: Distributed I/O.

In Figure 2.12, the centralized PXI chassis is located in the control cabinet along with the computer, at a distance from the sensors. The DAQ cube, in Figure 2.13, not only brings the I/O closer to the sensors, it also provides the ability to execute custom code. This could be used to execute PID control algorithms directly on the DAQ cube instead of the centralized computer.

2.3 Real Time Control

Controlling an engine test cell requires the use of a real time operating system (RTOS). The justification for using an RTOS concerns two issues. First, there is the need for deterministic control and data acquisition. Secondly, there is the importance of protecting the engine being tested. The expense of not only the engine, but the fuel and the time involved, warrants a level of guarantee from the operating system to perform timely in the event of a fault.

An RTOS provides the ability to execute code deterministically with very little latency [16]. Determinism is made possible by the use of priority. The RTOS provides a method to assign a priority to each section of code. These priorities are then used to determine which piece of code will execute at any point in time. Latency is the time from when an event is triggered, such as interrupts or timer expiration, until the code handling the event actually executes.

QNX Neutrino was chosen for the real time operating system for a number of reasons, as discussed below [10]:

1. A non-commercial license was free to test the operating system before purchasing a license.

2. It has a proven track record in the industry.

3. No kernel compiling is needed. The operating system is simply installed and provides real time performance.

4. Ethernet drivers run in real time, all other drivers do as well.

5. The learning curve was very quick due to excellent documentation. The development tools were very useful as well.

2.4 Graphical User Interface

There are both positives and negatives with using an RTOS for the control system. Most of the investment into an RTOS is spent perfecting and validating its deterministic performance. There is little effort invested in GUI development. Many RTOS implementations do not even support a graphic display, since they mostly run in deeply embedded applications.

The approach taken in this project was to execute the control and data acquisition on the RTOS as a terminal based application. The real time application provides a communication link to a remote computer running Microsoft Windows. The Windows computer is used to provide a user friendly graphical interface for interacting with the real time application. The communication between the two computers is performed over an Ethernet connection.

Separating the visualization from the control aspects is not uncommon. It is a widely used design pattern in industrial controls. For example, a human machine interface (HMI) is usually not included as part of the same device as a programmable logic controller (PLC). If the visualization computer fails or does not respond for an extended period of time the control computer will continue to operate independently.

With this architecture, the real time computer and the visualization computer could potentially be miles away from each other. This is not required for this control system, since the two computers are located at the same test cell. Typically, the two computers would be located within 4 meters of each other.

2.5 System Overview

The engine test cell includes many devices that are integrated into a complete control system. Not all of the devices that were integrated into the system will be discussed in this thesis. An overall block diagram of most of the devices is shown in Figure 2.14.

The two main components of the system are the QNX real time computer and the Windows visualization computer. The real time computer is responsible for collecting data using the data acquisition devices, as well as running all of the test sequences. Most of the data is acquired over Ethernet. Once an engine test is started, it does not require the Windows computer to operate.

However, the Windows computer provides the ability to monitor and update the status of the real time computer. It is normally turned on and communicating with the real time computer. The Windows computer contains all of the editing tools for configuring the test cell and creating the test sequences.

The ATI/ETAS calibration computer is used to communicate to the engine powertrain control module. This is independent of the engine control system developed. The ASAP3 protocol is used over Ethernet to integrate the engine test cell with the calibration software [11]. A CAN bus connection between the ATI/ETAS computer and the visualization computer allows test data to be transferred to the calibration software. This is configured using automatically generated, industry standard CAN DBC files [12].

The dynamometer controller is configured to communicate using both an RS-232 serial port and an analog interface. This provides for a fast analog response, as well as noise free digital data.

A device labelled emissions analyzers includes drivers written for different exhaust gas and combustion analysis equipment. The list of devices includes: AVL Smoke Meter, AVL Soot Meter, California Analytical Gas Analyzer, Horiba Mexa Gas Analyzer and CAS Cylinder Combustion Analyzer. Most of these drivers were written as separate utility applications that return the acquired data to the engine test control system over Ethernet.

The engine test control system also includes a web based remote monitoring tool. This provides the capability to view the status of the test cells from an office computer. This is discussed in more detail in Appendix I.
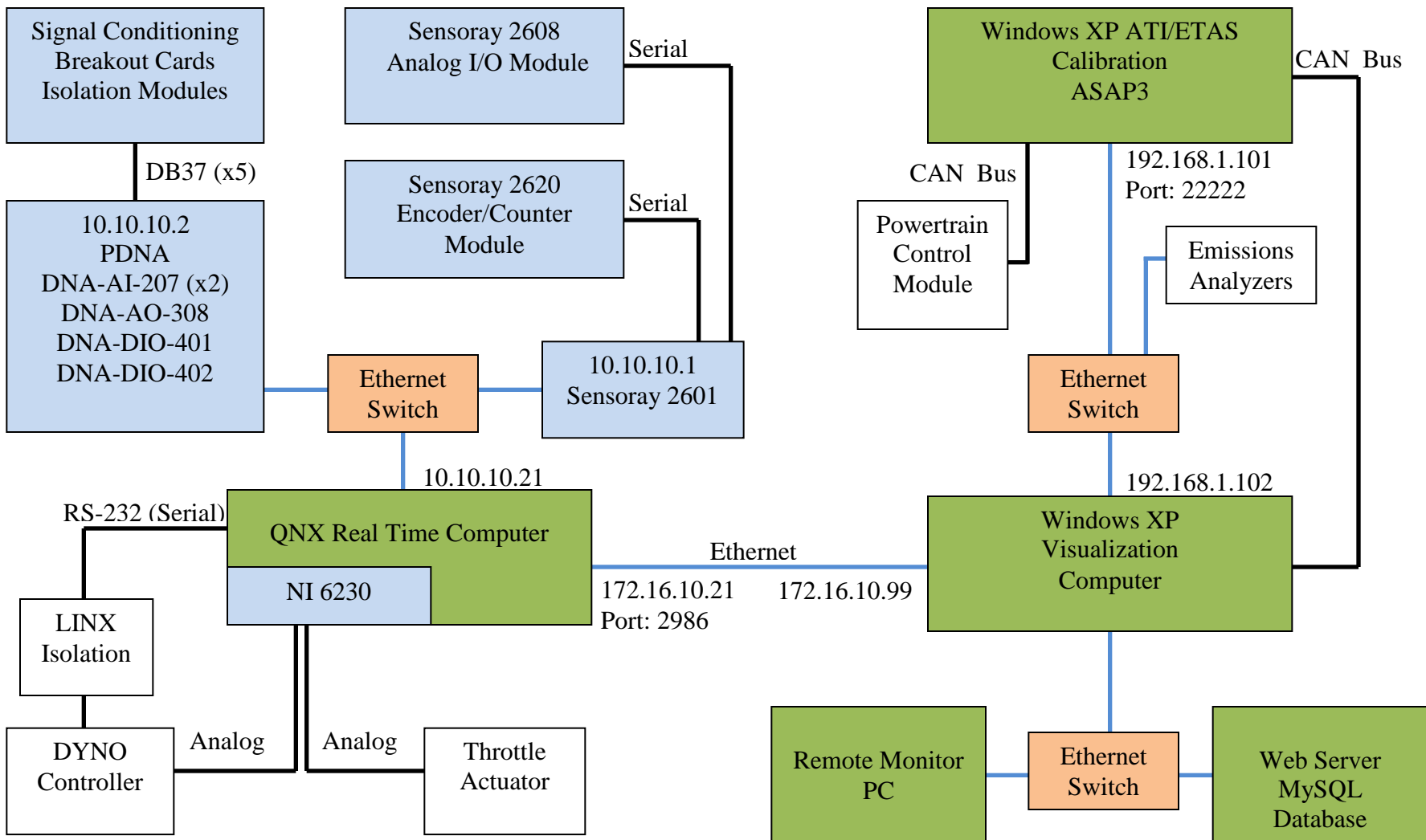
Figure 2.14: Control System Layout.

CHAPTER 3: IMPLEMENTATION

3.1 Ethernet DAQ Hardware

The main data acquisition device used in the design of the engine test cell control system was the PowerDNA cube, shown in Figure 3.1. The same device was used for the Tinker Air Force Base jet engine test cell upgrade [13]. This device was purchased from the United Electronics Industries Company. The PowerDNA (Distributed Networked Automation) cubes communicate over 1GBit/s Ethernet. Hardware scans are guaranteed in less than 1 millisecond, with up to 1000 inputs and outputs. This device has drivers for multiple operating systems, including QNX, Linux, RTX, and Windows. Each cube purchased can hold a total of six I/O cards. The cube supports a watchdog shutdown function. This function will reboot the cube and reset all its modules in the event of a communication interruption. This functionality offers a layer of security that shuts down the engine control system if communication is lost to the host computer. In addition, the PowerDNA cube has an SDK to support the development of applications to run directly on the device.

Figure 3.1: PowerDNA Ethernet DAQ, Image Courtesy of UEI Industries.

The cube was configured with I/O cards as listed in Table 3.1. The selected cards include a mix of analog and digital input and output cards. Should the need arise for more I/O in the future, another cube could be purchased and added to the system with little effort.

Table 3.1: PowerDNA Configuration.

Slot	Device	Specifications	Cost
1	DNA-AI-207	16- differential channel, 18-bit, 1 kS/s per channel, analog input, ±10V input range	$800
2	DNA-AI-207	16- differential channel, 18-bit, 1 kS/s per channel, analog input, ±10V input range	$800
3	DNA-AO-308	8-channel, 16-bit, 100 kS/s per channel, ±10V analog output Board	$800
4	DNA-DI-401	24-channel digital input board, 5-36V logic levels, OptoIsolated Input	$600
5	DNA-DO-402	24-channel digital output board, 80mA per channel output drive capacity	$600
6	Empty		

The PowerDNA cube supports a mode of operation called real time data map (RtDmap). The cube is configured, through the application software, to sample data at a specific rate. A hardware clock is used to sample the data at the specified rate. This data is then mirrored on the host computer when requested. This ensures hardware accurate sample times, even though the data is transferred over Ethernet. This is shown with a

block diagram in Figure 3.2 where the data values and transfer of Ethernet packets are shown [14].
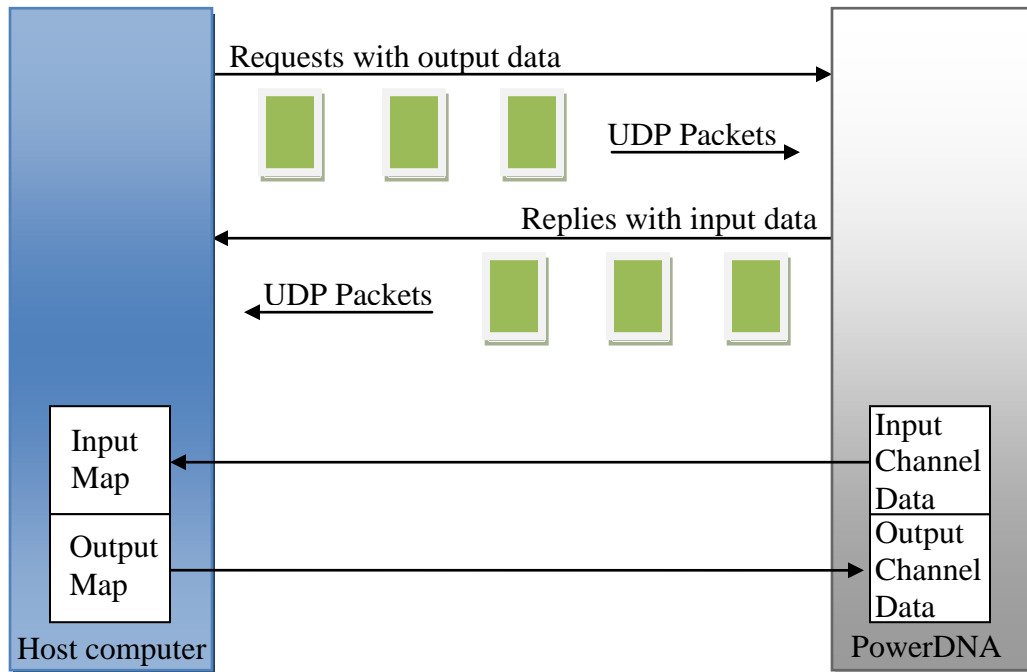
Figure 3.2: Real Time Data Mapping Operation.

A number of other data acquisition devices were used in the design of the hardware for this system. Each of these devices is listed in Appendix K. The sampled data, collected from the data acquisition devices, will be stored in memory and manipulated by the real time application. The sampled data returned is in the form of a voltage, current, frequency or binary state. These values are transformed into engineering units and stored in a new memory location. Each of these values will be referred to as a point throughout this thesis. The programming aspects of the various point operations can be found in Appendix A.

3.2 Real Time Application

The primary goal of the real time application is to control the operation of the engine, dynamometer and test cell, in a deterministic fashion. Deterministic means that the timing of the software execution is the same each and every cycle. When dealing with a large software project containing many different components, a good strategy is to divide the project into smaller sections. The code for each of the smaller projects is written and tested independently. The individual projects are then put together to form the final application. The real time application is referred to as the real time database, and has software components that perform data acquisition, communication, test execution and data logging.

3.2.1 Software Architecture

The real time database consists of a single application that executes in a console window of the RTOS. When the application is first started, it spawns a number of threads, each of which is shown pictorially in Figure 3.3. Each of the threads will be discussed briefly to provide an understanding of the internal operation of the real time application. Each of the threads is assigned a priority which is used by the operating system to decide which thread to run first, when more than one thread is ready to execute. Some of the threads execute cyclically based on a timer, others execute asynchronously waiting for an event to occur. The events could be a message from another thread or a communication message from an Ethernet or Serial port.
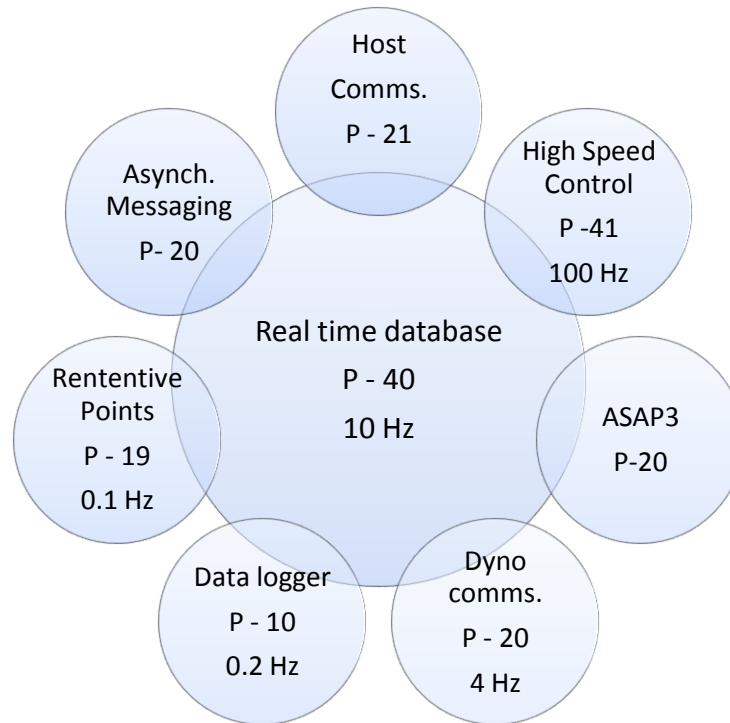
Figure 3.3: Real Time Application Overview.

From Figure 3.3, the letter P followed by a number is the priority of each thread. The highest number has the highest priority. If the thread is cyclic, a frequency value is included. Each of the threads performs a specific task. The configuration for each thread comes from a common SQLite database that is managed on the Windows computer.

All of the threads share access to a common data structure stored in memory. The data structure holds the values of points. Each point represents one value, such as engine coolant temperature (T_EngCoolant) or ignition power (IgnPower). This data structure uses a C++ Standard Library map (std::map) as a lookup table. The C++ Standard Library map is an associative container class [15]. The lookup table provides the ability to find the value of a point using its text name. The data structure itself, is sometimes directly referred to as the real time database in this document. The std::map contains the

current value of all of the real time points. The implementation of points is a very long topic which is discussed in full detail in Appendix A.

Since the std::map is shared with many threads, access to it is restricted to one thread at a time. This is accomplished through the use of a mutex. The mutex is also a global object that is shared among all of the threads. A mutex is a short form for mutual exclusion. It provides a software locking mechanism so only one thread can access an object or objects at a time. The operating system includes functions to lock and unlock the mutex in a safe manner [16].

A mutex is a software lock. It is the responsibility of the programmer to ensure it is being used properly. The operating system will make sure the mutex is unlocked before allowing access to the protected objects. It will also give the highest priority waiting thread access to the objects first.

3.2.2 Individual Thread Flow Charts

The following section will give a brief introduction to the function performed by each of the threads. A thread is a light weight process that shares a memory space with the process or application that spawned it.

The real time database is the process that gets launched from the operating system. It is responsible for creating all of the other threads during its initialization. Once the initialization is complete, it enters an endless loop performing the same operation at a rate of 10 Hz, as shown in Figure 3.4. All control functionality, performed by this thread, is defined by the end user in the control.as, user.as and script.as script files. These scripts are written in the AngelScript C++ language [17]. The control.as file contains code for the basic operation of the engine test cell. The user.as file is a generic

file containing code that could change frequently. The script.as file contains an automatic test sequence that is created dynamically by a code generator. The full details of all these operations are discussed in many different appendices, including Points (Appendix A), Alarms (Appendix B), Scripting (Appendix D), and Test Builder (Appendix E).
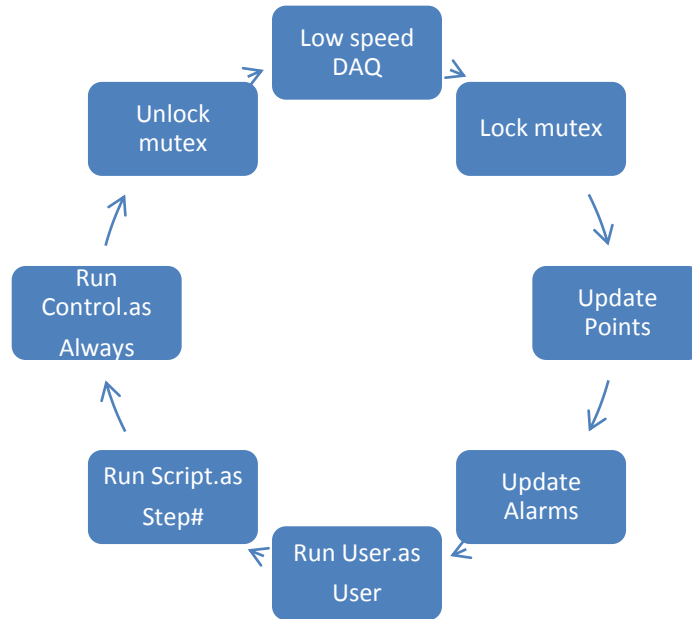
Figure 3.4: Main Process Flow Chart.

The high speed control thread, shown in Figure 3.5, is responsible for controlling devices that need to run at a higher cyclic rate, such as the dynamometer speed and throttle control loops. Each control operation that is required is defined within the HighSpeed function of the file control.as. The full details of Scripting are discussed in Appendix D.

28
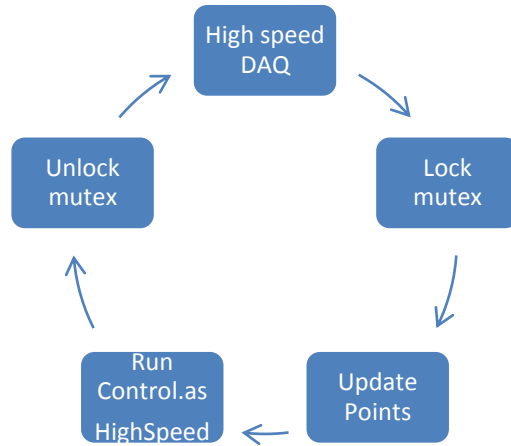
Figure 3.5: High Speed Thread Flow Chart.

The host communication thread sits idle, waiting for commands from the GUI application which is running on the Windows computer. This thread is shown in Figure 3.6. The commands come in the form of Ethernet UDP packets. Some of the operations could be a point value update or a database copy. The full set of commands is discussed in Appendix F. The mutex will be locked only if the command updates a point value.
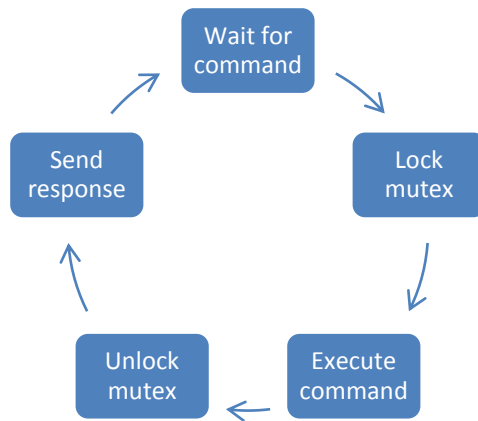
Figure 3.6: Host Communication Flow Chart.

The ASAP3 thread sits idle, waiting for commands from an application executing on a Windows computer which communicates with an engine calibration software

29

package. The commands come in the form of Ethernet UDP packets. The commands are discussed in Appendix F. The ASAP3 thread is shown in Figure 3.7. ASAP3 is a communication standard commonly used by engine calibration software packages [11].
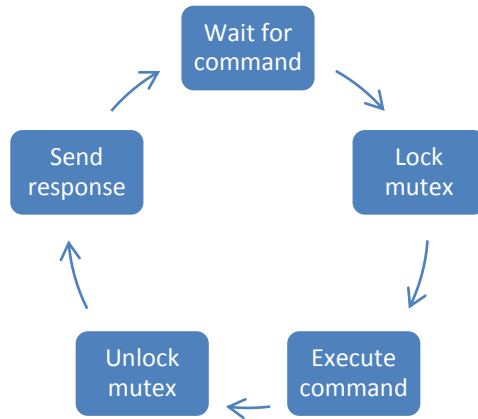
Figure 3.7: ASAP3 Flow Chart.

The dynamometer communication thread, shown in Figure 3.8, is a cyclic thread. It uses an RS-232 serial port to send and receive commands with a Dyne Systems Dyne Loc IV Eddy Current Dynamometer Controller [5].
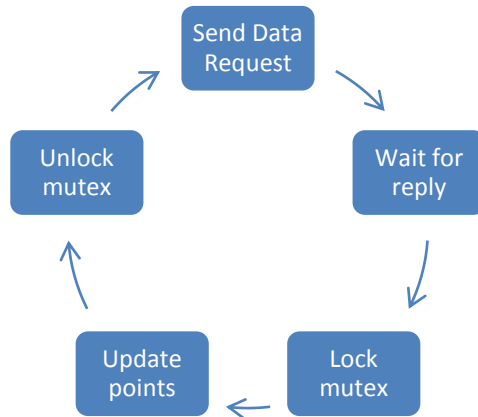
Figure 3.8: Dynamometer Flow Chart.

The data logger thread is a cyclic thread that executes once every 5 seconds. Figure 3.9 is a flow chart of the data logger thread. This thread performs write operations of engine test data to a SQLite database. This is a low priority thread, since disk access is typically a slow operation. It waits for approximately 50 write operations to be buffered in memory and then opens a transaction in SQLite to dump the data to disk. The details of the data logger are discussed in Appendix H. SQLite is a public domain, embeddable database [18].

Check for new data

Start SQL transaction

Perform SQL writes

End SQL Transaction
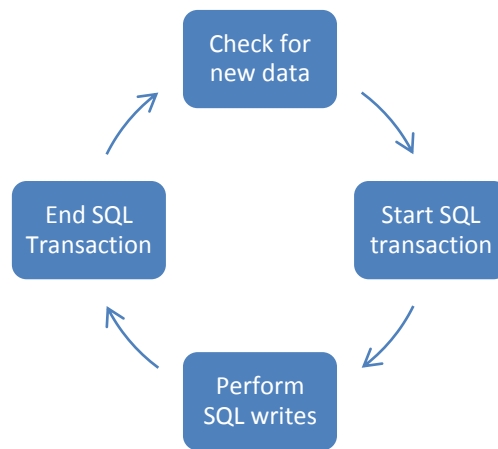
Figure 3.9: Data Logger Flow Chart.

The retentive points thread is a cyclic thread that executes once every 10 seconds. This is shown in Figure 3.10. It monitors for changes in points that are required to maintain state between an application start and stop. It then writes them to a SQLite database named monitor.db3. During the application start up, the retentive points are initialized to the last value that was stored in the monitor.db3 database. Details of this are located in Appendix A which discusses points.
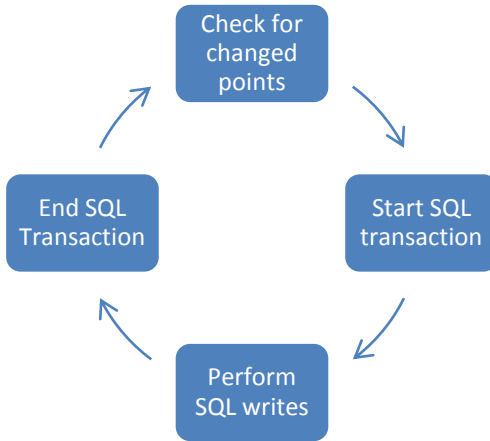
Figure 3.10: Retentive Points Flow Chart.

The asynchronous messaging thread, shown in Figure 3.11, sits idle waiting for messages from one of the other threads. It then relays the message data to the Windows application running on the visualization computer. This allows many threads to share a single Ethernet socket.
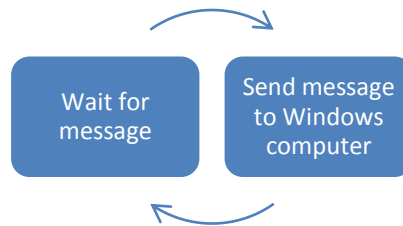
Figure 3.11: Asynchronous Messaging Flow Chart.

3.2.3 Summary

The real time application has been broken up into many small pieces of code. Each of the small pieces performs a very simple task that executes in a thread. All of the threads share access to a collection of global points which is protected by a mutex. Although the concept of threads and sharing memory appears to be confusing, it actually simplifies the code significantly. Changing the priority or cyclic rate of each thread can

be done independently of the other threads. If new functionality is required, a new thread can be added that performs the necessary operation. Extending the real time application in this format requires the application to be recompiled.

Recompiling the application is not done often. The application was designed to be completely dynamic. The end user defines the hardware topology, creates points to convert voltages to engineering units, and then develops an automatic test sequence. Alarms are also defined, to monitor the execution of the automatic test sequence. Each of these items can be modified dynamically while the application is executing. All of this is done in a graphical user interface (GUI) which will be discussed in the next section.

3.3 GUI Application

The entire operation of each component of the GUI application will not be discussed in the body of this thesis. It is important however, to understand the basic function that it provides. The appendices provide more details and explain how some of the code was implemented. Where appropriate, the reader will be directed to a specific appendix.

The GUI provides all the tools necessary to develop an automatic engine test sequence which can be executed on the real time controller. An automatic test sequence consists of a set of user defined steps. Each step consists of a set of instructions for the real time application to execute. In addition, the GUI provides visual controls that enable the creation, viewing and updating of points and alarms in the real time database.

The GUI design was based on existing applications used at PERDC. PERDC uses two engine calibration applications; Vision from ATI and INCA from ETAS. Both of

these applications have what is called a tabbed multiple document interface (MDI) main

window. An example of the INCA experiment window is shown in Figure 3.12.
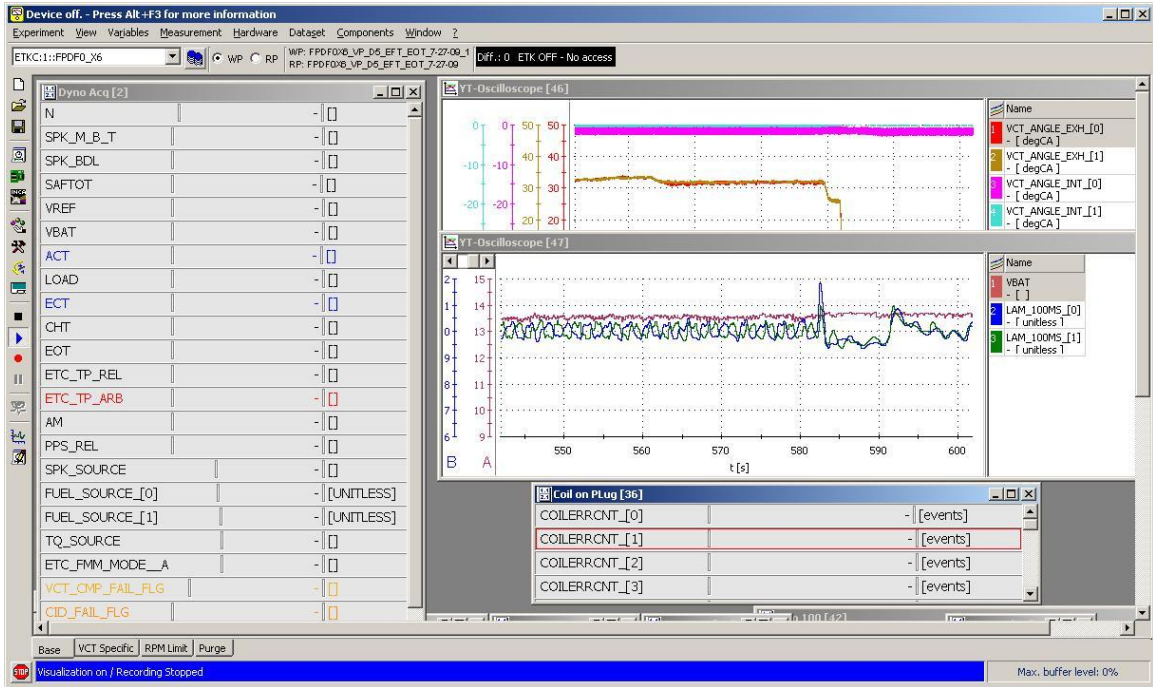
Figure 3.12: INCA Experiment User Interface.

This style of user interface is very familiar to the engineers and engine technicians

at PERDC, and was therefore used as the main window design for this application. This

can be seen below in Figure 3.13 below which shows the user interface of the engine

control system developed for this thesis. This user interface contains a large number of

visualization controls and tabs similar to the INCA screen. There are also a number of

supporting dialogs for creation of points, alarms and data logging.
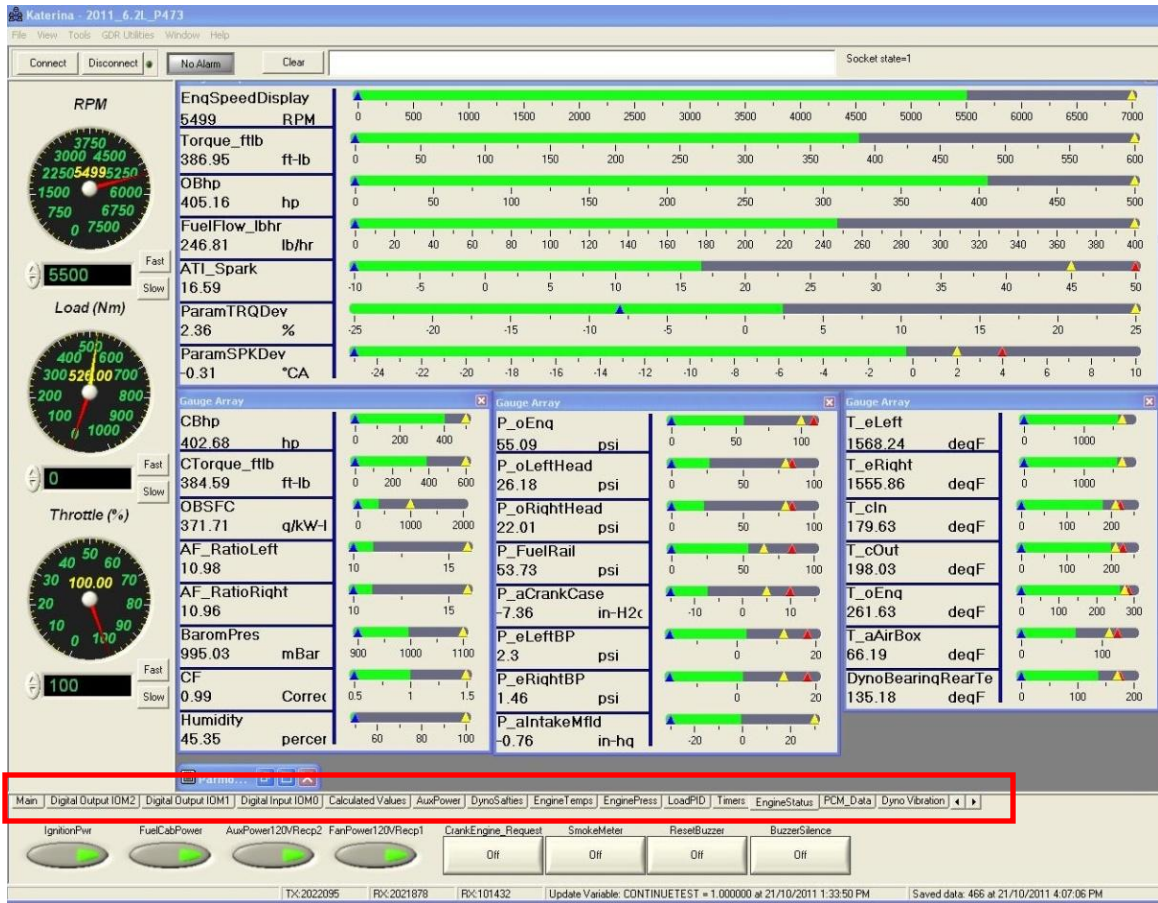
Figure 3.13: User Interface Main Window.

Each visualization control is assigned to one of the tabs. The tabs are identified with a red rectangle in Figure 3.13. Clicking a tab will display all of its associated controls. Tabs can be added or deleted using the tab manager shown below in Figure 3.14. This allows controls with related information to be grouped and displayed together.
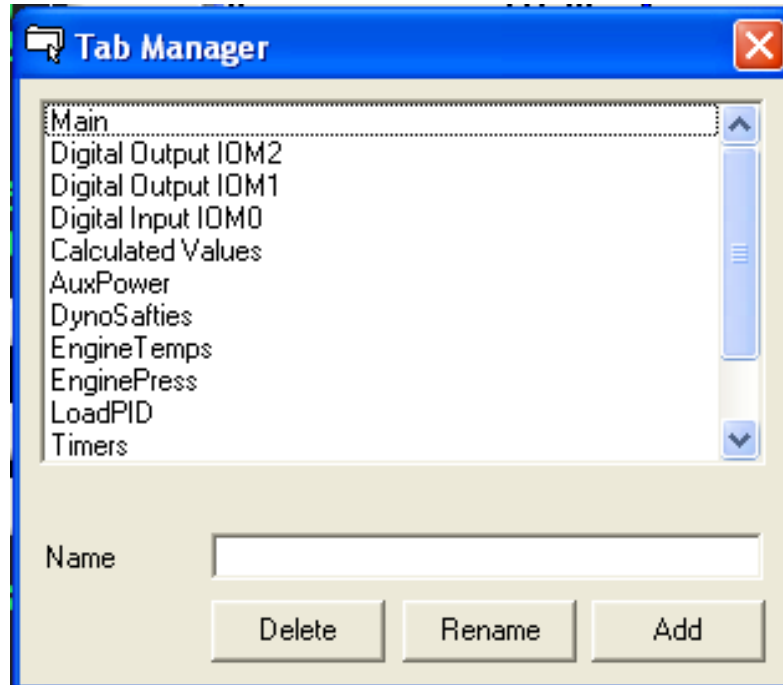
Figure 3.14: Tab Manager.

The positions of the controls, their properties, as well as the tab names are grouped together into a screen layout. A number of customizable controls were developed that are available to create the screen layouts. These controls are discussed further in Appendix J. Screen layouts provide the ability to view and modify data in the real time database. The screen design process is completely dynamic and done by the user while the application is running. There is no code written by the user to create the screen layouts. New controls are added to the screen using the view menu shown in Figure 3.15. Controls are automatically assigned to the currently selected tab. Positioning of the controls is performed using the mouse. This allows complete flexibility to quickly build a user interface for monitoring and controlling of the engine test.

Figure 3.15: Control Creation Menu.

The screen layouts are serialized to and from a SQLite database. The Layout Manager dialog, used to perform the storage and recall of layout screens, is shown in Figure 3.16.
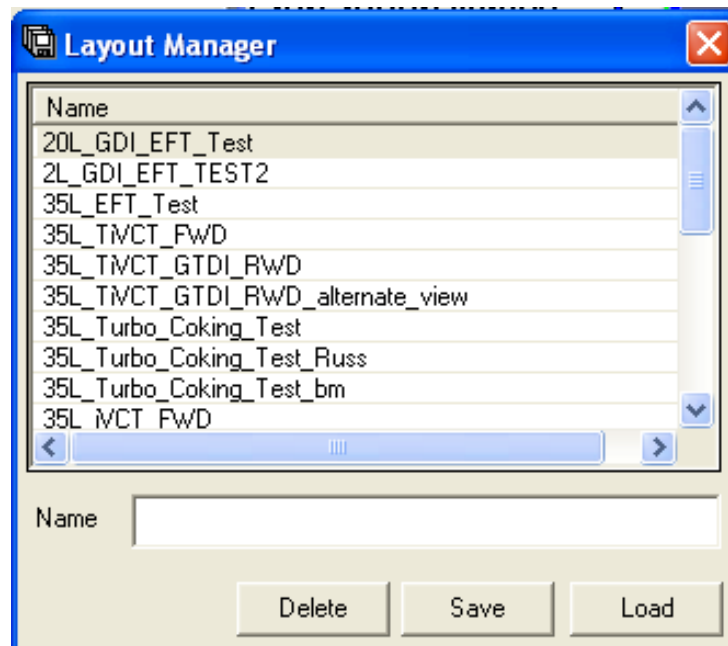
Figure 3.16: Layout Manager.

3.3.1 Real Time Database Link

The information to be displayed or updated by the controls, resides in the real time database. The real time database however, is executing on the real time computer. An ASCII communication protocol was designed to allow the two computers to communicate information back and forth. This communication protocol is fully discussed in Appendix F.

The protocol designed is based on simple commands. A command is sent from the Windows application to the real time controller, where it is then executed. If a response is required, it will be returned. For example, the Windows application can get the current value of all of the points in the real time database by sending the command "1|ListVars" to the real time controller. Internally, the Windows application has its own collection of points which it uses to store the same information locally. When the complete set of points is received, the application notifies the individual controls on the screen with an update message. Each of the controls then refreshes itself with the current point value. There is some optimization that is performed to ensure the screen updates are not overloading the CPU.

Controls that accept user input, such as tables and buttons, send update commands to the real time database. This is done asynchronously and only when an update is performed by the user. Since these controls are updating points, they use the update point command. For example, if a momentary button linked to the point "CrankEngine" is clicked, a message "3| CrankEngine |1" is sent to the real time controller. When the button is released, the message "3| CrankEngine |0" is then sent to the real time

controller. The exact values transmitted are configured by the user, without requiring manual entry of code. They do not need to be 1 and 0 as in the example above.

3.3.2 Point Editor Introduction

Data that has been acquired using a data acquisition board is normally in the form of a voltage, current, count, frequency, etc. While these data points are valuable, it would be much more intuitive to interpret the data if it was transformed into a pressure, temperature, speed, torque, etc. In order to accomplish this transformation, a number of point types have been developed. The point types range from a simple linear transformation, to an interpolated point, to a generic mathematical equation parser from the open source project muParser [19].

Points are defined using the point editor dialog shown in Figure 3.17. There are currently four point types; Linear, Interpolation, Equation, and Scratchpad. These point types are discussed in Appendix A. Point definitions can be updated while the real time application is running. This is useful when updating calibration tables of interpolation points or when adding a new point. It is not recommended to do this while the engine is running, but it is possible and has been done. The point editor provides a very intuitive and natural way to define points. There is no cryptic syntax that needs to be learned. All of the points are stored in one location. The field data for each of the points is stored in a table within a SQLite database.
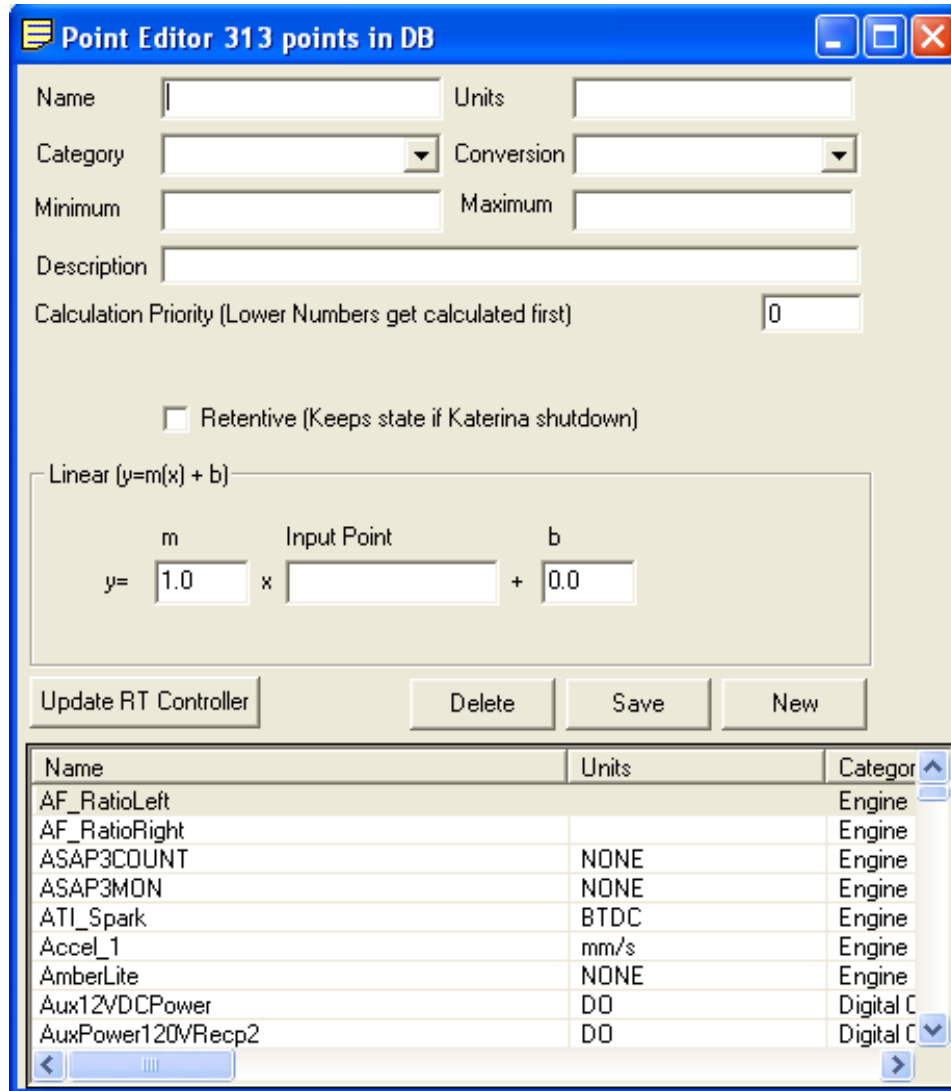
Figure 3.17: Point Editor.

3.3.3 Alarm Editor Introduction

An important part of the engine control system is the ability to detect fault conditions. An alarm editor was created for the purpose of defining these fault conditions. The alarm editor is shown in Figure 3.18. Alarms can be updated dynamically while the real time application is running. Appendix B gives full details of alarms and how to define complex alarm conditions. The engine control system is also

capable of advanced alarming using parameter monitoring which is discussed in Appendix C.
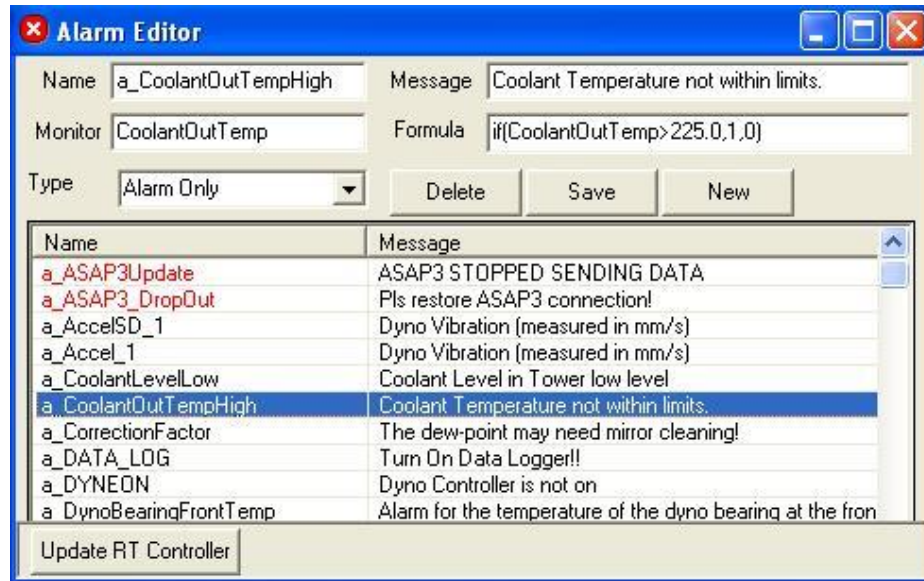
Figure 3.18: Alarm Editor.

3.3.4 Sequence Editor and Test Creation Introduction

In order to create an automatic engine test, test sequences are created using the test sequence editor, shown in Figure 3.19. Each test sequence contains a number of steps that are executed by the real time database. The steps of the test sequence contain the setpoints for the engine speed and load, as well as the time length of the step. Complex instructions can also be performed at each step using AngelScript C++ code. The information entered in the test sequence editor is used by a code generator to create a valid test. The full details of how this is accomplished are contained in Appendix D and Appendix E which discuss Scripting and Automatic Code Generation and the Test Builder. Similar applications of scripting and code generation have been used in other real time environments [10, 20 - 23].

41
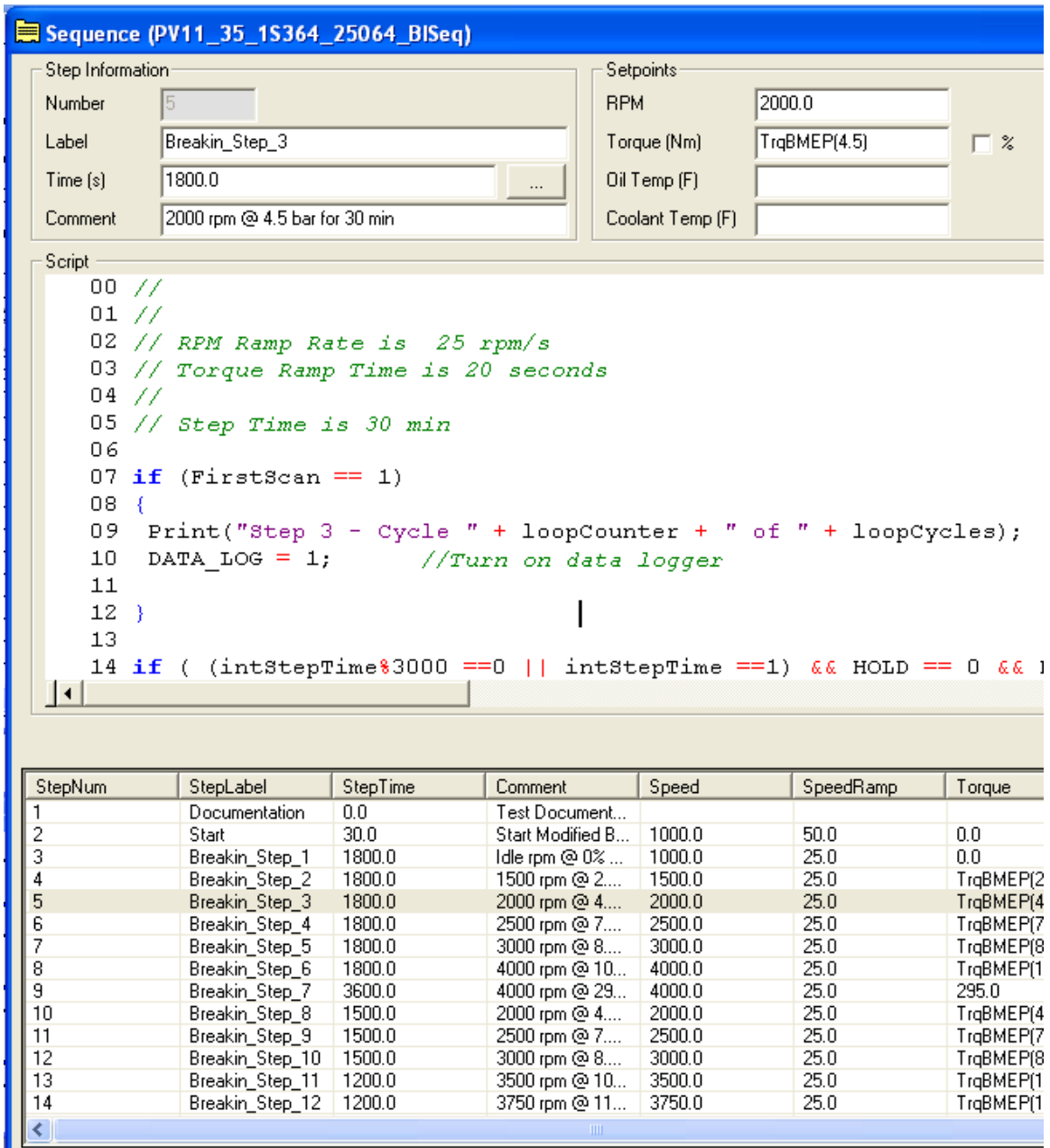
Figure 3.19: Test Sequence Editor.

3.3.5 Test Manager

To execute a test on the real time controller, it must first be loaded using the Test Manager shown in Figure 3.20. The Test Manager will dynamically generate the code required to execute the test sequence. The test information entered into the sequence

editor is used during this process. This code is then transferred to the real time controller where it is executed. During the test execution, the status is continuously updated and displayed on the form. The ability to skip steps or start and stop the test is provided through the push buttons on the form.
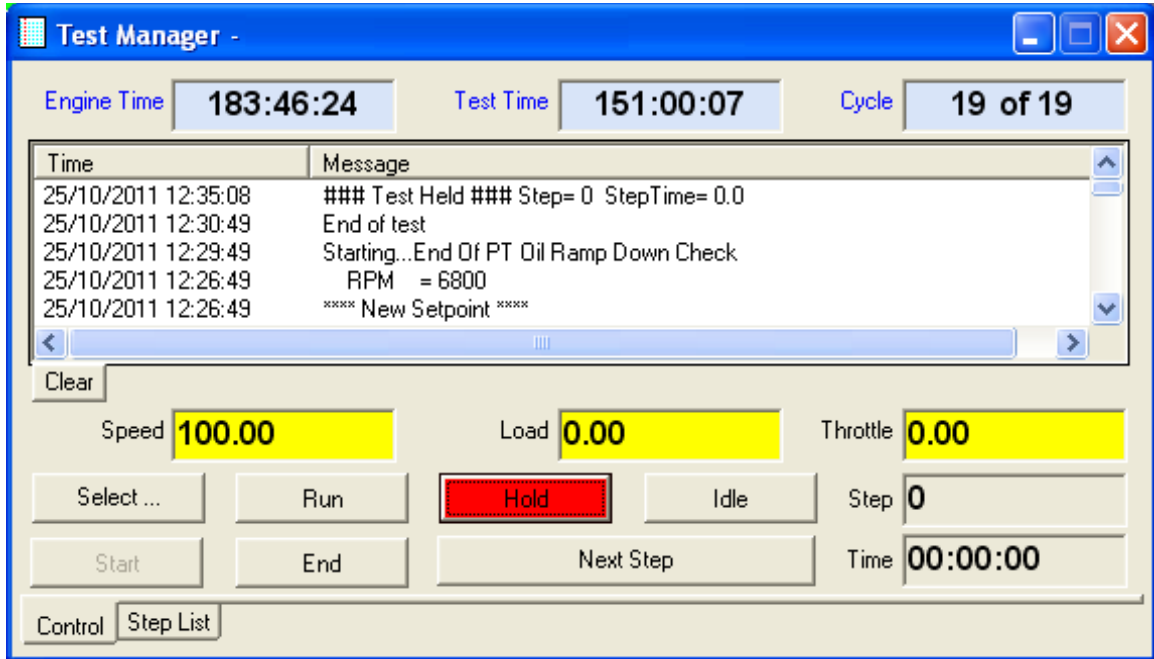
Figure 3.20: Test Manager.

3.4 Process Control

The test sequence editor provides the ability to assign setpoints to control loops at any step. The actual process control is done by separate threads, independently executing on the real time controller. Both manual and automatic modes of operation are available for the engine test cell. In manual mode, the setpoints are entered directly from the GUI application. There is no fixed number of devices that could be controlled by the system.

3.4.1 PID Controller

A commonly used process control algorithm is the PID controller. There are many different forms of this control algorithm. The one discussed here is a parallel topology, positional form PID controller. The parallel topology is shown in Figure 3.21. Positional form refers to the fact that the output from the PID controller is the actuator's position command. The velocity form, on the other hand, calculates an incremental change to the actuator's position. This is obtained by taking the derivative of both sides of the positional form.
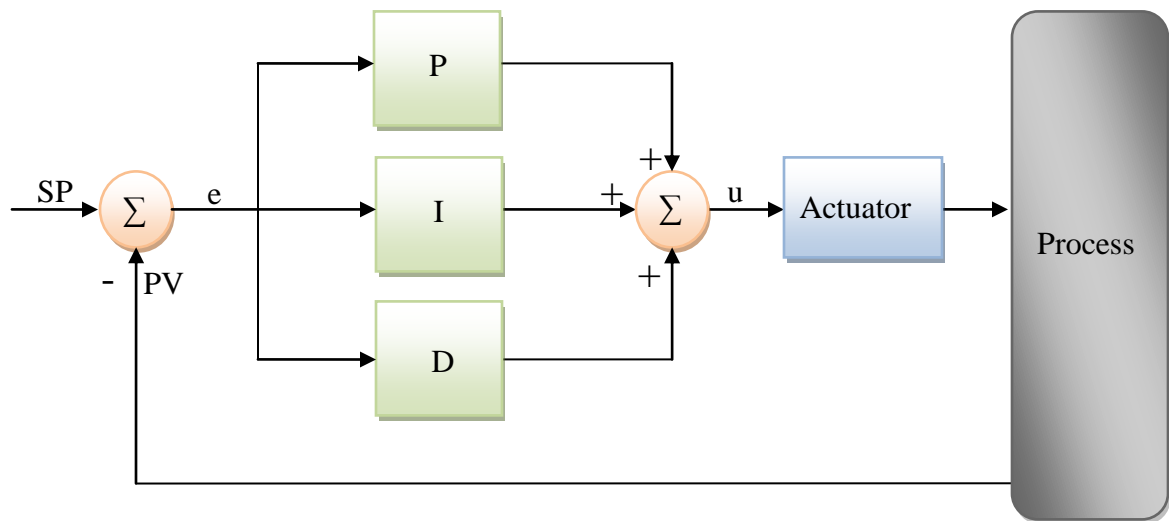
Figure 3.21: Parallel PID Topology.

The continuous time PID equation is shown in equation (3.1),

$$u(t) = K_P e(t) + K_I \int e(t) dt + K_D \frac{de(t)}{dt} \qquad (3.1)$$

where e is an error term defined as:

$$e(t) = SP(t) - PV(t) \qquad (3.2)$$

The terms K_P, K_I, and K_D are the gains for each component of the PID controller. To implement a discrete time version, a first order approximation can be defined using

Euler's method of numerical integration and the backward difference method of differentiation [24]. Using these methods, the discrete PID algorithm shown in equation (3.3) is obtained:

$$u[k] = K_P e[k] + K_I T_s \sum_{n=0}^{k} e[n] + K_D \frac{e[k] - e[k-1]}{T_s} \qquad (3.3)$$

The variable T_s is the data sampling period. In practical applications, a term named Bias is also added to the PID equation, as shown in equation 3.4 [24].

$$u[k] = K_P e[k] + K_I T_s \sum_{n=0}^{k} e[n] + K_D \frac{e[k] - e[k-1]}{T_s} + Bias \qquad (3.4)$$

The Bias term can be thought of as the initial condition for the numerical integration. Most control loops provide both a manual and an automatic mode of operation. When switching between the two modes, the output value should not have an instantaneous change in value. This is called bumpless transfer [25].

The term Bias is used to implement a bumpless transfer from manual to automatic mode. During the transition from manual to automatic, the current manual output value is stored in the Bias term. The summation for the numerical integrator is reset to zero and the current setpoint for the PID controller is set to the process value. The transition from automatic to manual simply holds the current output value until a manual change request is made.

When implementing the code for the discrete PID algorithm, care must be taken to account for a change in the gain term K_I. When this value is changed, the previous integration summation must be scaled using a ratio of the new and the old K_I values. If this is not done, a significantly larger or smaller value for the integral term would be obtained.

Wind up occurs when the output from the PID controller has a value that is physically impossible for the actuator to obtain. If the actuator is unable to force the process feedback value to the setpoint, the summation term will increase or decrease indefinitely. Since all actuators contain a physical limit, the summation term should be limited in value. In this thesis, wind up is limited by discontinuing the summation when the actuator output value is saturated at its upper or lower limit. The upper and lower limits of the PID output are normalized to a percentage (0% to 100% or -100% to 100%) and then scaled back to engineering units. It is natural to think of an actuator's position as a value between 0 and 100 percent open.

The derivative term will create an impulse output when the setpoint is changed. Many practical controller implementations modify the derivative term to act on the process value and not the error term. This is shown in Figure 3.22, where a PI_D controller topology is shown [26].
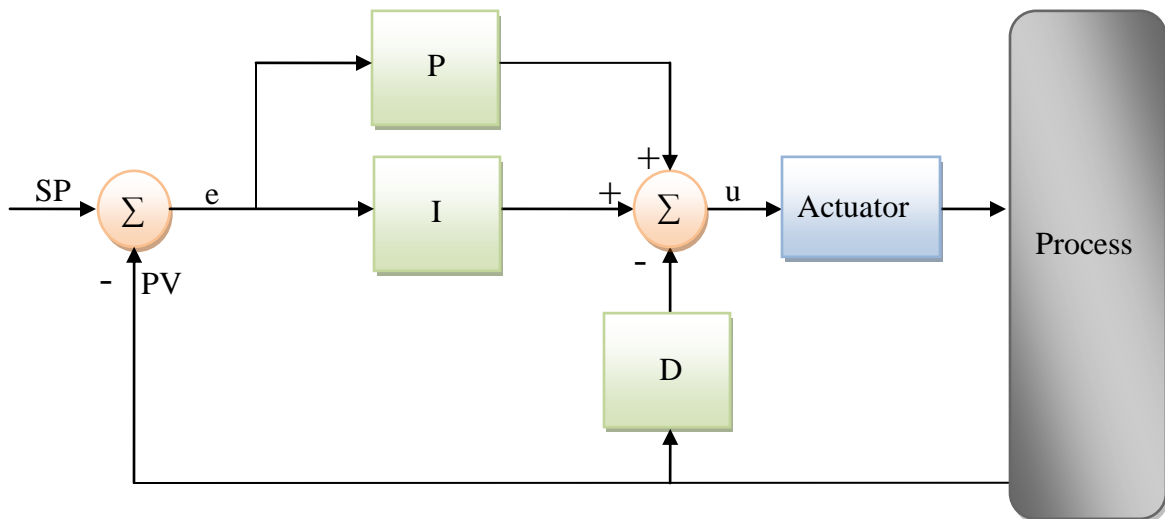
Figure 3.22: Parallel PI_D Topology.

The discrete implementation of the PI_D topology is given in Equation 3.5.

$$u[k] = K_P e[k] + K_I T_s \sum_{n=0}^{k} e[n] - K_D \frac{PV[k] - PV[k-1]}{T_s} + Bias \qquad (3.5)$$

The PID controller's function is to provide a setpoint for the actuator that will move the process in a direction that eliminates error. For a cooling loop, the actuator will often be moving in a positive direction, but will result in a decrease in the process temperature. This is called a reverse acting loop and the error term must be adjusted so that the actuator moves in the correct direction.

In some instances, a setpoint change should move the process to the new value as quickly as possible. In the case of engine speed and throttle control, it is required to ramp the setpoint from one value to another over a desired period of time. This is implemented by providing a ramp generator in front of the PID controller, as shown in Figure 3.23. The ramp generator has inputs for the setpoint, as well as the time required to reach the desired setpoint. The output from the ramp generator is incremented by an equal amount each sample period, until the setpoint is reached.
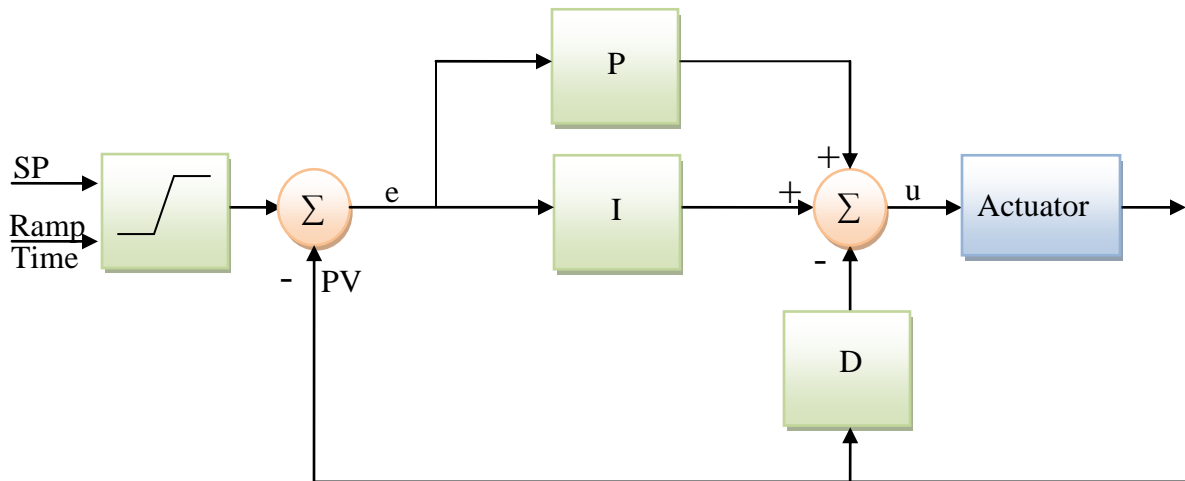
Figure 3.23: PI_D with Ramp Generator.

CHAPTER 4: RESULTS AND DISCUSSIONS

This chapter will first discuss the performance of the engine control system and the timing of the scripting environment used. Secondly, an example of the PID controller operation will be shown. Finally, results of test data gathered while running an engine using the control system will be discussed. Where applicable, the results will be compared to data from a commercially available system.

4.1 Real Time Performance

The performance of the RTOS can be affected by the hardware used or by errors in the software. In order to evaluate the deterministic performance of the engine test system, an eight hour recording of two cyclic timers was saved. One timer was configured to trigger an event every 100 milliseconds and the other every 10 milliseconds. The data samples were taken every 100 milliseconds. Over the eight hour period, no over runs were recorded. An over run is a condition where the computation time between time intervals is exceeded. This would result in a missed sample period. Table 4.1 shows the frequency of the time stamps for the 100 millisecond timer and Figure 4.2 is the for 10 millisecond timer.

Table 4.1: 100 ms Time Stamps.

Bin	Frequency
0.100032	195037
0.099932	92989

Table 4.2: 10 ms Time Stamps.

Bin	Frequency
0.009973	210882
0.010073	77102
0.009874	40
0.009774	2

From the data in Tables 4.1 and 4.2, it can be seen that there is a worst case latency of 226 microseconds for the times that were recorded over the 8 hour period. There is some uncertainty in the 10 millisecond numbers since only every 10th sample was recorded.

There is a trend in the bin values shown in Tables 4.1 and 4.2. The bin values are almost exactly 100 microseconds apart. This is no coincidence; the operating system has an internal tick timer that was programmed to run at 100 microseconds for this application [27]. This tick timer usually runs at 1 to 10 milliseconds. Running at 100 microseconds adds some additional overhead to the operating system. It will run a task at this interval to determine the highest priority process ready to execute.

Other control system applications, similar to this one, would likely use an internal PCI card, such as a data acquisition board, for timing purposes. The PCI board would trigger a hardware timed interrupt, instead of using an operating system timer. One of the goals of the original project was to have a completely distributed architecture. It was decided to only use the operating system timers to control the sample time. It

should be noted that the PCI card approach would reduce the worst case latency to tens of microseconds.

One other thing to note is that the total time for the 100 milliseconds timer adds up to 28802.52 seconds. There were exactly 288026 samples recorded in the data file, each with an expected interval of 0.1 seconds, for a total of 28802.6 seconds. This gives a difference in time of 0.08 seconds which is less than one sample, over the eight hour period. This proves that the timer is precise over long periods of time.

4.1.1 Real Time Application Performance

One of the interesting aspects of this project was the use of the scripting language for the core of the control application. When the project started, it was questionable if the timing constraints could be achieved using the scripting language for the application. A test was performed to determine what kind of computation is possible with the application created. Since the threads are all sharing a common mutex, the thread that locks the mutex the longest is the limiting factor. In the current application design, this is the main thread. This thread will be used to determine the possible performance that can be obtained.

The computer on which the data was collected contains an Intel Pentium Dual Core E2200 with a CPU speed of 2.20GHz. The code execution time, shown in Figure 4.1, includes all of the calculations of points, alarms, data logging and script calls made in the main thread at each timer event. The scripts running during the test include all of the code that is currently used for the test cell operation.

Code Execution Timing
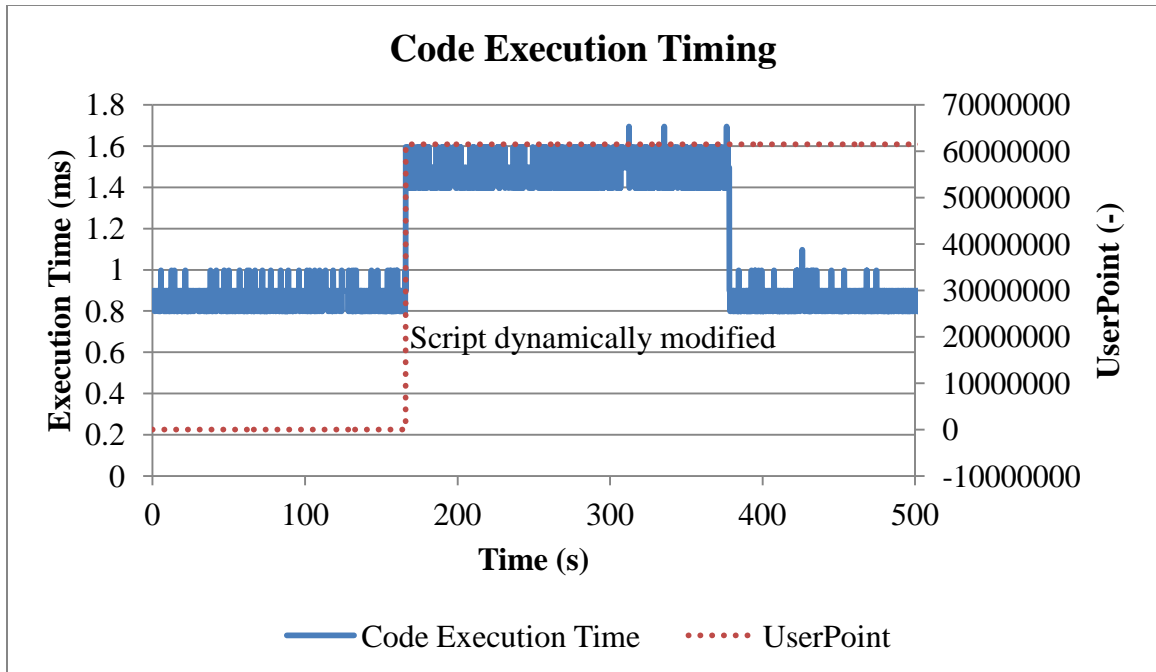
Figure 4.1: Code Execution Timing.

At approximately 160 seconds into the data log, a simple function with a loop executing 10000 times was dynamically added to the user.as script to stress the system. The code executed in the loop is shown below. The calculation performed is equivalent to a linear transformation. The code was dynamically removed from the script at about 380 seconds to show the system return to its original performance.

```
//Test script performance
void Stress()
{
        //Local variable
        double x=0.0;

        //Loop
        for(int i=0; i<10000; i++)
        {
                x += i * 1.23 + 4.56;
        }

        //Store result in real time database point
        UserPoint = x;

}
```

51

From the code above, the variable UserPoint in the function Stress is a point that exists in the real time database. Its value is correctly calculated by the script as 61539450 and is shown on the chart in Figure 4.1. The modified script added approximately 600 microseconds to the total code execution time. This is a small price to pay for the dynamic nature of the system. Not only are high execution rates possible, but these scripts can be changed on the fly while the application is running. The script is configured in a different thread, so no time is lost when loading a new script. Once the script is compiled successfully, a simple memory pointer is swapped. All of the points in the real time database are exposed to the script when it is compiled.

4.2 Dynamometer Torque Calibration

The load cell used to measure torque is connected directly to the dynamometer controller which has a built in signal conditioner [5]. The calibration is performed in two steps. Calibrating the load cell should be performed with the cooling water supply turned on and the engine drive shaft removed from the dynamometer [2]. First, the zero calibration is performed with both calibration levers and weight trays attached without any mass on the weight trays (refer to Figure 2.4). Second, the full scale torque is applied by loading the weight trays with the full set of calibrated masses. When this is complete, a validation of the calibration is performed using a number of different masses. PERDC has a specification for durability test cells that requires the torque measurement to be accurate within 1 Nm below 200 Nm and ±0.5% above this value. The calibration of dynamometer torque is discussed in detail in an SAE paper, along with some recommendations to follow when specifications cannot be met [28]. Figure 4.2 below shows the results of the dynamometer torque calibration.
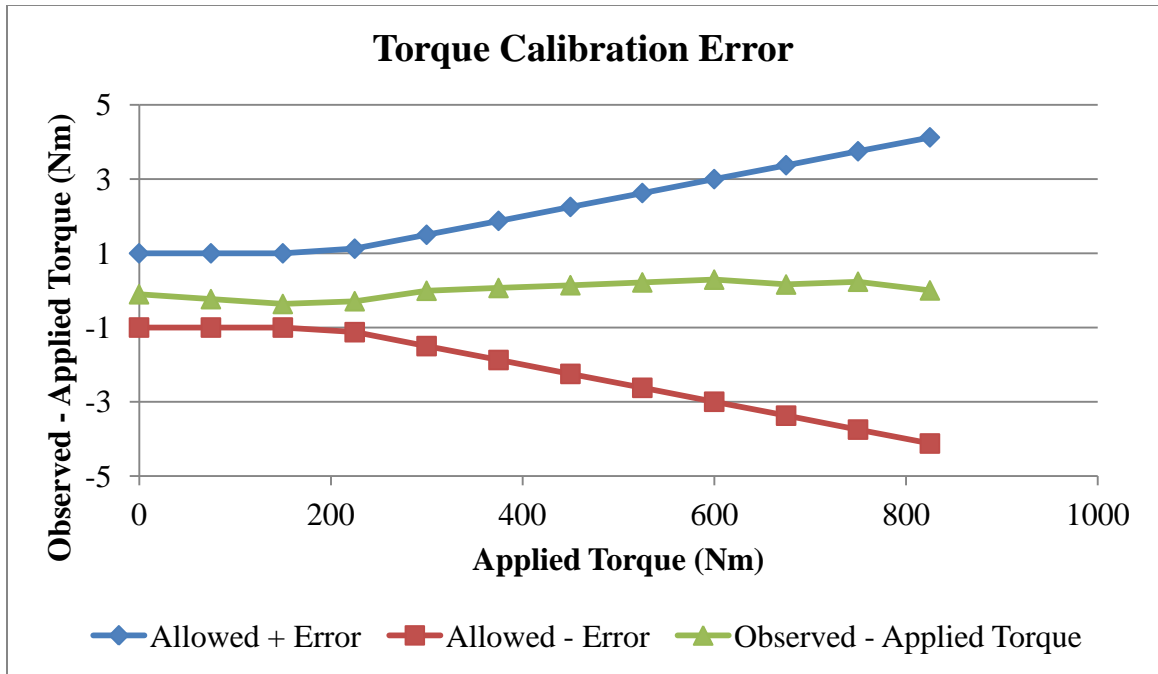
Figure 4.2: Dynamometer Torque Calibration.

4.3 Load PID Control

The control of engine load by throttle presents a number of challenges [1]. The complexity is due to the number of controllers and the non linearity of the torque versus the speed of the engine. The powertrain control module may also have a non linear output versus pedal position. This is compounded by the fact that the system must be able to control many engine types, each of which has their own characteristics. A few researchers have tried to apply multivariable controllers with limited success [29] [30]. A self tuning procedure for dynamometer torque control was evaluated by another researcher [31].

Since the eddy current dynamometer does not have motoring capability, the adjustment of throttle position will have an impact on speed, particularly at light loads. The application of gain scheduling may be appropriate in some circumstances to reduce

light load oscillations. Many of these dynamics can be shown with a few illustrations. Figure 4.3 shows five sections where speed and load are being adjusted.

1. The engine is idling with zero throttle and produces zero torque output.

2. Speed and load are ramped up simultaneously from idle. This presents a difficult situation, since the load controller is running open loop. The load controller starts to ramp the throttle, but does not see any change in the torque value that it is monitoring. This is because the dynamometer controller has not started braking the engine. It will not start to brake until the engine is at the desired speed. When the engine does reach its desired speed, the dynamometer controller will quickly start loading the engine. At this point, the load controller sees a very large torque increase and backs off the throttle. This causes the speed to drop. This can lead to uncontrollable oscillation if both the load and dynamometer speed controllers are tuned tightly.

3. The engine is stabilized and is holding load and speed setpoints.

4. The speed is ramped while the load is held constant. The dynamometer controller will decrease the braking force to allow the speed to increase. This results in a drop in torque. The load controller will try to reject this disturbance by increasing the throttle. When the engine reaches the speed setpoint another jump in torque is seen, since the throttle has over shot its value in an attempt to maintain torque. A similar drop and increase in torque would be seen even if the throttle were held constant in manual mode.

5. The engine speed is held constant while the load is ramped. This is much easier to control since the dynamometer speed controller and load controller

are effectively working together. The load controller increases throttle to produce torque, while the dynamometer controller increases the braking force to maintain a constant speed. This braking force creates the torque the load controller is looking for.
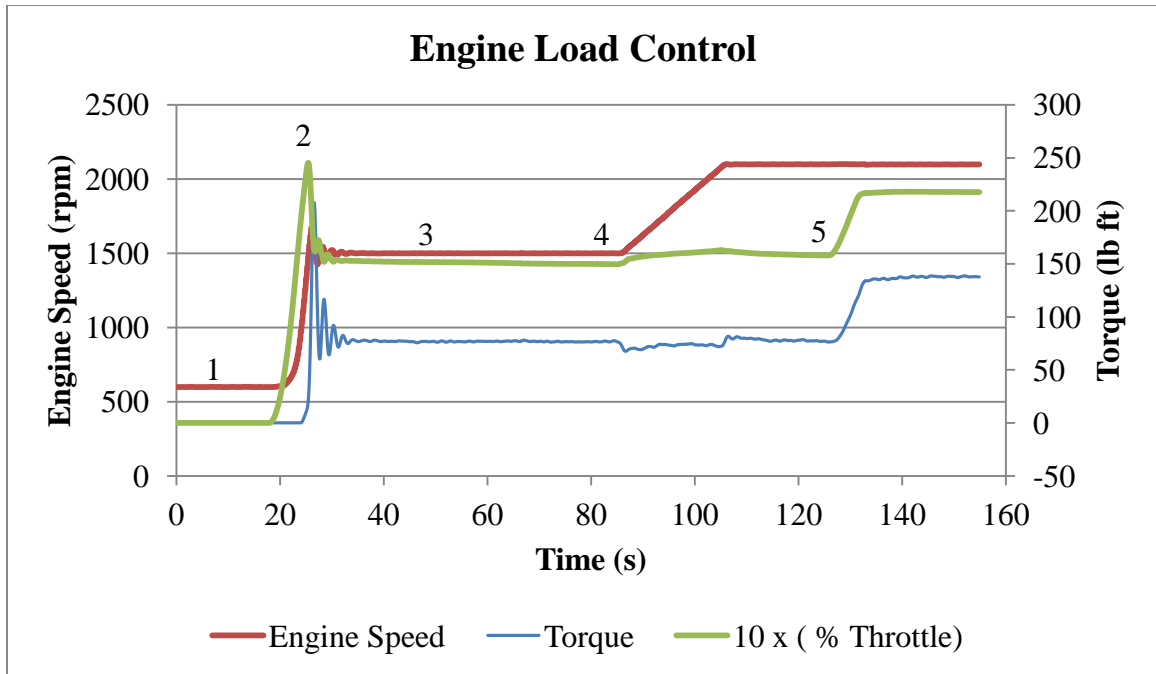
Figure 4.3: Engine Load Control.

A low speed and low load condition is shown in Figure 4.4. The engine speed and load are simultaneously ramped from 680 rpm and 50 lb ft. At the beginning of the ramp, the dynamometer and load controller will fight each other which cause the oscillations shown. This effect can be lessened with gain scheduling at low loads. Depending on the operating conditions of the test, it may not be required to make these modifications. Holding such a low load on an eddy current dynamometer is very

challenging. If the load controller is tightly tuned, there is the possibility of oscillation at low loads due to loss of speed when decreasing throttle position.
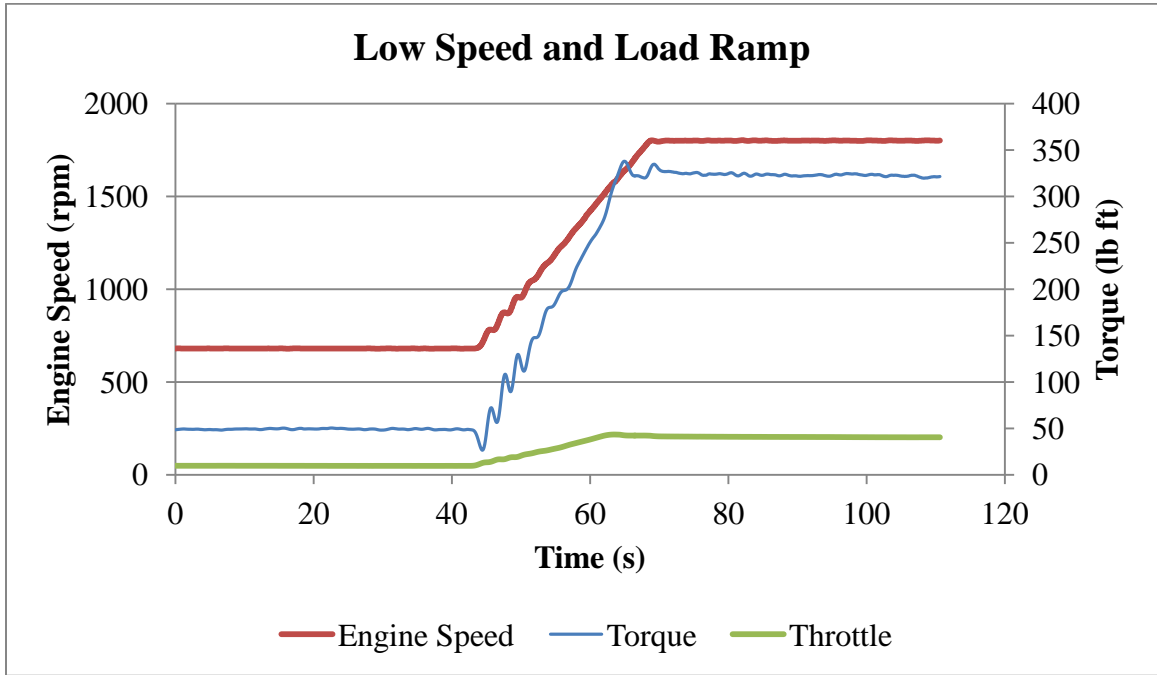
Figure 4.4: Engine Low Load Ramp.

Although still challenging, the region outside of the low speed and low load is easier to control. A medium speed and load ramp is shown in Figure 4.5. Here it can be seen that the speed and load curves ramp in a controlled manner.
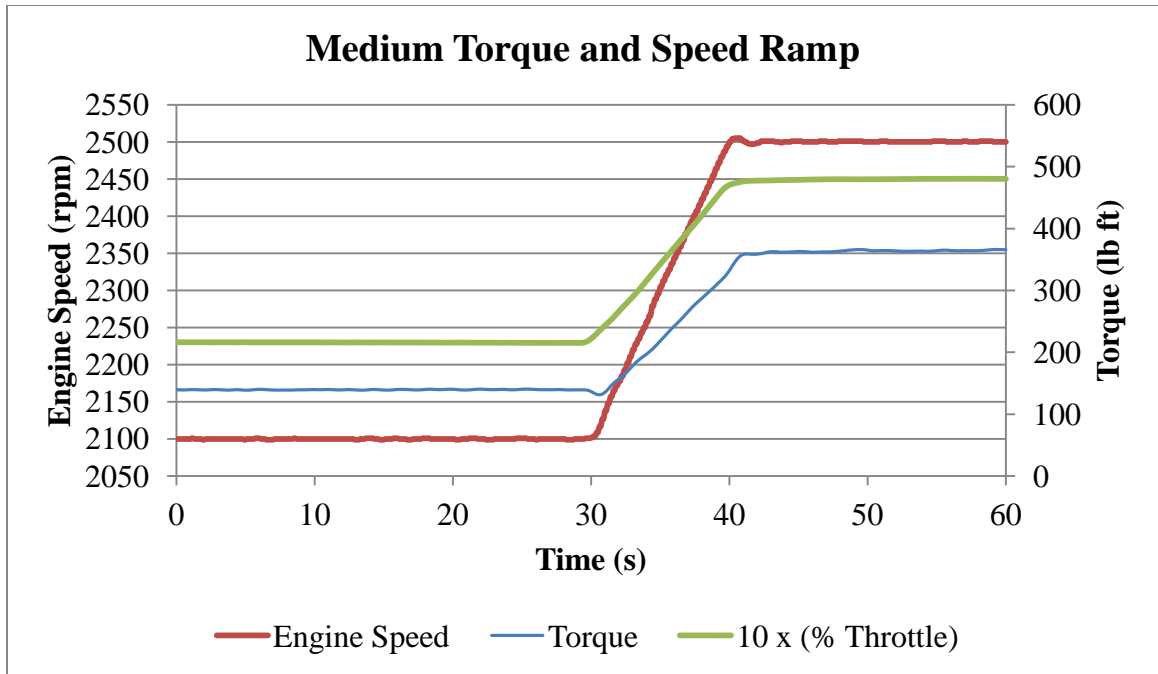
Figure 4.5: Engine Medium Load Ramp.

4.4 Test Results

In order to validate the data acquired from the engine test control system, a durability test was performed on an engine of undisclosed specification. The tests performed on this engine included:

1. Break-in Test

2. Pre Test Power Test

3. Engine Fatigue Durability Test

4. Post Test Power Test

The execution cycle for this engine was created on-site by another engineer using the tools developed for this thesis. The tests were performed by PERDC engine technicians. The results of the pre and post test power tests and the engine fatigue durability test will be discussed. The break-in test is simply a test to gradually break-in

the mechanical components of the engine. This test will not be discussed further. The engine successfully finished and passed each of the tests without failure.

4.4.1 Power Test Results

The power test is simply a sweep of engine speeds while holding the pedal position wide open to produce maximum power output. One of the criteria for a successful test is less than a 5 percent drop in output torque between the pre and post test power tests. The two power test curves are shown together in Figure 4.6. The percent difference in output torque is also plotted. The data values were generated by averaging the corrected torque values at each engine rpm test point. Average values are typically used in this plot, since there will be variation in instantaneous torque readings. Overall there was a drop in torque from the pre to post test power test curves, but it did not exceed the specified limit of 5 percent.
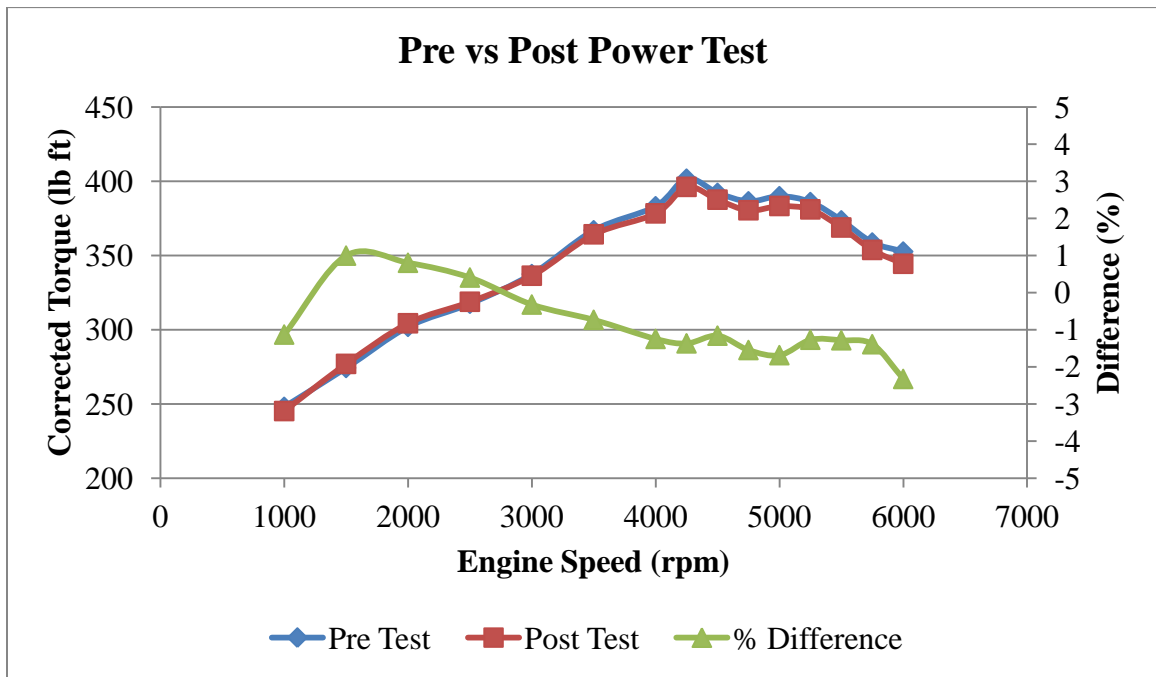
Figure 4.6: Pre vs. Post Test Power Test.

A different engine, from the same family and specification, was run on a commercially available test cell control system. The test cell used to perform this test contains an AC Dynamometer and a control system that was updated in 2010. All of the facilities such as fuel, fluids, electricity, and air handling come from the same source. The engines were manufactured and tested in the months of October and November, 2011. The engines tested are for different vehicle platforms and two different PCM calibrations were used during the tests. Each test was designed to have an equivalent number of engine cycles, but running at slightly different engine speeds. A different number of test hours were run to compensate for the speed difference. The total engine hours for each engine are shown in Table 4.3.

Table 4.3: Test Engine Hours.

	Commercial System	System Developed
EFT Test Hours	138	131
Total Engine Hours	168	152

The results of the pre and post test power tests for this engine, run using a commercially available control system, are shown in Figure 4.7.
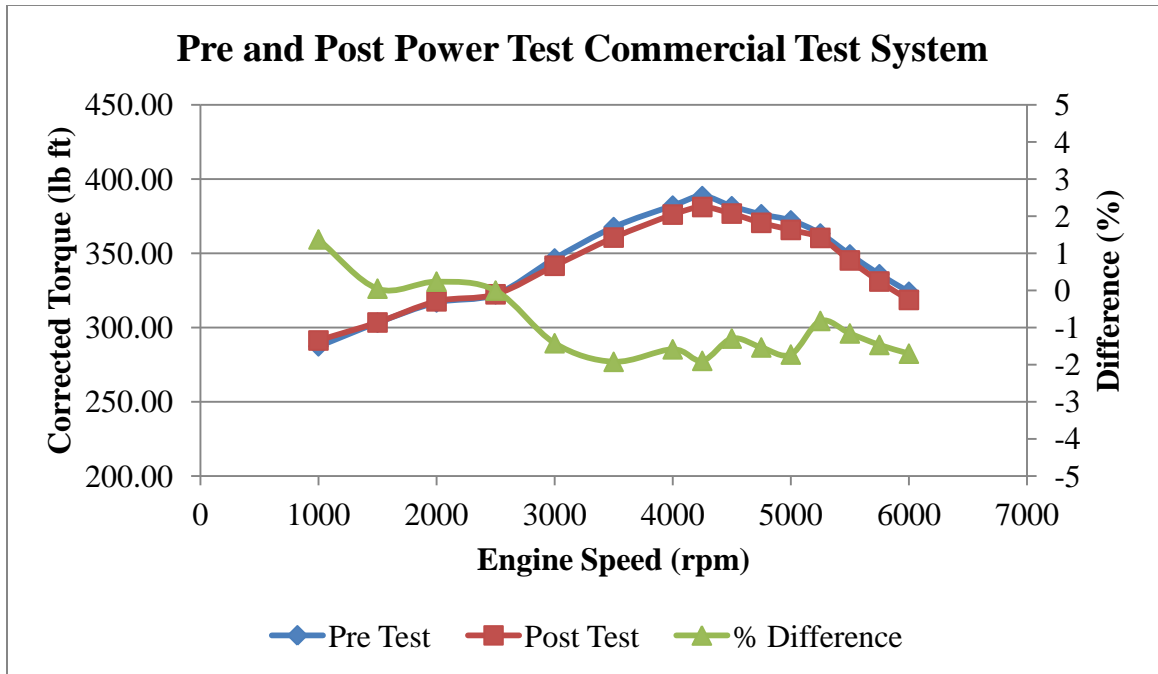
Figure 4.7: Pre vs. Post Test Power Test from Commercial Test Cell.

There are some minor differences in the performance of each of the engines, which will be true with any two engines. However, the general trend in the loss of output torque over the course of the test is similar.

The pre and post test corrected torque data displayed in Figure 4.6 was averaged from the corrected torque data shown in Figure 4.8 below. Similar data is not available from the commercial system since it is not capable of collecting long term data. The correction factor used to calculated corrected torque is based on the SAE Standard J1349 [32]. Variables such as air temperature, pressure and humidity are accounted for in the correction. The transients visible during the speed transitions are not included in the averaged data shown in Figure 4.6.
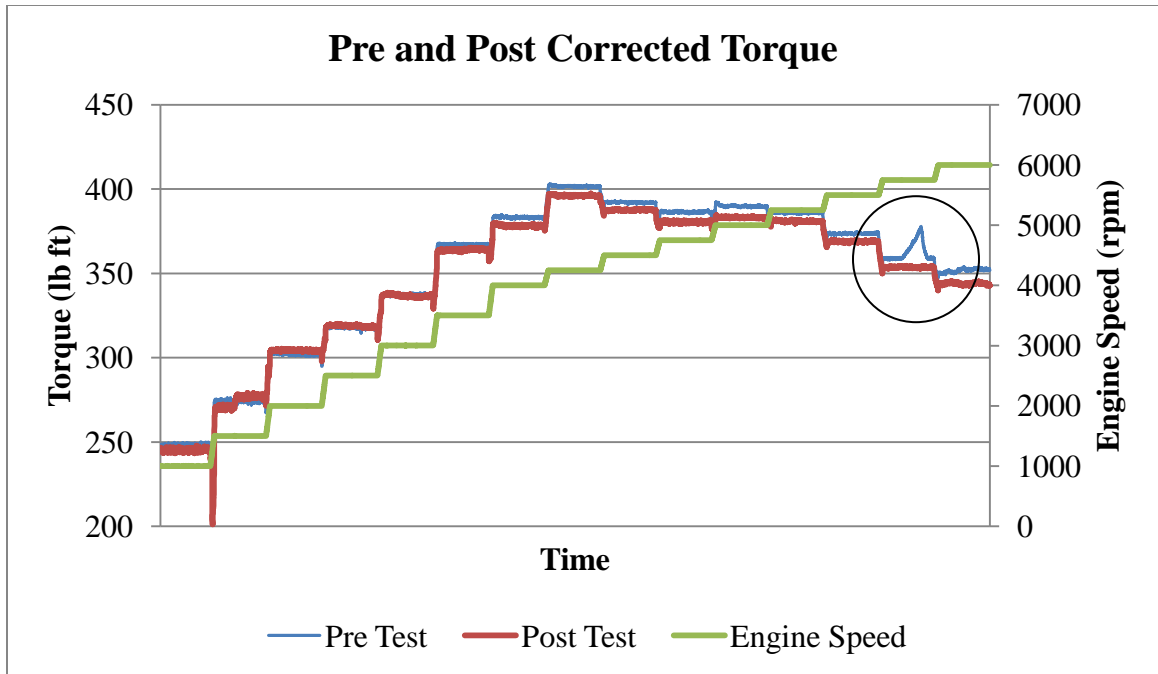
Figure 4.8: Pre vs. Post Test Power Test Corrected Torque Data.

During the 5750 rpm test point, the pre test torque curve shows an anomaly in torque values. This is the result of a humidity measurement device skewing the correction factor. The raw data showing uncorrected and corrected torque, as well as vapour pressure, which is calculated from the humidity measurement, is shown in Figure 4.9 for the period in question.

From Figure 4.9 it can be seen that the actual torque curve is relatively flat, while the corrected torque and vapour pressure curves both spike at the same time. It is this change in vapour pressure that is responsible for the corresponding spike in the corrected torque, since the correction factor takes into account vapour pressure. This error that was detected in the vapour pressure data would not have been found in our commercial test cell, since they do not have the ability to store long term data sets.
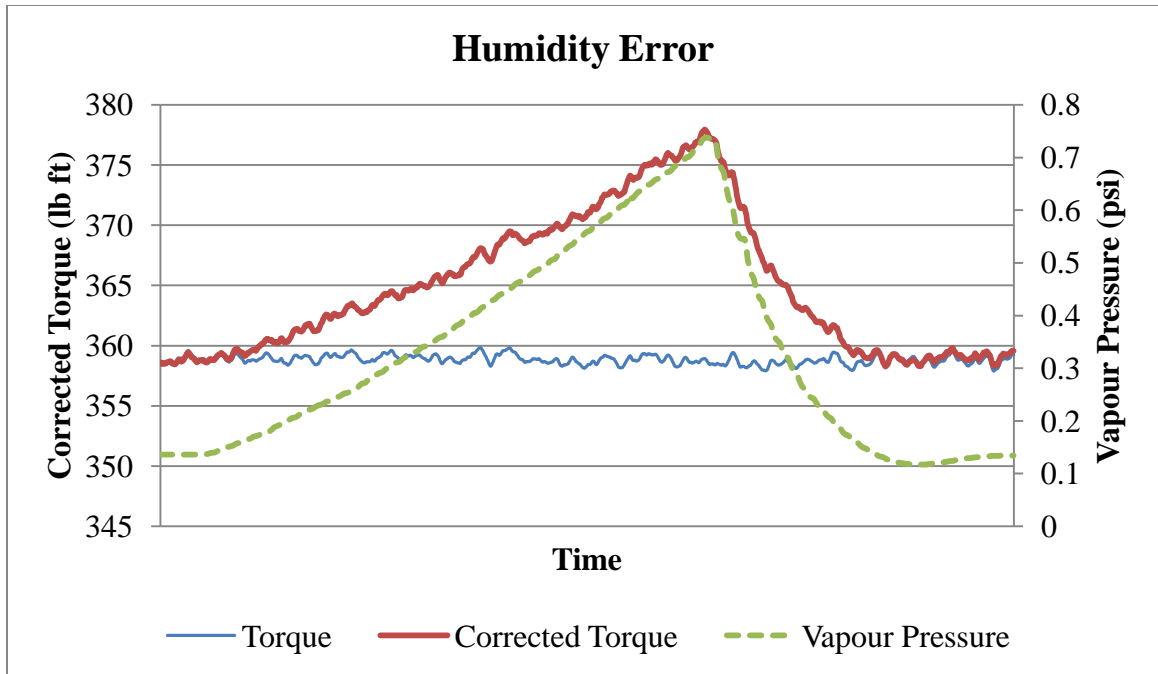
Figure 4.9: Power Test Humidity Error.

The power test for this engine type is a sweep of engine speeds from 0 to 7000 rpm. However, the power test curves plotted show speeds only up to 6000 rpm. During the pre-test power test, an alarm condition was triggered which caused the engine to shutdown at 6250 rpm. This shutdown was the result of an incorrect alarm configuration prior to the start of the test. Rather than paste the two data sets together, this will be used to show an example of a shutdown alarm sequence. The actual shutdown sequence performs the following operations simultaneously:

1. Ramp engine speed to 1250 rpm in 20 seconds
2. Ramp throttle position to 0% in 10 seconds
3. Sound buzzer

There are two alarms that are shown active in Figure 4.10. They are the AlarmActive and ShutdownActive alarm which were both triggered by an engine over speed condition. The chart is configured so that time 0 coincides with the moment the ShutdownActive alarm is triggered. The alarm values are scaled from 1 to 100 for legibility. AlarmActive is a warning alarm that triggers prior to the shutdown alarm. Both alarm values return to zero as soon as the engine speed is lowered below each alarm trigger point. The shutdown sequence continues to execute until completion, even after the alarm condition has ended. This ensures that the engine is brought down to a safe operating point until a technician acknowledges the alarm.
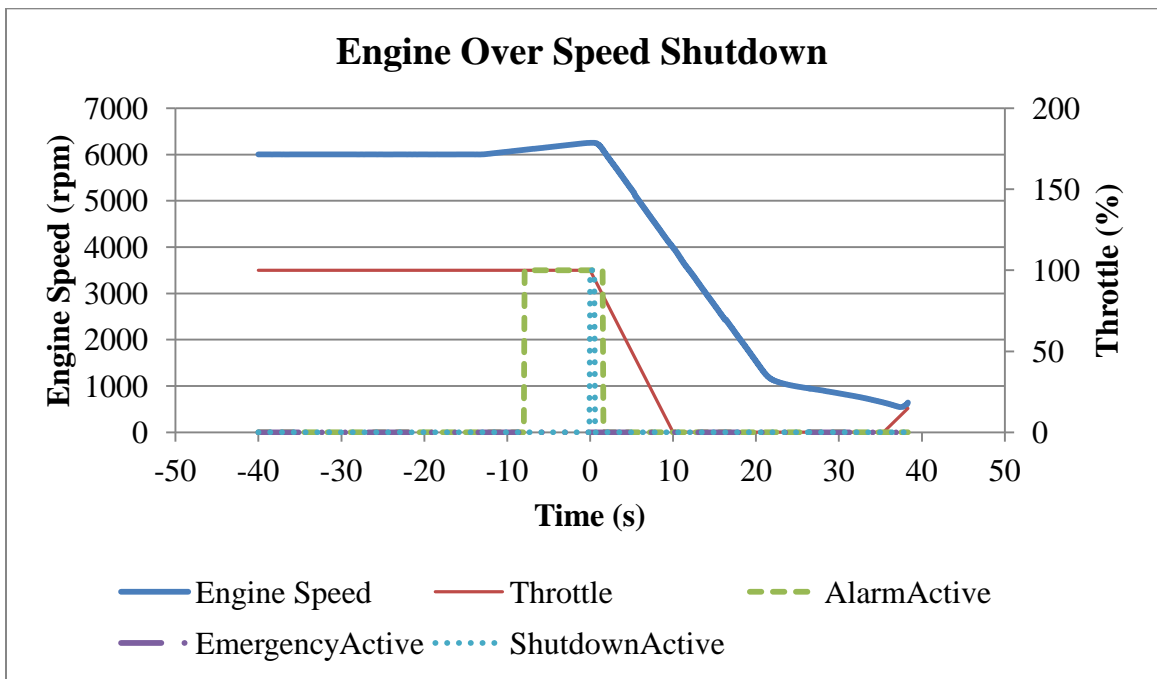
Figure 4.10: Engine Over Speed Shutdown.

4.4.2 Engine Fatigue Test Specification

An engine fatigue durability test was performed on the engine. This test is designed "to evaluate the engine's robustness to structural fatigue caused by repetitive

mechanical loading at high engine speeds and cylinder pressures" [33]. This is one of the most common tests performed in the eddy current dynamometer test cells at PERDC. The test is executed as a number of cycles of a sequence. The steps of the engine fatigue test sequence are illustrated in Figure 4.11 [33].
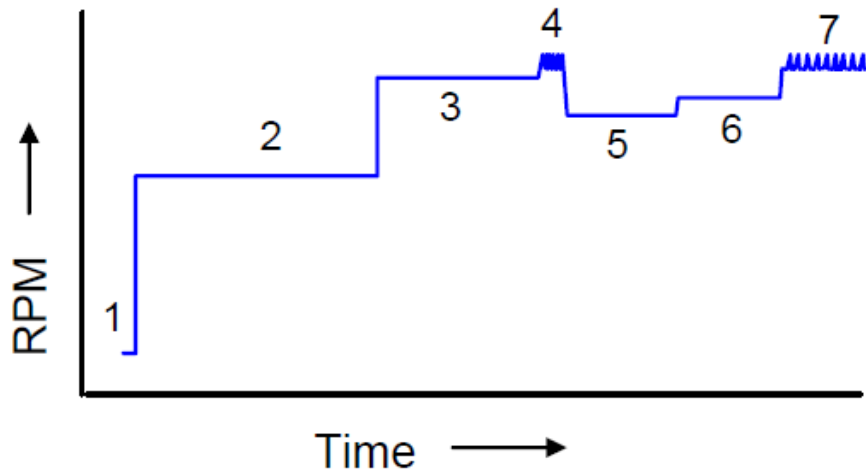
Figure 4.11: Engine Fatigue Test Sequence.

The individual steps from Figure 4.11 are summarized below:

1. Idle

2. Peak Torque, Full load

3. Peak Power, Full load

4. High speed oscillation, Light load

5. Intermediate speed #1, Full load

6. Intermediate speed #2, Full load

7. High speed oscillation, Full load

4.4.3 Engine Fatigue Test Results

Forty cycles of the sequence described above were completed successfully during the engine fatigue durability test. The data from a number of these cycles will be

analyzed to show long term stability of the data acquisition and control system. One of the objectives will be to show that the drop in engine torque from the pre to post test power test was not related to the measurement system, but to the gradual engine degradation. The test cell, in which this data was collected, uses a Honeywell PID temperature controller for both the engine coolant and oil temperature control loops. These devices run in automatic mode with a manual setpoint. The setpoint for the coolant temperature was 200 degrees Fahrenheit and the oil temperature setpoint was 265 degrees Fahrenheit.

Figure 4.12 shows the controlled coolant temperature. If the engine does not produce enough heat to exceed the setpoint, the temperature will not be controlled. At the beginning of each cycle, the engine is shut off. This is the reason the coolant temperature is low at the beginning of the cycle. At the end of the cycle, the engine is brought down slowly which accounts for the slow decline in temperature.
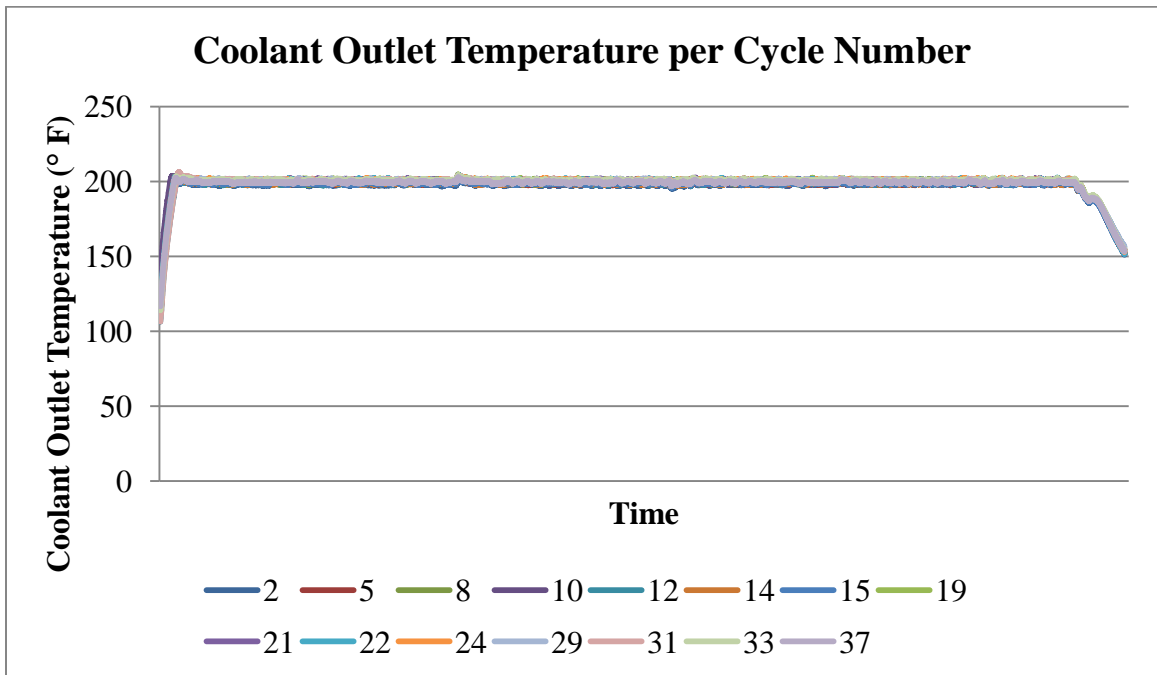
Figure 4.12: Coolant Outlet Temperatures by Cycle.

The average coolant temperature from a portion of each cycle was extracted and is shown in Figure 4.13. The average is targeting the desired temperature of 200 plus or minus 1 degree Fahrenheit. There is a definite shift in temperature at cycle 19. There could be multiple reasons for this shift in temperature. Some of these will be discussed after a review of the engine oil temperature charts.
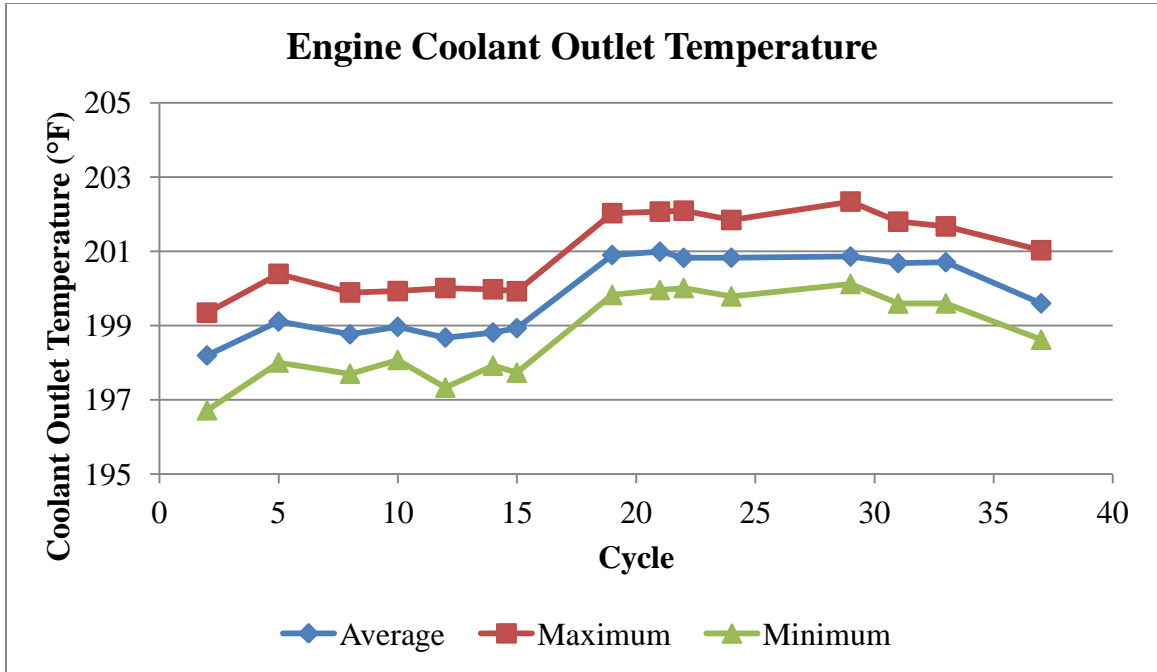
Figure 4.13: Coolant Outlet Temperature Averages.

The cycle by cycle engine oil temperatures are shown in Figure 4.14. This temperature is not under active control for most of the test, since the engine oil does not often reach the limit value of 265 degrees Fahrenheit.
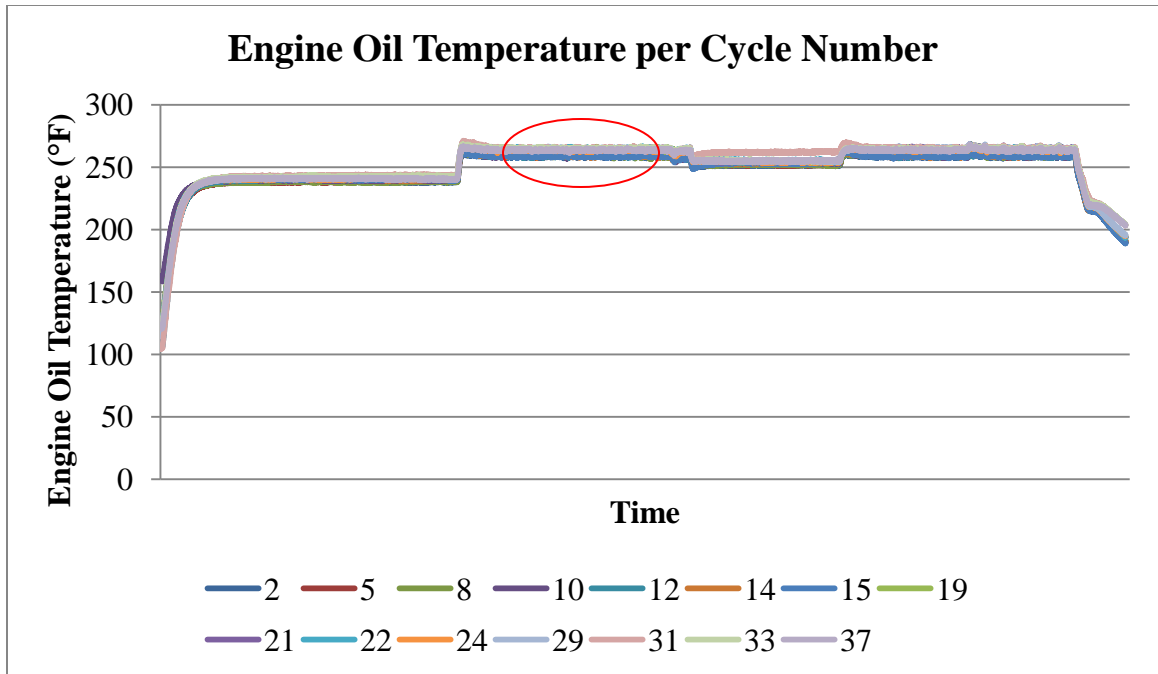
Figure 4.14: Engine Oil Temperature by Cycle.

One portion of the curve, highlighted in the chart, where engine oil temperature does reach 265 degrees Fahrenheit on most cycles is shown in Figure 4.15. This curve shows the same shift in temperature at cycle 19, as the coolant temperature curve in Figure 4.13. The most probable reason for the shift in temperature was a manual change in setpoint. There is an engine service at cycle 15 that requires the engine technician to manually adjust these temperatures. It is believed that the target temperatures were not returned to the original setpoints after the service was completed. The other possibility is a change in position of the engine cooling fans. Fans are aimed directly at the engine during testing. Changing the angle of the fans can have an effect on these temperatures.
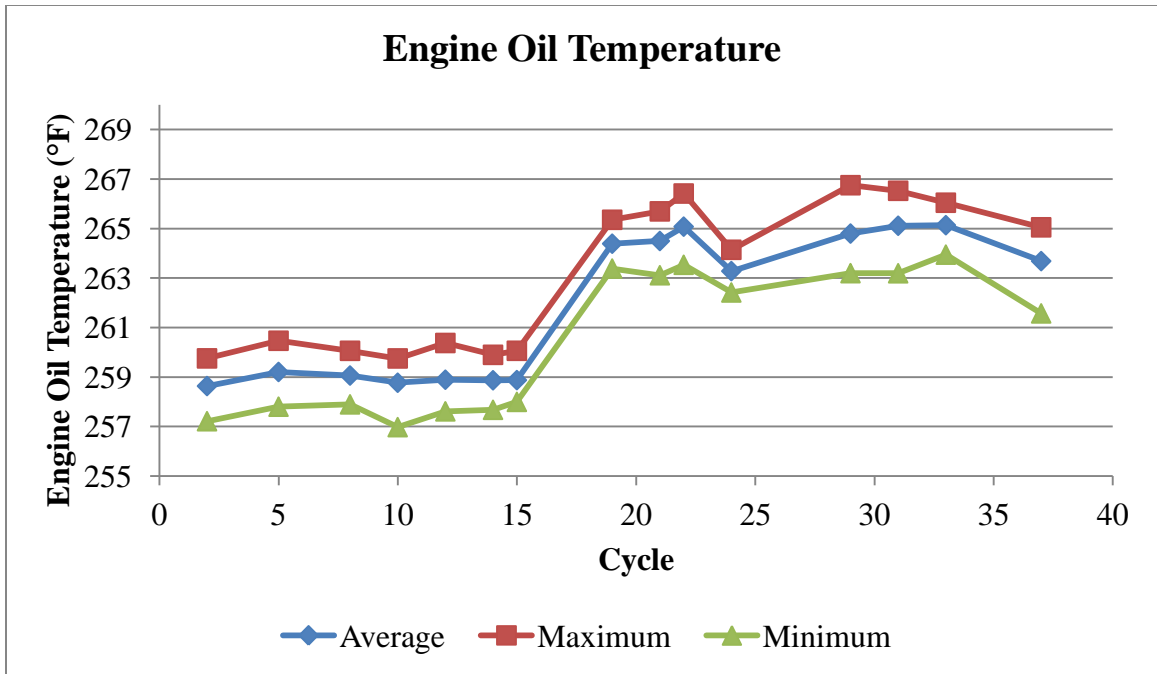
Figure 4.15: Engine Oil Temperature Averages.

The engine speed for each cycle is shown in Figure 4.16. The repeatability of engine speed from cycle to cycle is very high, so it will not be discussed further.
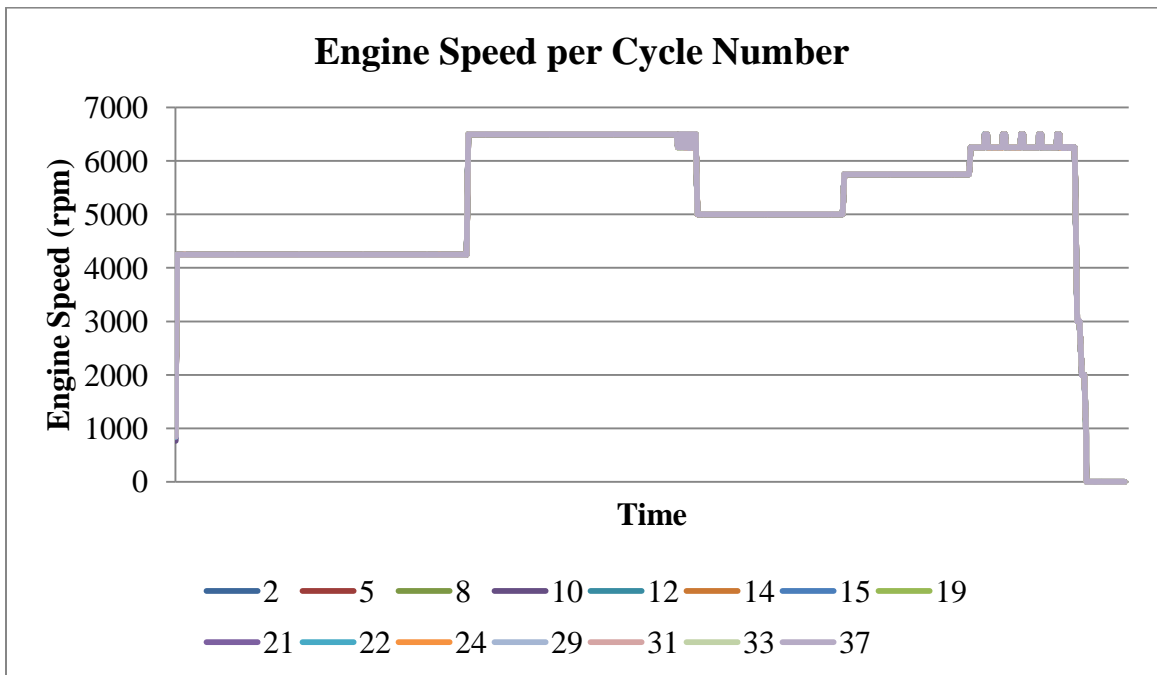
Figure 4.16: Engine Speed by Cycle.

The corrected torque measurements, for each of the cycles, are shown in Figure 4.17. A lot of cycle to cycle variation can be seen in the corrected torque. The same issue, discussed previously, concerning the humidity measurement device is seen on many of the cycles. The sections of the EFT cycle are labelled in Figure 4.17 as 2, 3, 4, and 5.
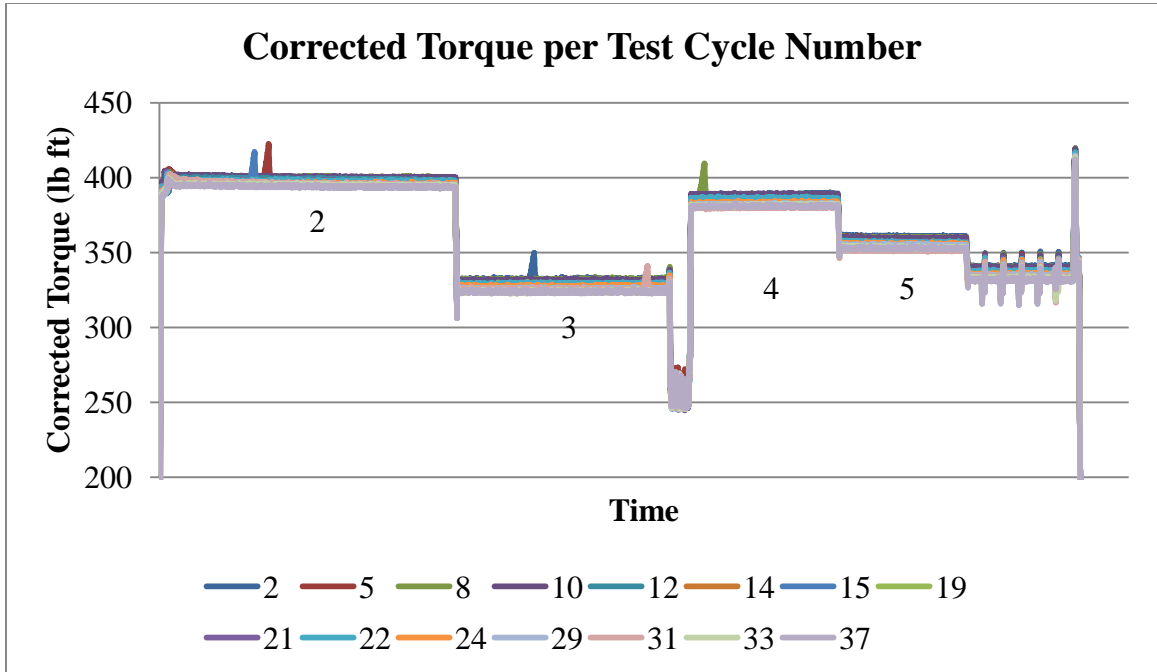
Figure 4.17: Corrected Torque by Cycle.

Figures 4.18 to 4.21 show the corrected torque for each section of the EFT cycle. Each of these figures shows that the variation is not random. There is a definite decrease in output as the cycle number increases. This indicates that the drop in engine torque is gradual, over the course of the test, and most likely caused by decreased engine performance after hours of testing.
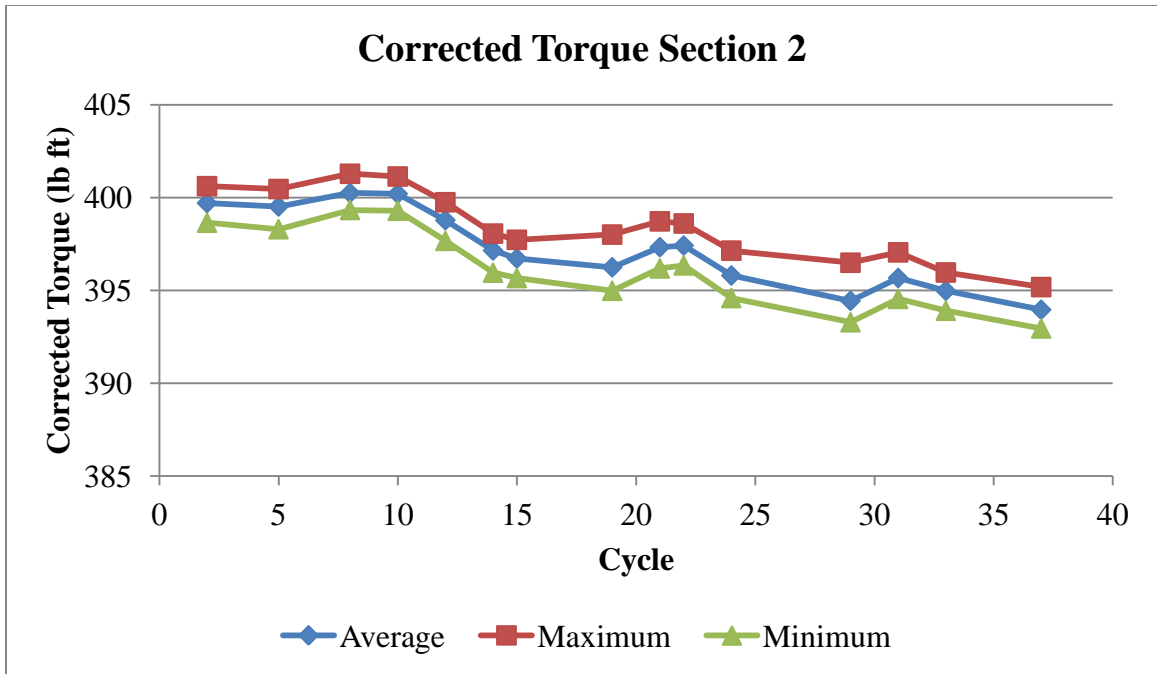
Figure 4.18: Corrected Torque Average Section 2.
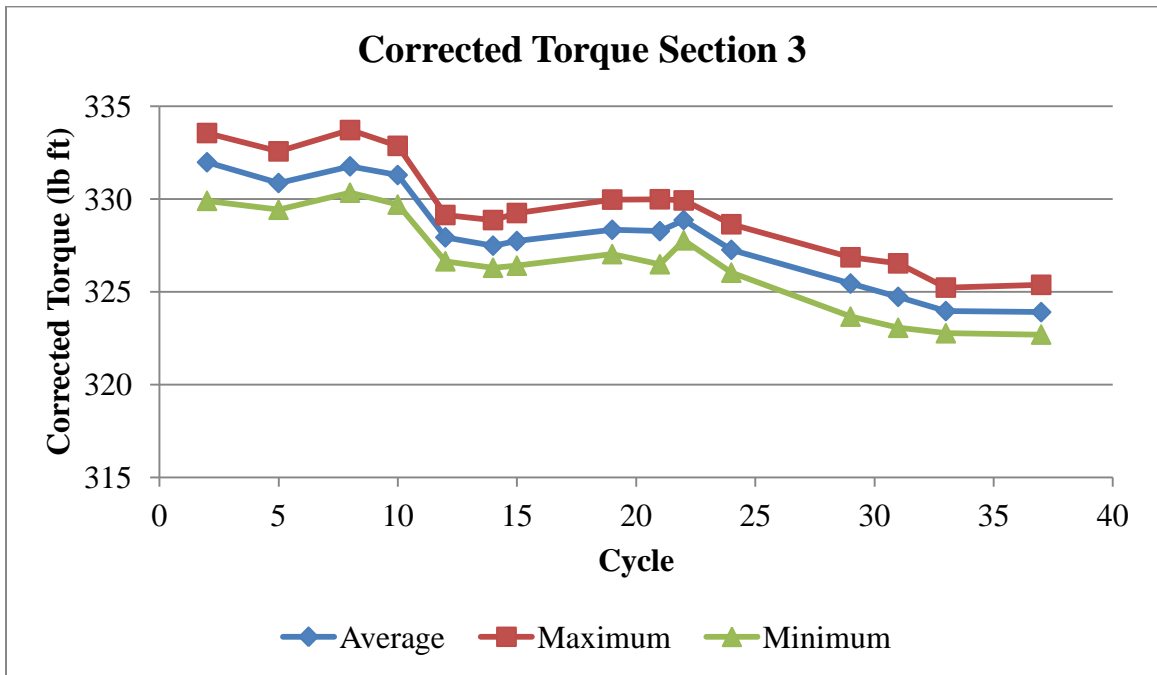
Figure 4.19: Corrected Torque Average Section 3.
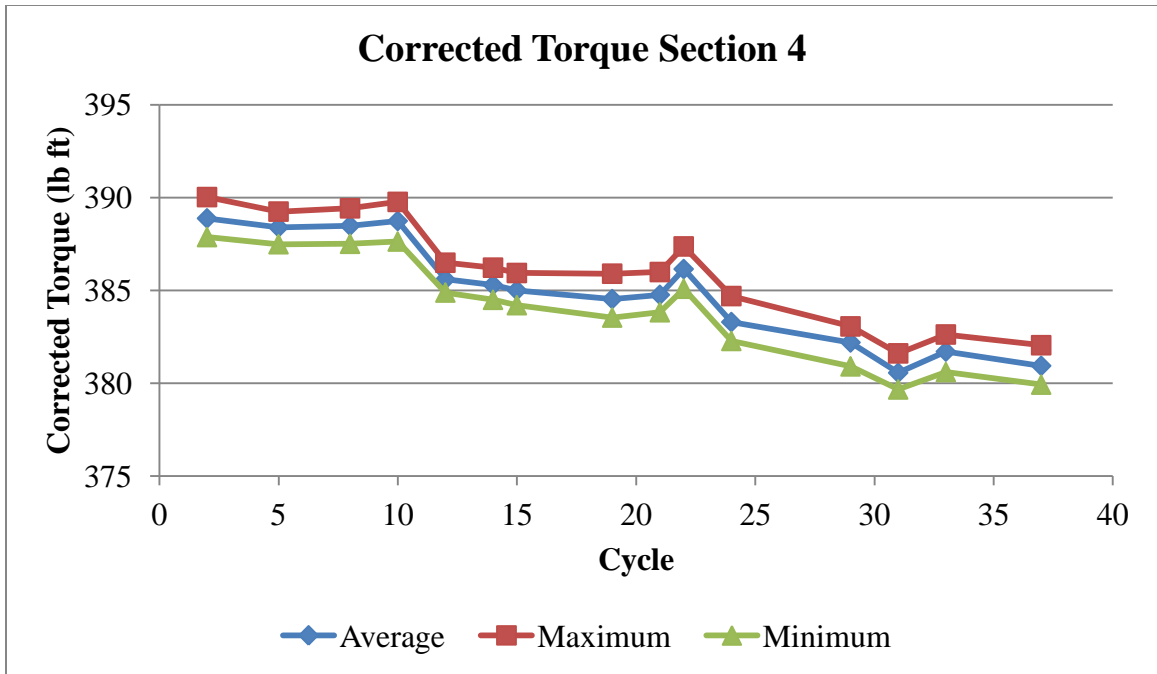
Figure 4.20: Corrected Torque Average Section 4.
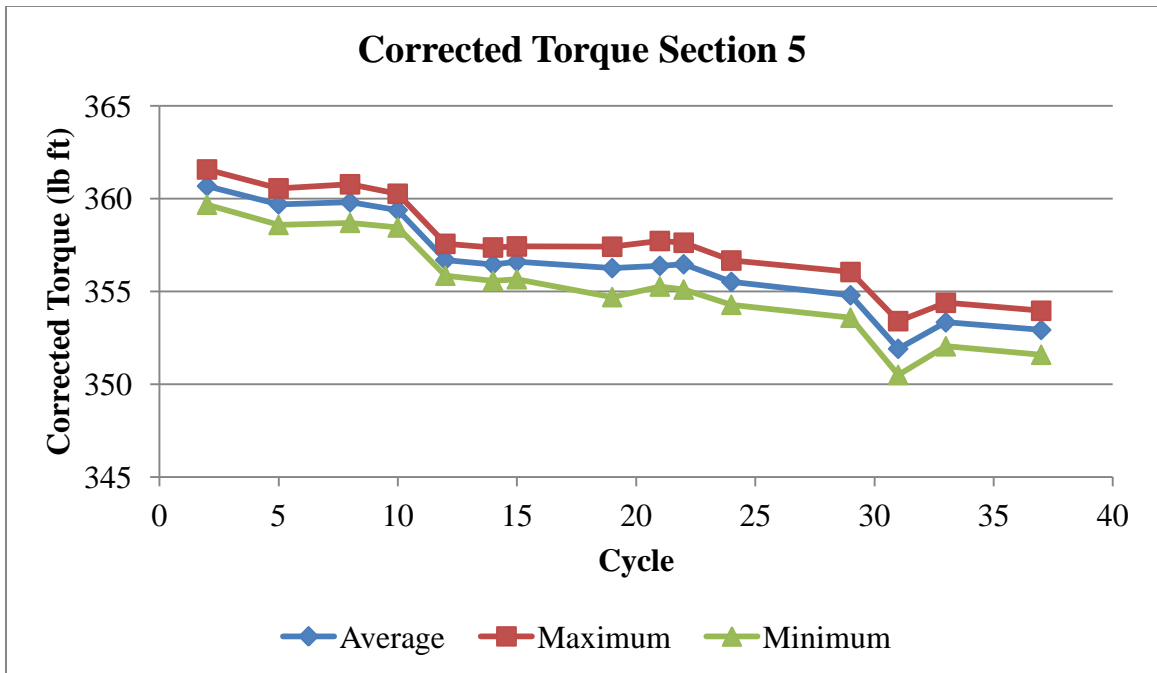
Figure 4.21: Corrected Torque Average Section 5.

4.5 Dynamometer Vibration

In order to show how the dynamometer vibration condition monitoring system behaves at different portions of the EFT test, a single cycle was selected from the EFT test.

The vibration monitoring sensor is used to protect the dynamometer from excess vibration that could result in bearing failure or more catastrophic damage. Figure 4.22 shows vibration data from the Ifm Efector sensors mounted on the dynamometer during the EFT test. The driveshaft would be connected to the front of the dynamometer. From Figure 4.22, it can be seen that there is a resonance that occurs in the system at an engine speed of 5750 rpm. The resonance is very pronounced on the front of the dynamometer and is highlighted with a red circle in the figure. This is believed to be caused from using a long driveshaft (42 inch). PERDC is in the process of modifying the bedplates that the engine and dynamometer rest on, to reduce the length of the driveshaft required.
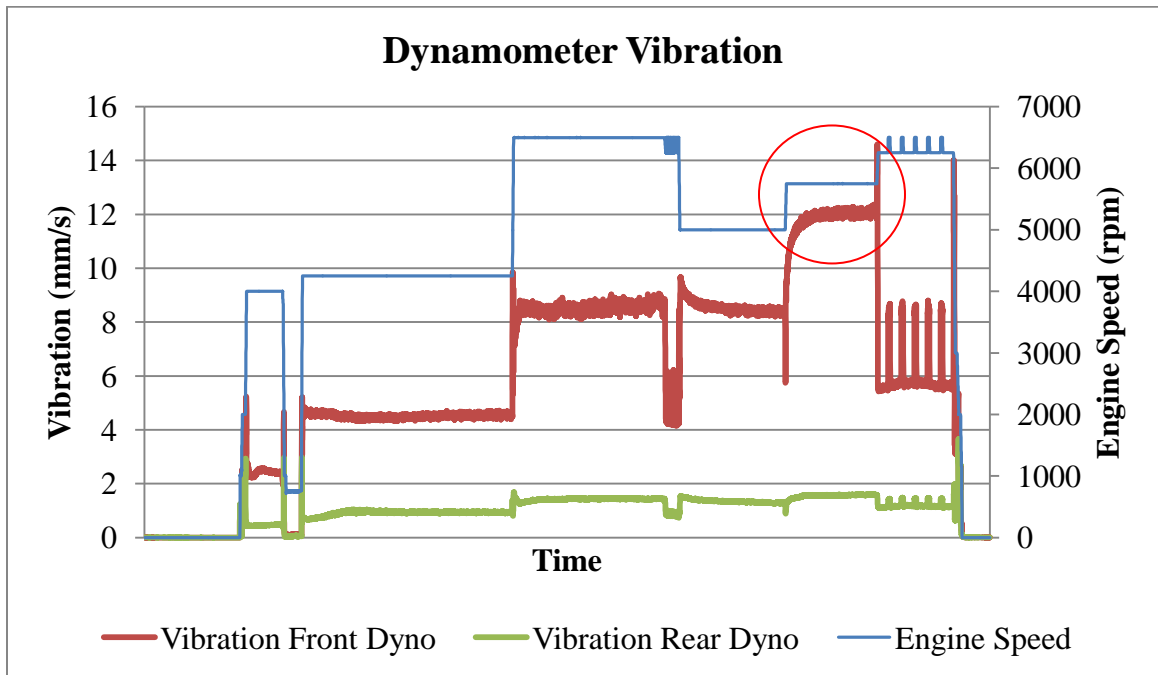
Figure 4.22: Dynamometer Vibration.

CHAPTER 5: CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

This thesis, in part, targeted to challenge Martyr and Plint's statement that it would be beyond the capabilities of any one person to develop an engine test control system [1]. By completing this thesis work, it was demonstrated that an engine test system could be created by an individual. The completed engine test cell control system was installed in three of the PERDC eddy current dynamometer test cells. A number of specific objectives were set and each of them will be addressed. Many aspects of the control system created exceeded the expectations of the engineers and engine technicians.

1. An engine testing control system was successfully developed, installed and tested at the PERDC facility. It has the capability of running automatic test sequences in real time. It is currently being used in three eddy current dynamometer test cells twenty four hours a day, seven days a week.

2. A GUI was developed using a template from existing applications that the PERDC personnel are already comfortable operating, thus making the GUI very intuitive.

3. The engine test cell control system was developed with the cost of approximately one tenth of that of the commercially available systems. The majority of the cost is contained in the hardware which is flexible in configuration.

4. The data output from the control system is superior to the commercially available systems. It also includes an analysis package that is not available

with the other systems. The 10Hz data logging is now a best practice at

PERDC.

5. The engine control system that was developed is easily learned. The

operation is made possible by ordinary algebraic syntax and simple logic.

There is no cryptic or foreign syntax needed to create points or test sequences.

All of the tests and data used in the results portion of this thesis were created

and collected by PERDC personnel.

This research could not have been completed as efficiently without the

AngelScript library, muParser, or the SQLite database used in the development of the

software. The authors of each of these libraries have generously provided their work in

source code form for anyone to use. None of them could have envisioned that their work

would end up being used as part of a real time engine test cell control system.

In a way, Martyr and Plint were ultimately correct; this work could not be

completed by one person no matter how well versed they are in engine testing. What

they might have failed to realize is that generous people have freely provided

extraordinary tools that can be used to assemble an engine test control system. By

harnessing the work made freely available by others, and incorporating it into the

architecture designed for this thesis, one person was able to finish this project

successfully.

5.2 Recommendations

Every one of the commercial engine test systems that exist today started with a

very simple concept and progressed into the massive works that Martyr and Plint

describe. This thesis marks the completion of version 1.0 of an engine control system

code-named "Katerina". More development on this project is currently being pursued. Some of these ideas will be discussed below for the next person willing to challenge Martyr and Plint in the dynamically changing world of engine testing.

The calculation of points was done using an abstract base class that was inherited by a number of different point classes. This concept can be extended to include complex calculations, such as a PID type points and many others. Doing this would provide better encapsulation of data. It would also allow custom dialogs to be opened that access all of the points as a group.

The choice to use QNX Neutrino as the RTOS for this thesis added some additional cost to the project. There is an ongoing project to provide a real time kernel for Linux. This project is currently in the form of a patch, named Preempt-RT. Using Linux was always a vision for this engine testing control system and the code was written to be easily ported when the Preempt-RT patch was complete. The only code that would need to be modified is the timer objects. Part of the Preempt-RT patch includes support for high resolution timers that are compatible with the QNX timers, since they are both based on the POSIX standard. The current home for the Preempt-RT project is located at www.osadl.com.

One of the original ideas for this project was to implement it on inline test stations in manufacturing facilities. The operation is slightly different, since the data is collected only when a part arrives at the test station. The script environment could easily be extended with functions that start and stop data acquisition, as well as analyze the collected data.

REFERENCES

1. Martyr A. J., Plint M. A., "Engine Testing", Elsevier, 2007.

2. Horiba Automotive Test Systems, "Operating Manual Eddy Current Dynamometer of the WT Series", 2007.

3. Heywood J. B., "Internal Combustion Engine Fundamentals", McGraw-Hill, 1988.

4. Wilson J. S., "Sensor Technology Handbook", Newnes, 2005.

5. Dyne Systems, "Dyn-Loc IV Digital Dynamometer Controller Installation And User Manual", 2001.

6. Seitz J., "Designing With Thermocouples", Application Note AN-1953, National Semiconductor, 2009.

7. Measurement Computing, http://www.mccdaq.com/PDFs/specs/temperature_measurement.pdf, accessed on 16th November 2011.

8. Ifm Efector, http://www.ifm.com/, accessed on 21th November 2011.

9. Wright M., Peltier H. R., "Applying Distributed Architecture To The Modernization Of US Air Force Jet Engine Test Cells", IEEE, AUTOTESTCON, 2008, pp.480-483.

10. Eisert D.E., Bosch R.A, Jacobs K.D., Kleman K.J., Stott J.P., "A New Real-Time Operating System And Python Scripting On Aladdin", IEEE, PAC, 2003, pp. 2373-2375 vol.4.

11. Working Group On Standardisation Of Application Systems Interface Specification, "ASAP3-MC", 1998.

12. Vector CANTech Inc., http://www.vector.com/vi_candb_en.html, accessed on Nov 20th 2011.

13. United Electronics Industries, "UEI PowerDNA I/O Hardware Helps Tinker AFB Achieve Design Goals For New Jet Engine Test Cell", http://www.ueidaq.com/media/static/apps/appnote-028_tinker.pdf, accessed on 16th November 2011.

14. United Electronics Industries, "DNA-PPCx-1G PowerDNA Gigabit Ethernet I/O Cube User Manual", 2009.

15. Josuttis N. M., "The C++ Standard Library", Addison-Wesley, 1999.

16. QNX Software Systems, "System Architecture 6.4", 2008.

17. AngelScript, http://www.angelcode.com/angelscript, accessed on 19th November 2011.

18. SQLite, http://www.sqlite.org, accessed on 17th November 2011.

19. muParser, http://muparser.sourceforge.net, accessed on 19th November 2011.

20. van Breemen A.J.N., "Scripting Technology And Dynamic Script Generation For Personal Robot Platforms", IEEE, IROS, 2005, pp. 3487-3492.

21. Jiang C., Liu B., Yin Y., Liu C., "Study On Real-Time Test Script In Automated Test Equipment", IEEE, ICRMS, 2009, pp. 738-742.

22. Clark D.L., "Powering Intelligent Instruments With Lua Scripting", IEEE, AUTOTESTCON, 2009, pp. 101-106.

23. Downey A.E., "'ATG' Test Generation Software", IEEE, Int. Test Conf., 1989, pp. 829-837.

24. Chapra S.C., Canale R.P. "Numerical Methods For Engineers", McGraw-Hill, 1988.

25. Cooper D.J, "Practical Process Control",

http://www.controlguru.com/pages/table.html, accessed on 19th November 2011.

26. Vukic Z., Kuljaca O., "Lectures on PID controllers",

http://arri.uta.edu/acs/jyotirmay/EE4343/Labs_Projects/pidcontrollers.pdf,

accessed on 19th November 2011.

27. Krten R., "Getting Started With QNX Neutrino 2, A Guide For Realtime Programmers", Parse Software Devices, 2001.

28. Throop M., "High Accuracy Dynamometer Torque Calibrations: A Systems Approach", SAE paper No. 2004-01-1792.

29. Ge H., Zhou X., "Design Of A Multivariable Controller For An Engine Eddy Dynamometer System", IEEE, Int. Conf. CMCE, 2010, pp. 476-479.

30. Bunker B.J., Franchek M.A., Thomason B.E., "Robust Multivariable Control Of An Engine-Dynamometer System", IEEE, Transactions On Control Systems Technology, 1997, pp. 189-199.

31. King P.J., Burnham K.J., James D.J.G., Norton J., Sharpe S.R, "Implementation Of A Self-Tuning Controller To The Dynamometer Torque Loop Of An Engine Test Cell", IEEE, Int. Conf. On Control, 1991, pp. 110-114 vol.1.

32. SAE J1349, "Engine Power Test Code – Spark Ignition And Compression Ignition – As Installed Net Power Rating", http://standards.sae.org/j1349_201109, accessed on 19th November 2011.

33. Herr M., "Summary Of Dynamometer Durability Test Procedures", Ford Internal Document, 2009.

34. Lua Users, http://lua-users.org/wiki/FloatingPoint, accessed on 21st November 2011.

35. Wikipedia, http://en.wikipedia.org/wiki/Red-black_tree, accessed on 21st November 2011.

36. Cippola R., Smale J., "B-Curve Parameter Monitoring", Ford Internal Document, 2002.

37. Lasota P., "Multi-Cell Monitoring System", Horiba, 2007.

38. Sensoray, http://www.sensoray.com, accessed on 21st November 2011.

39. National Instruments Measurement Hardware Driver Development Kit, http://sine.ni.com/nips/cds/view/p/lang/en/nid/11737, accessed on 21st November 20011.

40. Mustang Dynamometer, http://mustangdyne.com/mustangdyne/products/engine-dynamometers/throttle-actuators, accessed on 21st November 2011.

41. Allen Bradley, "Ultra 3000i", http://ab.rockwellautomation.com/Motion-Control/Ultra3000-Servo-Drive, accessed on 21st November 20011.

42. Kelly C., Fountaine A., "Katerina Software Users Manual", Ford Internal Document, 2011.

APPENDIX A: POINTS

Data that has been acquired using a data acquisition board is normally in the form of a voltage, current, count, frequency, etc. While these data points are valuable, it would be much more intuitive to interpret the data if it was transformed into a pressure, temperature, speed, torque, etc. In order to accomplish this transformation, a number of a point classes have been developed. The classes range from a simple linear transformation to a generic mathematical equation parser, from the open source project muParser [19].

Each one of these classes, outputs a single value that is called a point. It would be relatively easy to have multiple data types for points using a variant data type. A variant in its simplest form is a union of multiple data types such as integer, float, double, etc. However, the decision was made early on in the project that the only data type to be used would be double. A double is a 64 bit value with 52 bits assigned to the fractional component, 1 bit for sign and 11 bits for the exponent. This single data type exceeds the precision of a 32 bit float as well as a 32 bit integer. There is often the belief that integers will not be exactly represented in floating point format [34]. When storing an integer numeric value of 1 in a double, it is exactly defined and can be used in comparison operations. This is important since a large part of this project uses logical comparisons.

A.1 Point Containers

Having defined what is meant by a point, point transformations and point data types, a place to store these values is needed. In many programming languages there is an associative container. In the C++ standard library one of these associative containers is the std::map. This container will allow one data type as a lookup for another data type. In this project, a global std::map is created that maps std::string types to a double data

type. Most implementations of the std::map are based on a Red Black Tree algorithm and have lookup and insertion performance of O(log n) [35]. With logarithmic performance, a lookup or insertion with 1,000 points would have a worst case performance of about 3 searches. In the very unlikely case where 10,000 points have been defined, a lookup would be about 4 searches. It should be noted that the std::map is a sorted container. The insertion order is not the order that would be found when traversing the container in a loop.

Using the std::map with a string as the lookup, allows indexing of points using point names. This is the foundation for the real time database. This allows the user to define points using English names such as Speed or Torque. The standard programming language restrictions apply for point names; being that they must start with a letter and cannot contain spaces. This restriction is required, since each of the points will be exposed directly to the scripting engine. By exposing the points to the script engine, the user has the ability to modify a point's value through the execution of custom script code.

A.2 Conversion Classes

The actual point transformation starts with a base class named CConversion. This is the base class for a number of different types of transformation classes. It contains very little code but provides the basic structure that is required for any conversion to be equated. The class contains a pure virtual function named Convert that takes a reference to the std::map of data points. The basic definition is shown below.

```cpp
class CConversion{
public:
        CConversion();
        virtual ~CConversion();
        virtual void Convert(DATAPOINTS &db)=0;
};
```

This allows a container of CConversion objects to be created that can hold any of the derived transformation classes. This polymorphic behaviour allows us to abstract many conversion types, and simply call the Convert function on each of them. The actual implementation of each of the Convert functions will be completely different for each derived conversion class. The end result of the convert function is to update a single point in the real time database. Each derived conversion class will have its own unique set of variables and functions needed to perform the transformation.

A.2.1 CLinear

The CLinear conversion class performs a standard linear transformation of the form $y = mx + b$. This provides the ability to scale and offset a point. This conversion type could be used in a number of different ways. A simple copy of a point could be made by setting the scaling to 1 and the offset to 0. If a sensor is known to be linear through a range of interest, the proper scaling and offset values could be used to create a new point. The definition of the CLinear class shows that it inherits from the CConversion class and implements the Convert function.

```
class CLinear : public CConversion{
public:
        CLinear();
        virtual ~CLinear();
        virtual  void Convert( DATAPOINTS &db);
        double m;
        double b;
        double max;
        double min;
        std::string PointName;
        std::string RawName;
};
```

The CLinear class contains the required scaling and offset variables m and b. In addition, hard bounding variables min and max limit the magnitude of the value. There

are two string variables defined that are names of points in the real time database. The

variable RawName is the source point and PointName is the output point of the linear

transformation. The implementation of the Convert function for CLinear is shown below.

```cpp
void CLinear::Convert(DATAPOINTS &db){
        double temp = 0.0;

        temp = m*db[RawName] +b;
        if (temp>max) {db[PointName] = max;}
        else if (temp < min) {db[PointName] = min;}
        else {db[PointName] = temp;}
}
```

A.2.2 CScratchPad

The CScratchPad conversion class does not perform any mathematical

conversion. It was created to allow a point to exist that could be used to control a portion

of the script. Typically this would be the script engine or a driver for a gas analyzer. An

example of a scratch pad point would be the coefficients of a PID loop.   There is no

calculation associated with these points; they are simply entered by the user to be used by

the PID calculations. The definition of the class is shown below which has only a point

name variable added to the base CConversion class.

```cpp
class CScratchPad: public CConversion{
public:
        CScratchPad();
        virtual ~CScratchPad();

        virtual  void Convert( DATAPOINTS &db);
        std::string PointName;

};
```

The implementation of the convert function is a simple copy of the point value to

itself, as shown in the code below. This is done to ensure that the point is created within

the std::map. When an index is requested from a std::map that does not exist, the default

behaviour is to create a new object in the container.

```
void CScratchPad::Convert(DATAPOINTS &db){
        db[PointName] = db[PointName];
}
```

A.2.3 CInterpolate

CInterpolate is a class that performs an interpolated lookup conversion. The use

of a lookup table has many applications, but primarily as a calibration table for sensors.

For example, if a pressure sensor with a voltage output is calibrated at ten unique points

over its operating range, a table would be created that has ten voltage values each

associated with a unique pressure value, in engineering units such as psi. In the

definition of the CInterpolate class there is the addition of the interpolation table,

coincidentally named table. This table is a std::vector of CDoublePair objects which is a

simple structure shown below.

```
class CInterpolate : public CConversion{
public:
        CInterpolate();
        virtual ~CInterpolate();
        virtual  void Convert( DATAPOINTS &db);
        std::vector<CDoublePair*> table;
        std::string PointName;
        std::string RawName;
        double max;
        double min;
};

struct CDoublePair{
        double Raw;
        double EUUnits;
};
```

The implementation of the convert function uses the RawName variable, as the

source point, to lookup in the interpolation table. When a value is found that exceeds the

source point, the lookup index is recorded and the actual interpolation is performed. Interpolation is not performed beyond the boundaries of the table. A saturation function is implemented that limits output values to include only values defined by the user in the table.

```cpp
void CInterpolate::Convert(DATAPOINTS &db)
{
        unsigned int i=0;
        double temp=0.0;
        double sum=0.0;
        for( i=0;i<table.size();i++)
        {
                if (table[i]->Raw >= db[RawName]) break;
        }

        if(i==0) temp= table[i]->EUUnits;
        else if(i== table.size() )  temp= table[table.size() -1]->EUUnits;
        else  temp= table[i-1]->EUUnits +
        ((db[RawName] - table[i-1]->Raw)/(table[i]->Raw - table[i-1]->Raw))
        * (table[i]->EUUnits - table[i-1]->EUUnits);

        if (temp>max) {db[PointName]= max;}
        else if (temp < min) {db[PointName]=min;}
        else {db[PointName]=temp;}
}
```

A.2.4 CFormula

The CFormula conversion class is the most flexible and powerful of all the conversion classes. The primary purpose of the class is to perform evaluation of mathematical equations. This is a very valuable feature, since it will allow users to enter a formula in a standard algebraic format. For example to calculate observed brake horsepower, the equation would be OBhp= (EngSpeed * Torque) / 5252. EngSpeed and Torque are points that are calculated from another point conversion object. This functionality is made possible through the use of the open source muParser project [19]. muParser is a mathematics parser engine that performs parsing and evaluation of complex algebraic equations. The CFormula class is shown below which contains a mu::Parser class named parser.

85

```
class CFormula : public CConversion
{
public:
        CFormula();
        virtual ~CFormula();
        virtual  void Convert( DATAPOINTS &db);
        virtual void Init();

        mu::Parser parser;
        std::string PointFormula;
        std::string Message;
        std::string Name;
        std::string PointName;
        std::string MonitorPoint;
        int Type;
        double max;
        double min;
        double MonitorValue;
        int AlarmStored;
        int error;
};
```

The implementation of the Convert function for CFormula is simply a call to the muParser objects Eval function. In order for this call to succeed, the muParser object needs to first be initialized. A separate function is used to initialize the muParser object during the instantiation of the CFormula object. This ensures that the equation is valid and also performs the one time parsing and conversion to byte code. The initialization function requires two pieces of information. First, the equation that is to be evaluated is required. The second item is a callback function to a variable factory. When muParser encounters an unknown variable during the parsing process, it will execute the callback function. This is implemented in a function named AddVariable. AddVariable simply returns a pointer to the requested point in the std::map.

```cpp
void CFormula::Init(){
        error=0;
        try
        {
                parser.SetVarFactory(AddVariable,&parser);
                parser.SetExpr(PointFormula);
                parser.Eval();
        }
        catch(Parser::exception_type &e)
        {
                error=1;
                std::cout  << "Equation Parser error for point or alarm: " <<
                PointName << e.GetMsg() << std::endl;
        }
}

void CFormula::Convert(DATAPOINTS &db){
        try
        {
                if(error ==0){parser.Eval();}

                if (db[PointName]>max) {db[PointName]= max;}
                else if (db[PointName] < min) {db[PointName]=min;}

        }
        catch(Parser::exception_type &e)
        {
                std::cout   <<  "Equation  error  for  point  or  alarm:  "  <<
                PointName << e.GetMsg() << std::endl;
        }
}

double *AddVariable(const char_type *a_szName, void *a_pUserData)
{
    return(&dataPoints[a_szName]);
}
```

A.3 CConversion Container

In the real time database, each of the CConversion objects is loaded into a std::vector container. The std::vector container is basically an array that has the ability to dynamically change size at runtime. It is not a sorted container, so the order in which objects are inserted is the same order in which they are traversed in a loop. This is important since it allows the definition of priority. If point "xyz" has a dependence upon a point "abc", then the point "abc" should be calculated before "xyz". This implies that

"abc" would be inserted into the vector before "xyz". In the current application, the priority or order of operations is defined by the user, using a numeric value. Many points could have the same priority. The current priority scheme uses numbers such as 10, 20, 30, 40 etc. The first points to be evaluated are given the highest priority which is number 10. Work is currently being done to automatically order the objects based on knowledge of their dependencies.

The choice to hold the CConversion objects in a separate container from the data values was intentional, to allow point objects to be updated dynamically at runtime. Since the point values themselves are stored in a separate container, they will not be invalidated if a CConversion object is deleted from memory. This is important since the script object will be holding pointers to the points in memory. The script engine has no knowledge of the CConversion objects. It simply knows of the output points stored in the std::map. This functionality is important, since it would be tedious to require the application to shutdown and restart when modifying parameters of the CConversion objects.

A.4 Retentive Points

Some points should be able to maintain value after shutting down and restarting the real time database. To implement this functionality, a property named retentive is assigned to each point. A point that has its retentive property set, will have its value updated in a SQLite database every ten seconds, as well as during a shutdown of the application. When the application is restarted, the last value written to the database will be the initial value for that point. This is important for a number of different reasons. One would be to maintain calibration values such as the PID coefficients for a control

loop. Another would be to store the state variables of a current running test, such as the step, step time, or cycles completed.

A.5 Point Editor

The discussion so far has been around the real time implementation aspect of points. A GUI application was created to enter the definition of each point. This dialog manages a table, named points, within a SQLite database. The basic dialog layout can be seen in Figure A.1. The Conversion combo box, selects the point type from the Linear, Interpolation, Equation, or ScratchPad types. After selecting the point type, the appropriate controls are displayed for that type of point. The current view shows the controls for a linear point. Each point also contains a unit field to define the engineering units.

Figure A.1: Point Editor.

As discussed earlier, a linear point requires three pieces of information to perform its conversion. These are the input point, the scaling constant m, and the offset constant b. This is shown in Figure A.2.

Figure A.2: Linear Point Editor.

An interpolation point requires the input point as well as the name of a data table that stores the actual points for the interpolation calculation. This is shown in Figure A.3.

Figure A.3: Interpolation Point Editor.

The edit button, shown in Figure A.3, is used to open the interpolation table editor shown in Figure A.4. If the table does not already exist, it is automatically created. The number of items in the table is not limited. As a convenience, the value of the input point can be copied directly into the Raw value text box with the use of the button labelled "<<". The table in Figure A.4 contains two items.

Figure A.4: Interpolation Table Editor.

The equation point type requires a valid algebraic expression to be entered. The syntax of the equation must be formatted to the specifications defined in the muParser manual. Any other point that is defined can be used in the equation. This is seen in Figure A.5.

Figure A.5: Equation Point Editor.

A.6 Low Level Points

The real time application uses the term point to refer to a single measured or calculated value. The lowest level points are normally hardware I/O points. Rather than give the ability to randomly name these low level points, a structured naming convention was designed. These low level point names are then available to be used in any other point calculations. Fixing the low level point names makes it easier to identify the exact origin of a measurement value.

The prefix for each of the low level points uses the short form name of the hardware device used to make the measurement. Table A.1 contains the short form names.

Table A.1: Hardware Short Form Names.

Device	Short Form Name
PowerDNA	PDNA
Sensoray	IOM
National Instruments	NI

The prefix is followed by a number which indicates the slot or port number that the data acquisition device is installed in. For the PDNA devices, these are slot numbers. For the Sensoray boards, these are port numbers. There is no number associated with the National Instruments card since there is only one card being used.

The device number is followed by a single letter short form indicating the type of measurement that was performed. Table A.2 lists the short forms.

Table A.2: Measurement Type Short Form Names.

Measurement Type	Short Form
Analog	A
Digital	D
Frequency	F

The measurement type is followed by the directional short form for the measurement. These are listed in Table A.3.

Table A.3: I/O Type Short Form Names.

Direction	Short Form
Input	I
Output	O

Finally a channel number is assigned to the point, based on which physical channel is being referred to. The channel numbers start at zero and increment by one to the highest available channel.

An example of a hardware point name would be PDNA1AI06. From the point name it is easy to identify that this is an analog input, connected to the PowerDNA cube slot 1, channel 6. There is no flexibility in this naming convention. This is simply a case of structure overriding freedom.

APPENDIX B: ALARMS

The objective of engine testing is to perform a requested test sequence while acquiring data that leads to a result. The engines being tested are often prototypes that are either very expensive or are assembled with preproduction parts. The test procedure will contain a number of alarm conditions that should be monitored during the execution of the test. The alarm conditions will generally be given as a limit on a parameter such as engine coolant temperature or oil temperature. If one of the alarm limits were exceeded, it would expose the test engine to an unsafe condition.

The ability to create complex alarm conditions has been added to the engine test control system. The design of the system is such that any alarm created will return a true or false condition. If a true condition is returned as the result of the alarm computation, the parameter monitored would have exceeded its acceptable limits.

Three categories of Alarms have been created. The Alarm Only category is designed for minor fault conditions that should be alerted to the engine technicians. A more serious fault would fall into the Coast Shutdown category. This is an indication that a fault has occurred, but there is no suspected failure of the engine. A possible action would be to bring the engine to an idle condition. The highest priority alarm is the Emergency Shutdown category. A fault in this category would mean a serious condition exists that requires immediate attention. The engine would be brought to a complete stop with all electrical power and fuel source removed.

An alarm, as defined, is basically a comparison between a measured value and a constant. The constant being the threshold, that if exceeded, would cause a fault condition. This turns out to be a special case type of equation point. One of the more

powerful features of the muParser mathematical parser is the built in ability to perform an "if" statement. The actual implementation of the "if" statement within muParser is an "if … then …else… ". The format of the muParser "if" statement is shown below. When the logical expression is evaluated, *value* will be assigned *trueResult* if the expression is true, otherwise *falseResult* will be assigned to *value*.

$$value = if(\ logical\ expression, \quad trueResult, \quad falseResult)$$

B.1 Alarm Example

As an example, a test requester may have a condition defined that restricts engine coolant temperature from exceeding 225 degrees Fahrenheit. This would be implemented in muParser as shown below. An alarm calculation, as previously defined, should return a 1 or a 0. These are the two result values used in the "if" statement. The variables CoolantOutTemp and a_CoolantOutTempHigh are points in the real time database. The point CoolantOutTemp would be created in the point editor. This is the measured value of coolant temperature. The point a_CoolantOutTempHigh is defined from within the alarm editor. This represents the result of the comparison between coolant temperature and the temperature limit. It will contain a value of 1 if the coolant temperature is greater than 225 degrees Fahrenheit or 0 if it is below. A standard naming convention has been implemented where all alarm points have a prefix of "a_". This allows them to be easily differentiated from other points.

$$a\_CoolantOutTempHigh = if(\ CoolantOutTemp > 225.0, \quad 1, \quad 0)$$

B.2 Alarm Editor

The alarm editor shown in Figure B.1 is the user interface designed to allow easy data entry of the alarms into the SQLite database. The current displayed alarm, in Figure

B.1, is for the engine coolant out temperature as discussed above. Creating a new alarm requires a few pieces of information. The "Name" field is the name of the alarm. This name is used to create a new point in the real time database that gets assigned the output value of the logical comparison. "Monitor" is the name of a point whose current value is stored when the alarm is triggered. As discussed above, the "Formula" field contains the equation for the alarm. An additional "Message" field contains a text message that explains the reason for the fault. Finally, the "Type" field sets the category of the alarm to Alarm Only, Coast Shutdown, or Emergency Shutdown. An alarm can be set inactive by simply setting both the true and false conditions to 0. Any alarm that has been disabled is displayed with a red foreground color.

Figure B.1: Alarm Editor.

B.3 Alarm Monitor

The alarm monitor shown in Figure B.2 is a dialog window that allows the user to identify active alarms. The window displays the list of active alarms that were received from the real time control application. This transfer of alarms from the real time control

application to the Windows computer is done over Ethernet and is covered in Appendix F which discusses the communication protocol.

The list of active alarms is internally maintained in the Windows application. The time the alarm was first active, as well as the last time the alarm was active, are also displayed. This is useful for glitch alarms, as well as to identify the order in which alarms were triggered. Each alarm listed contains the name of the triggered alarm, as well as the value of the point requested to be monitored. If an item is selected from the list, a user friendly message is displayed. The equation text is also displayed as a reference.

Figure B.2: Alarm Viewer.

B.4 Implementation

Internally, every alarm point is associated with a point of the type CFormula. Below we can see that the class, CAlarms, has a vector of pointers of type CFormula, named alarms. Each of these items represents one alarm point.

```cpp
class CAlarms{
public:
        CAlarms();
        virtual ~CAlarms();
        virtual void Init();
        virtual void UpdatePoints( DATAPOINTS &db);
        virtual std::string CheckAlarms(DATAPOINTS &db);
```

98

```
private:
        std::vector<CFormula*> alarms;
};
```

The Init method is used to retrieve each of the defined alarms from the SQLite database. The full details of this are not discussed here. More information can be found in Appendix G which discusses SQLite. The UpdatePoints method traverses the alarms vector and calls the Convert method on each of the CFormula objects. This executes the logical expression that was defined for each alarm. Lastly the CheckAlarms method, shown below, is responsible for inspecting each of the alarm points for a fault condition. The returned value from this method is a string containing the list of active alarms and the values of the monitored point.

```
std::string CAlarms::CheckAlarms( DATAPOINTS &db){
        std::string names;
        std::vector<CFormula*>::iterator it;
        double currentValue=0.0;
        db["AlarmActive"]=0.0;
        db["ShutdownActive"]=0.0;
        db["EmergencyActive"]=0.0;

        for (it=alarms.begin(); it < alarms.end(); it++)
        {
                char buffer[25]={0,};
                int size=0;
                if ( db[(*it)->Name] == 1)
                {
                        names+= (*it)->Name;
                        names+= "=";

                        if ((*it)->AlarmStored==0)
                        {
                                (*it)->AlarmStored=1;
                                currentValue = db[ (*it)->MonitorPoint ];
                                (*it)->MonitorValue = currentValue;
                        }
                        sprintf(buffer,"%f", (*it)->MonitorValue);
                        buffer[size-1]=0;
                        names+= buffer;
                        names+=",";
                        if ((*it)->Type ==0) db["AlarmActive"]=1.0;
                        if ((*it)->Type ==1) db["ShutdownActive"]=1.0;
                        if ((*it)->Type ==2) db["EmergencyActive"]=1.0;
                }
                else
                {
```

```
                        (*it)->AlarmStored=0;
                }
        }

        return names;
        }
```

The CheckAlarms method is also responsible for assigning the value of each alarm category point. If an Alarm Only fault is found, the point AlarmActive will have a value of 1 otherwise it will be 0. The Coast Shutdown alarm condition is stored in the point ShutdownActive.   Finally if any alarm configured in the Emergency Shutdown category is triggered, the point EmergencyActive will be 1.

APPENDIX C: PARAMETER MONITORING

While a test is running, it is important to ensure that the engine being tested is operating within an envelope of acceptable performance. Most engine test systems provide some form of alarm condition monitoring that compares a measured variable to a single defined value. This is usually accomplished in the form of a less than or greater than comparison. For example, there may be an alarm condition set on engine coolant temperature exceeding a preset temperature value. These types of fault monitoring are necessary and quite easy to configure. This method of fault detection would be global in nature and independent of the engine's current operating set points.

Some fault conditions are not easily detected by these global single condition comparisons. An engine may experience a drop in torque due to any number of conditions. The deviation in torque value could be less than ten percent and would more than likely go unnoticed, even if a trained technician was monitoring a test. Durability tests are run twenty four hours a day and are typically not monitored continuously by an engine technician.

An engine durability test is normally preceded by a break in test and a power test. The break in test is typically not rigorous. As the name indicates, it is designed to slowly break in mechanical components of the engine. The power test is an engine performance test to validate the engine's output power against a specification. The power test will ramp the engine speed, in increments of hundreds to thousands of rpm, while holding the throttle position at WOT. There are tolerances on each of the specifications and each engine will have slightly different characteristics. One of these specifications is the

torque versus speed. An example of a torque versus speed curve from a power test is shown below in Figure C.1.

Torque vs Engine Speed

Figure C.1: Torque vs. Speed.

This curve shows that the output torque of an engine is a function of speed with the throttle at WOT. After the break in and power tests have completed, the actual durability test will begin. It is desirable to ensure that the torque measurements from the power test are maintained during the durability test.

C.1 Methodology

Accomplishing this task with single point monitoring of torque is not possible. A simple modification to the strategy will be effective. The statement of comparison "torque versus speed" is actually the solution to the problem. By creating a lookup or interpolation table, that has speed as the index and torque as the output, the expected torque at any speed can be obtained [36]. This torque value can be compared to the

current measured torque during the durability test, resulting in a deviation value. This deviation would then be compared to the torque specification deviation allowance.

The interpolation point type has already been implemented. This provides the required lookup table functionality. The points for the lookup table are simply copied from the power test torque versus speed curve, as shown below in Table C.1

Table C.1: Torque vs. Speed.

Engine Speed (rpm)	Torque (lb ft)
1000	308
1500	342
2000	359
2500	373
3000	377
3500	404
4000	413
4250	417
4500	415
5000	393
5500	378
6000	321

C.2 Implementation

Incorporating online parameter monitoring within the test system requires a number of small code blocks to be executed at the proper time. The actual implementation of torque monitoring will be discussed. The same methodology could be used to monitor any parameter online.

A decision is required of when parameter monitoring should be enabled during a test. In the case of torque monitoring, the actual torque table is only valid when the throttle is wide open. This was the position of the throttle when torque was measured in

the power test. If the throttle opening is fifty percent, it would be unlikely that the engine would produced the same torque as it did at WOT. Therefore during the design of the durability test, each step running with the throttle wide open should monitor for torque deviations.

The first step is to create the points that are required for monitoring the expected torque output. There are two levels of deviation being monitored. One level is simply a warning, and the second is a shutdown. This requires a total of four points. The expected torque trending point is named ParamTorqueCurve and is defined as an interpolated point. The setup of this point is shown below in Figures C.2 and C.3.

Figure C.2: Parameter Monitoring Point.

Figure C.3: Parameter Monitoring Table.

Another point, named Param_Torque_Limit, is created as a scratch pad point that stores the result of the torque monitoring calculations. The other two points, TorqueWarn and TorqueShutdown, are equation points that are configured as constants and hold the amount of deviation allowed. These values are typically 0.95 and 0.90, respectively. This corresponds to an allowed deviation of 5 percent for a warning or 10 percent for a shutdown.

These points are used in an script function named ParMon. This function is normally defined within the global functions of a test, and can be called at any step of a test. The ParMon function contains the required statements for monitoring each parameter. The torque monitoring example is shown below, along with the associated function MonitorLowLow. There are also many functions, like MonitorHigh and MonitorLow, which monitor only a single level of deviation. Other functions could be implemented directly within the script that would allow any form of monitoring required for a specific application.

```
void Parmon(){

        Param_Torque_Limit = MonitorLowLow(Torque,
                                           ParamTorqueCurve * TorqueWarn,
                                           ParamTorqueCurve             *
TorqueShutDown);
        //Insert additional evaluations here
}


double MonitorLowLow(double paramValue, double Warning, double Shutdown){

                if(paramValue < Shutdown)
                {
                        return 2;
                }

                if(paramValue < Warning)
                {
                        return 1;
                }

                return 0;    //** OKAY
}
```

When the ParMon function call is complete, the point Param_Torque_Limit will

hold a value of 0 if the torque is within the specification. It will hold a value of 2 if the

deviation is more than 10 percent or a value of 1 if the deviation is more than 5 percent.

This can be used within the alarm configuration to setup the associated warning and

shutdown alarms.

An additional function was created to help with the alarm configuration, and to

ensure all conditions are met for parameter monitoring to be enabled.   The

SetupParamMon function shown below, checks that the throttle is wide open for ten

seconds and the engine has ramped to its speed setpoint. The end result of the function is

to set the point ParamMonitorEnable to 1 or 0 to enable or disable parameter monitoring,

respectively. A more complex function is actually implemented that configures many

different forms of parameter monitoring. The SetupParamMon function is called outside

of the test, since it is common for all tests. It is called from the Always function within

the control.as script file.

```
void SetupParmMon(){
            //Check for WOT Stabilized
            if( Throttle < 100 || DynoSpeed != DynoSpeedSp) ThrottleWOTcnt
=0;

            if (Throttle == 100.0 ) ThrottleWOTcnt += 1;

            if (ThrottleWOTcnt > 100) WOTStable = 1.0;   //At WOT for 10
seconds
            else{
                  WOTStable = 0.0;
            }

            if (ParamMonitorDisable == 0 && IgnitionPwr == 1 && RUN ==1 &&
            HOLD == 0  && WOTStable ==1  &&  DynoSpeed == DynoSpeedSp){
                  ParamMonitorEnable =1;
            }
            else{
                  ParamMonitorEnable =0;
            }
}
```

APPENDIX D: SCRIPTING

For this project, the decision was made to use an open source scripting language embedded in the core of the control system software. This decision was based on the successful use of scripts in the gaming industry which must meet many of the same performance characteristics needed for engine testing. Some of the key things considered in choosing a scripting language were the ease of integration, syntax similar to the C and C++ languages, cross platform capability, and proven use in other applications. The language should be similar to C and C++, since this would allow any of the scripts to be compiled into the core, if the performance was poor. The possibility of dynamically compiling the code into a shared library or dll and loading it at runtime was also an option that was explored.

The scripting language that was chosen for this project was AngelScript. AngelScript has a syntax that is based on C and C++. Unlike most scripting languages, it has strict variable typing and does not include support for a variant data type. It is cross platform compatible with many different computer architectures and compilers. Most of the applications to date have been in commercial and indie games. The zlib license used for the AngelScript library is very liberal, allowing use in commercial applications. The only request is that you give recognition to the author, Andreas Jönsson [17].

D.1 Script Engine Setup

AngelScript was embedded and extended within the core of the engine testing application. All of the user configurable logic is executed using the scripting language. The core of the application performs all of the input and output, point manipulation, alarm monitoring, communication and data logging. As shown in Figure D.1 below,

there are three different scripts executed under the control of the core application: control.as, user.as and script.as. All of the scripts have an extension of .as which is the default for AngelScript files.

Figure D.1: Script Files.

AngelScript was extended to expose functions from the core application that the user could call from a script. Adding a function that can be called from within a script is done by registering the function with the scripting engine. For example, a function that returns the current date was registered with the script engine using the code below. The full details of the proper syntax can be found in the AngelScript manual [17].

```
engine->RegisterGlobalFunction("string@ Date()",
                                asFUNCTION(GetDate),
                                asCALL_CDECL);
```

The Date function takes no parameters and returns a reference to a string. This function could be used in a script as shown below.

```
string currDate = Date();
```

An example of some of the functions that were added and a description of the action they perform is shown in Table D.1.

Table D.1: Custom Script Functions.

Function	Description
SetSpeed(double rpm, double rampRate)	Change the engine speed setpoint and ramp rate
SetLoad(double load, double rampRate, double percent, double mode)	Change the engine torque setpoint and control mode to throttle position or torque control
ASAP3UpdateParam(string &in, double value)	Update a calibration parameter in the engine controller using ASAP3
Print(string &in)	Send a message to the test manager for display
Error(string &in)	Send a message to the error message window
InitDataLog(string &in, string &in)	Create a new datalog on the real time controller
WinDataLog(string &in, string &in)	Write to a Windows datalog
string@ Date()	Return the current date
string@ Time()	Return the current time

In addition to the functions added, every point in the real time database is directly accessible to the script. Each of the points in the real time database is registered as a global point in each of the script engines. This is done at runtime and does not show up in the actual script file. The actual point registration is done using the code below. The list of points in the real time database is traversed and registered in the script engine, one at a time with a double data type. This means that any point in the real time database can be referred to using the exact name defined in the point editor. When registering each point, it should be noted that a pointer is actually registered. This is the reason why the values of each point are separated from the calculation objects in the real time database.

If a variable from the calculation object was registered and subsequently deleted from

memory because of a dynamic update, the script would hold a dangling pointer.

```
void CRTScript::AddVariables(DATAPOINTS &db)
{
            std::string myPoint;
            DATAPOINTS::iterator it;

            for (it=db.begin(); it != db.end(); it++)
            {
                    myPoint.erase();
                    myPoint = "double  ";
                    myPoint += it->first;
                    engine->RegisterGlobalProperty(myPoint.c_str(),
                                                          &db[ it->first]) ;
            }
}
```

D.2 Script Files

The control.as script file is responsible for all low level control functionality, such

as PID loops, timers, engine starter control, test cell functions and condition monitoring.

Changes to this file do not take effect unless the application is shutdown and restarted.

Due to the critical nature of the functions controlled by this script, it was decided not to

allow it to be reloaded at runtime. There are two entry points that the core application

calls continuously in this script. The function Always is called at a rate of 10Hz and the

function HighSpeed is called at a rate of 100Hz. The user can add any required code to

these functions that needs to be executed at these rates.

From Appendix B on Alarms, it was stated that the user could configure the

actions that need to be performed when an alarm is triggered. These actions are stored in

the control.as script file as well, and are called only when an alarm is triggered. These

functions are listed in Table D.2

111

Table D.2: Alarm Function Definitions.

Alarm Type	Trigger Point	Function Called
Alarm Only	AlarmActive	AlarmGeneral
Coast Shutdown	ShutdownActive	AlarmCoast
Emergency Shutdown	EmergencyActive	AlarmShutdown

The Always function contains the base control strategy for the test cell. A reduced example of this function is shown below. The function starts with a check to see if the script is running for the first time, by inspecting the value of initScript. If this is the first run it performs the Initialize function, otherwise it does nothing. Subsequently, a number of functions are called which perform monitoring and control. Finally, the alarm functions are called only if an alarm has been triggered.

```
void Always()
{
        if (initScript < 1){
                initScript = 1;
                Initialize();
        }

        ExhaustCoolingWater();
        UpdateTimers();
        SetupParmMon();
        StackLights();
        TempCtrl1PID();
        TempCtrl2PID();
        TempCtrl3PID();

        if(      EmergencyActive == 1)  { AlarmEmergency();}
        else if( ShutdownActive  == 1)  { AlarmCoast();}
        else if( AlarmActive     == 1 ) { AlarmGeneral();}
}
```

The user.as file is a general purpose script. It was created to allow users to add any additional functionality required to implement a test procedure. The script is completely dynamic and can be reloaded on the fly at runtime. This is useful for testing

algorithms since the core application does not need to be restarted. The entry point of this script is a function, named User, which is called at a rate of 10Hz by the core application.

The actual test procedure to be executed is stored in the script.as file. Since a test procedure may be modified and need to be reloaded, this script file can be dynamically loaded at runtime. The script.as file contains one function for each step of a test sequence defined by the user. The naming convention developed for the functions is to use the step number prefixed with the word Step. The function name for step number 1 would be Step1. This file also contains any global functions and variables defined in the test procedure. This is described in detail in the code generation and test builder section found in Appendix E.

D.3 Calling Script Functions

Earlier it was stated that functions such as Always, HighSpeed, and User in the script files were called from the core application at a specified frequency. In order to call these functions, they must exist in the script file. To call a function that is defined in the script, the address of the function must first be found using its signature. As part of the initialization of the script engine for each script file, there is a script validation routine that was created to check for the existence of the critical functions. If these functions do not exist, a fault is triggered and the application is shutdown. The names of each of the required functions are loaded into a vector, named funcNames. While traversing this vector in a loop, each function name is requested from the script engine using the method GetFunctionIDByDecl. If the function signature exists in the script, a number greater

than or equal to 0 is returned. The code to check for the existence of functions in the script is shown below.

```cpp
int CRTScript::CheckFunctions()
{
        std::vector<std::string> funcNames;
        std::vector<std::string>::iterator Iter;
        std::string Name;

        funcNames.push_back("void Always()");
        funcNames.push_back("void AlarmGeneral()");
        funcNames.push_back("void AlarmCoast()");
        funcNames.push_back("void AlarmEmergency()");
        funcNames.push_back("void HighSpeed()");


        for( Iter = funcNames.begin(); Iter != funcNames.end(); Iter++)
        {
                Name= *Iter;
                int    funcId    =    engine->GetModule(0)->GetFunctionIdByDecl(
                Name.c_str());

                if( funcId < 0)
                {
                        std::cout  <<  "Function "  <<  *Iter    <<  "  must  be
                        included in the control.as file." << std::endl;
                        ctx->Release();
                        ctx = 0;
                        engine->Release();
                        engine = 0;
                        return -1;
                }
        }

        return 1;
}
```

All of the functions in each script are also stored in a std::map, named functions, during the script initialization phase. The code below shows how to populate the map.

```cpp
std::map<std::string, int> functions;
funcCount = engine->GetModule(0)->GetFunctionCount();
for(int theCount =0 ; theCount < funcCount; theCount++)
{
        int            theFuncId            =            engine->GetModule(0)-
>GetFunctionIdByIndex(theCount);
        functions[engine->GetModule(0)->
                        GetFunctionDescriptorById(theFuncId)->
                        GetName()] =  theFuncId;
}
```

A couple of overloaded Run methods have been created that allow the core application to call a function defined in a script. They are shown below, along with the supporting RunFunction method. One method calls Run with a string parameter containing the name of the function that is defined in the script. Another calls Run with an integer step number which is used to create the string name of the function to execute in the script. This is used later during the execution of a test sequence.

```
void CRTScript::Run(int Number)
{
        char stepNum[128]={0,};
        sprintf(stepNum,"Step%d",Number);
        int functionID = mySteps[stepNum];
        RunFunction(functionID);
}

void CRTScript::Run(char* Name)
{
        int functionID = mySteps[Name];
        RunFunction(functionID);

}

void CRTScript::RunFunction(int fID)
{
        r = ctx->Prepare(fID);
        r = ctx->Execute();
}
```

It should be noted that all of the error checking code has been removed for brevity. The RunFunction shows an object, named ctx, being used to ultimately call the requested function. The context stores the current state of all the variables and objects, as well as preparing the stack for parameters passed or returned to a script function.

There have been some details left out about how the script engine itself is configured and used. These details are wrapped into the CRTScript class that was created. For a full explanation of what is required the reader should refer to the AngelScript manual [17].

115

APPENDIX E: CODE GENERATION AND TEST BUILDER

Appendix D on scripting described what a script was and how they are used in the real time application. It is possible for a user to code a script.as file from scratch that implements a test procedure. This task would be easier if there was an editor that provided a basic template and syntax highlighting, for test entry. This is the purpose of the test manager. The test manager is a GUI application that provides an easy method to enter a test and create the required script.as file. The tests are stored in a SQLite database and can be easily copied or imported from another database. Figure E.1 shows the main dialog for the test manager. This dialog is used to enter the basic identifying information for the test and the engine being tested.

Figure E.1: Test Manager.

A complete test can be created in the test editor without entering a single line of script code. Much more complex tests can be created if the user adds custom script code.

Applications that perform the function of taking user input and outputting source code are called code generators. Code generators are very popular for writing database applications. This is where the basic idea for test manager was derived. They perform the function of relieving the user from entering repetitive code. This minimizes the chance of introducing errors.

E.1 Sequence Editor

In order to manage the database records, a few buttons such as New, Delete, Copy, etc were created (see Figure E.1). The button labelled sequence opens the sequence editor shown in Figure E.2 below. This is where each of the steps for the test is created.

Figure E.2: Test Sequence Editor.

One of the difficult problems to solve with the sequence editor was determining how to allow the user to create a jump, without needing to manage the step numbers. The difficultly arises with the insertion and deletion of steps. The GUI allows the insertion or deletion of steps without restriction. If only the raw step numbers were used, it would be left to the user to ensure that jumps were maintained when inserting and deleting steps. Internally, the real time application expects the step to be an integer number. The solution was to create global variables in the script file for each step that contained a label. This allows the use of the label name, instead of the step number to perform a jump. Looking at the solution today it seems obvious, but at that time a number of days were spent resolving this single item.

One of the problems when using other engine test software is the inability to execute code as part of a step. Most of the test builders from other test software, refer to an "if" instruction as a step. The basic idea being that each step is a logical, set point or flow control step. Having full control of the design of this software, it was decided to make a step nothing more than a number. Essentially, a test could be created that did nothing but go from step to step passing time. This would never be the case, but it illustrates the fact that a step has no meaning unless the user defines actions.

Using this approach provides the capability to add complex logical condition checking and flow control. Looking at this from an eagle eye view, what was created is nothing more than a simple state machine. By default, the state machine changes steps only when the current step time has expired. If the user adds script to the step, they have the ability to perform any logical check and can change steps at will. There is no predefined notion of what can or cannot be performed during a step.

There were still some common items that would be performed in most engine test sequences. For example, speed and torque set points would be used in all tests. These items are seen in the sequence builder displayed in Figure E.2. The fields exist, but are completely optional. If no data is entered into the fields, the code generator does not output any code. If data is entered into the engine speed field, the code generator will output proper instructions to perform a speed set point modification. The only required field is the time field which is the basis for the state machine.

Adding these commonly used fields simplifies the creation of tests. The fields were designed not to be type checked. This allows for more complexity. Logically, the engine speed should be a numeric value and this should be enforced in the GUI. However, doing so puts a restriction on the user to only enter numeric values. The text entered is used by the code generator. Therefore, the opportunity exists to write code in the RPM field as well. This is a very powerful tool that can be used in many ways. One simple idea is using an array, indexed by a counter, instead of the numeric value. This allows a common test to be designed that takes an array of rpm setpoints as the input. The test could then be easily copied and the array modified for a different set of test requirements.

E.2 Global Points

To create an array, the global portion of the test editor is used. The global variables editor is shown in Figure E.3 below. This dialog has only two text boxes. The user is free to enter any valid code in the text boxes. In order to separate the global variables from the global functions, two text boxes were created. In the dialog box, two

arrays are declared with 13 values in each. The RPMArray contains the set points for engine speed. The TorqueArray contains the set points for the load.


```
Globals (PV11_35_1S364_25064_PWR)

Global Variables | Global Functions

000 //
001 //
002 int        PTHRFL        =    0;        //
003
004 //Define RPM Step Array
005 //
006 int RPMPoints;
007
008 double[] RPMArray = {
009   1000.0,
010   1250.0,
011   1500.0,
012   2000.0,
013   2250.0,
014   2500.0,
015   3000.0,
016   3500.0,
017   3900.0,
018   4500.0,
019   5000.0,
020   5500.0,
021   5750.0,
022 };
023   double[] TorqueArray = {
024   348.44,
025   432.5,
026   488,
```

Figure E.3: Script Global Variables.

Since AngelScript has support for the basic set of data types from the C and C++ languages, any of these variable types can be used. This includes support for a string data type. Without the ability to create strings in the script, it would be difficult to argue that this project was a complete solution. It should be noted that the variables declared and used in any part of a test are not known to the real time database. The fact that the script has intimate knowledge of the real time database means that the point names cannot be

used as variable names in the script. There are methods to allow this inspection and modification of script variables, but this has not yet been implemented.

E.3 Global Functions

The global functions text box shown in Figure E.4, allows the entry of any valid script. However, it was intended to create functions to assist the test. No function defined will be called unless the user specifically makes a call from within a step. This is the portion of the test manager where parameter monitoring was designed and implemented. A number of functions were written to perform checks on parameters. The user is responsible for calling these functions from the steps, if parameter monitoring is desired.


```
Globals (IP11_37_1S374_15089_EFT)

Global Variables   Global Functions

000 /////////////////////////////////////////////////////////////////
001 //This function writes the control header to a GDR Data File
002 //Author:    Tony Fountaine
003 //Date:      Aug 25, 2011
004 //Rev:       1.0
005 /////////////////////////////////////////////////////////////////
006
007 void WriteGDRControlHeader()
008 {   //string outFile = testFile;                     |
009
010     string strHeader =   "^CONTROL_BLOCK\r\n"
011                          "GDR_VERSION\t2.0\r\n"
012                          "FILE_CREATE_DATE\t"    + Date()    + "\r\n"
013                          "FILE_CREATE_TIME\t"    + Time() + "\r\n"
014                          "^HEADER_BLOCK\r\n"
015                          "DISCIPLINE\t"           + gdrDiscipline    + "\r\n"
016                          "DISPLACEMENT\t"         + "3700\t" + "cm3\r\n"
017                          "data_source\t"          + gdrDataSource    + "\r\n"
018                          "dos_filename\t"         + gdrDosFilename   + "\r\n"
019                          "TEST_ORDER\t"           + gdrTestOrder     + "\r\n"
020                          "TEST_DATE\t"            + gdrTestDate      + "\r\n"
021                          "TEST_NAME\t"            + gdrTestName      + "\r\n"
022                          "ENGINE_NO\t"            + gdrEngineNo      + "\r\n"
023                          "TEST_TYPE\t"            + gdrTestType      + "\r\n"
024                          "ITEM_NUMBER\t"          + gdrItemNumber    + "\r\n"
025                          "TEST_ROOM\t"            + gdrTestRoom      + "\r\n"
026                          "REQUEST\t"              + gdrRequest       + "\r\n"
027                          "RUN\t"                  + gdrRunNumber     + "\r\n"
028                          "CALIBRATION\t"          + gdrCalibration   + "\r\n"
029                          "HDR_COMMENT1\t"         + gdrHdrComment1   + "\r\n"
```

Figure E.4: Script Global Functions.

121

E.4 Generating the Script

The basic concept of what the test manager's function in creating a test has been described. The last item to be discussed is how the script.as file is actually created. Naturally, one would think that there was complex and confusing code responsible for creating the script.as file. The truth is, the code generator does nothing more than concatenate strings and write them to the file script.as.

First, the test header information is read from the SQLite database and written at the top of the script.as file. This provides descriptive information that can be used to identify which test was used when the script.as file was created. This would look like the code below where we see valid comment sections used for the information.

```
//Script for Test:ZA3726_SS_Aging_Cycle
//Automatically Generated Date:31/05/2011 2:55:16 PM

/* Description:
 *
 */

/* Comment:
 *
 */

/* Cell Name:
 * EEP_3
 */

/* Engine Type:
 * Scorpion Diesel
 */

/* Engine Displacement:
 * 6.7
 */
```

Next, a series of variables are declared that represent any labels that were defined by the user. There is a one to one correlation between the label value and the step number. This is completely managed by the code generator. This can be seen below.

```
//Auto Generated Step Label Defines
```

122

```
//DO NOT EDIT
int Documentation         = 1;
int Test_Start            = 2;
int Emissions_Cycle_Start = 3;
int Block_Start           = 4;
int Block_Step_Start      = 5;
int Aging_Cycle_Step      = 6;
int Aging_Cycle_Loop      = 7;
```

This is followed by the user defined global variables. The code below shows an example, along with the descriptive warnings, that are generated. Only one variable is displayed for brevity.

```
//All variables declared here a global to the script.
//They cannot be accessed from outside of the script.
//Any variables that need to be global to the system need to be
//    declared in the Point Editor.
//Any function or Step can access them.
//Any variable declared locally in a step or function does not
//           maintain its value from call to call.

int PTHRFL = 0;                  // Part Throttle Flag
```

Logically, the user defined global functions are inserted next. The code below displays the Parmon function discussed in Appendix C on parameter monitoring. Again, many more functions would exist.

```
//////////////////////////////////////////////////////
//This function is used for Parmon calculations.  Put all
//functions that you want to execute for every Parmon() call below
//Author:  Chris Kelly
//Date:
//Rev:    1.0
//////////////////////////////////////////////////////
void Parmon()
{
    Param_Torque_Limit = MonitorLowLow();
    Param_Spark_Limit  = MonitorHighHigh();
    //Insert additional evaluations below
}
```

The final item in the script.as file is each of the functions created from the steps of the test sequence. All of the data and scripts that the user entered for each step are put

together in the form of a function. One function is defined for each step. The name of the function is the word Step followed by the integer step number. For example, the function for step 8 is shown in the code below.

```c
//Comment:Set Engine Rpm/Load and allow to stabilize
void Step8()    //Regen_Set_RpmLoad
{
   if (FirstScan == 1)
   {
      StepTime = 10.0;
   }
   StepLength = 10.0;
   if ((FirstScan == 1) || (CONTINUETEST == 1))
   {
      //SetSpeed( speed RPM, ramp RPM/s)
      SetSpeed(Regen_Rpm[int (A_Block_Step)], 10.0);
      //SetLoad(Torque   Nm,   Ramp   s,   Throttle   Position,   Mode   0=Load
1=Throttle)
      SetLoad(Regen_Torque[int (A_Block_Step)], 10.0, 0, 0);
      CONTINUETEST = 0;
   }
   NO_TIME = 0;
// ************************************************************
// ************************************************************
//
// This step is used prior to starting Regen to allow Rpm and Load to
stabilize.
//
   if (FirstScan == 1)
   {
      Print("Starting Regen");
   }
   NO_TIME = 1; // Set Variable to Turn Test Time OFF
}
```

The code generator has added some necessary logic. The variable FirstScan is set during a transition from one step to another. It contains a value of 1 for precisely one scan. For the remainder of the step, it has value of 0. This is used to perform any initialization that is required for the step. The variable StepTime holds the current step time in seconds. This value decrements until a value of zero is reached, which will cause a step transition. StepLength contains the total time of the step in seconds. When a test is paused, there is a chance that either the RPM or load set points will be changed by the user. During restart, the variable CONTINUETEST will have a value of 1. This

124

condition or a step transition will cause the RPM and load set points to be reset to the value configured in the test.   This is performed using the functions SetSpeed and SetLoad. These functions, as well as the Print function were discussed in Appendix D on scripting.   Finally, the variable NO_TIME allows a step to execute without the accumulated time being added to the test timer.   It appears odd to see the variable assigned a value twice in one function.   The assignment of the value 0 is done by the code generator as a precaution in the event that a user forgot to implement this. Assigning a value of 1 was done by the user.

Putting all of these pieces together would create a valid test that could be executed by the real time application.   The user can still create the script.as file manually. Conversely, the user could also extend the code generator concept to do more automatic code generation.   A Ford specific utility has been written that generates functions to automatically create a file formatted to a given specification.

APPENDIX F: ASCII COMMUNICATION PROTOCOL

When developing the control system, it was decided to separate the visual components and the control components onto two different computers. These two computers need to communicate data and commands with each other for the system to work. The most obvious choice for communication was selected, which is an Ethernet network. This section will deal primarily with the communication link between the GUI application running on Windows and the console application running on the real time operating system QNX.

F.1 Sockets

The socket API was used to implement the communication link. The socket API provides a library of functions to establish a connection, then send and receive data using the Ethernet hardware. The socket API is available on most operating systems and supported by many programming languages. When a socket is created, three pieces of information are required. These are the IP address, the transport protocol, and a port number. The two most popular transport protocols are TCP and UDP.

The transmission control protocol (TCP) is very popular and used in many applications, especially internet based. The protocol is connection based and streaming in nature. Two computers must first negotiate a connection. They are then able to stream data back and forth over the connection. The data is guaranteed to be transferred and received in the same order it is sent. This guarantee can sometimes introduce delays in data transfer. Since the data is streamed, there is no starting or ending point identified in the stream. The two applications must properly locate the beginning and ending of each data frame using a well designed protocol.

The user datagram protocol (UDP) is also very popular. It is a message based protocol that does not include a guarantee of successful packet transmission. Basically this means there is no error checking or hand shaking to ensure data was sent properly to the other computer. It is equivalent to putting a letter in the mail. A letter is first addressed to the desired recipient and then it is sent. It is up to the receiver to tell the sender that the mail was received. Sometimes, the sender is not even interested in knowing the mail arrived at its destination. UDP is typically used where smaller amounts of data are transferred rapidly between two computers. UDP is connectionless; so many different clients can use the same port to communicate.

The port number that the applications will use for communication is simply an unsigned 16 bit integer value. There are some reserved port numbers, such as 80 for HTTP, 21 for FTP and in general any number less than 1023. There is also well known port numbers above 1023 that certain protocols use. On a UNIX based system, the used port numbers are found in the /etc/services file. The freely usable port numbers range in value from 49152 through 65535.

In this thesis, the decision was made to use the UDP protocol. This protocol is a much better fit for the way the applications send commands and receive data. One of the useful things about the UDP protocol is the datagram concept. This basically means that data is received the way it was sent, similar to the letter in the mail concept. There is no need to frame the data. If you send a message "Hello World", it will be received by the other application as a single packet containing the message "Hello World". It should be noted that there is a restriction on the size of UDP packets. This varies in range from 512 bytes to 8192 bytes. Testing is the best way to find this exact size. There is a small

chance that some packets could be lost, but the chance is very low on a two computer network with a 3 foot patch cable. In addition, the real time application was designed to run independently. In the worst case scenario there will be lost data.

The IP address, transport protocol and port numbers allow each application to open a socket and start transmitting and receiving data. The difficult part is to decide the protocol for the messages being sent between the two computers. Many different protocols exist. Some of them are text based and some are binary. Binary protocols can be more efficient to parse if the data is packed into a structure that is known to both applications. The benefit of text based protocols is that they are easier to debug and can easily be used in a scripting language. With a packet sniffer application, like Wireshark, it is very easy to see the contents of each packet being sent and received. If the contents of the packet are plain text, it makes it much easier to find errors. A text based protocol was chosen for this thesis.

F.2 ASCII Protocol

A very simple, text based protocol was designed using an integer number to identify commands. The command is followed by the parameters needed to execute the instruction. Each of the items is separated by a pipe "|" character. The entire string is then appended with a carriage return and line feed. The full list of commands that the real time application will respond to is shown in Table F.1. For example, the command to update a point named Step to a value of 1 would be "3|Step|1\r\n".

Table F.1: ASCII Communication Protocol.

Command Number		Description	Parameters
1	**LISTVARS**	Get the full list of points from the real time database including the current value	None
2	**SETTRANSVARS**	Not implemented	None
3	**UPDATEVAR**	Sets the current value of a point	Point Name, value
4	**ALARMTRIG**	Gets list of triggered alarms	None
5	**UPDATEDB**	Copy the latest database	None
6	**RELOADSCRIPT**	Reload the test script.as file	None
7	**CHANGERPM**	Change the current RPM set point	RPM, ramp rate
8	**CHANGELOAD**	Change the current load set point	Value, ramp rate, Mode
9	**GETTEST**	Get the current test name	None
10	**GETLOG**	Get the current data log name	None
11	**SETLOG**	Create a new data log	None
12	**RELOADUSER**	Reload the user.as file	None
13	**GETDATALOGS**	Get a list of data log files	None
14	**GETDATALOG**	Transfer a single data log to Windows	None

The real time application has a thread that sits idle waiting for a new command. When a command is received, it is executed and a response is sent back to the sender if required. Since this is based on UDP, the commands can come from a number of different sources. This design is sometimes referred to as a master and slave configuration. The real time application would be the slave in this case since it does nothing unless commanded.

A small portion of the real time application code that implements the protocol is shown below. Basically, a packet is received and broken into individual commands. Each command is then broken into its individual parameters and processed. The command number is selected from a list of valid commands using a switch statement. The update variable and update database commands are the only ones shown.

```cpp
recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
                    (struct sockaddr *) &echoClntAddr, &cliAddrLen);
StringTokenizer strtok = StringTokenizer(tempStr,"\r\n");
int cnt = strtok.countTokens();

for(int i = 0; i < cnt; i++)
{
        int cmdRequest;
        std::string tempStr ="";
        std::string retMess="";
        tempStr=strtok.nextToken() ;
        StringTokenizer strmes = StringTokenizer(tempStr,"|");
        cmdRequest = strmes.nextIntToken();

        switch (cmdRequest)
        {
            case UPDATEVAR:

                    char buffer[25]={0,};
                    int size=0;
                    DATAPOINTS::iterator it;
                    std::string varName = strmes.nextToken();
                    double varValue = strmes.nextFloatToken();
                    it = dataPoints.find(varName);
                    if ( it != dataPoints.end())
                    {
                            this->lock();
                                dataPoints[varName]=varValue;
                                dataPoints["xVars"] += 1.0;
                            this->unlock();
                            size = sprintf(buffer,"%f",varValue);
                            retMess = "Update Variable: " + varName;
                            retMess += " = ";
                            retMess += buffer;
                    }
                    else
                    {
                            retMess  =  "Update  Variable  not  found  :  "  +
                            varName;
                    }
                    break;

            case UPDATEDB:
                    time_t rawtime;
                    time( &rawtime);
```

130

```
                    system("cp ./katerina.db3 ./katerina.db3.old");
                    system("cp /root/test/katerina.db3 ./katerina.db3");

                    retMess = "Database update completed.";
                    std::cout << ctime(&rawtime)  << retMess << std::endl;
                    break;
        }
```

This is a very simple design that can be compared to event based GUI applications. The GUI application sits idle waiting for an event, such as a button click. When the button is clicked, the application receives the event and performs the button click action. In the case of this engine control system, the button click from the GUI is received and simply retransmitted through a socket to the real time application. The real time application then performs the requested action.

The design also required the ability for the real time application to transmit data asynchronously to the GUI application. This is required for the implementation of alarm notification and data logging. The design and processing of the protocol is very similar. The largest difference being that the commands are actually string based, rather than integer numbers.

APPENDIX G: SQLITE

Before explaining the details of how SQLite was used in this project, it should be noted that this approach is not how a database would typically be used. The concepts are the same and could be used as a starting point in understanding the basics of databases. However, the referential design of the database has been completely ignored in this application. The intended application of SQLite was as a file replacement. This is one of the recommended usages of SQLite [18].

SQLite is a tiny embeddable library that can be used in an application to provide similar features to those available in a network based database server. It supports the same basic INSERT, UPDATE, SELECT, and DELETE commands. These commands are part of a language known as SQL. SQLite has tables and each of the tables have fields. Each database is stored in a single file. The database file and the library are both cross platform compatible. SQLite is very fast and said to be the most widely deployed database in the world [14]. Best of all, it is in the public domain and free to use.

One of the most beneficial properties of SQLite, is that it is ACID compliant [18]. Without going into a lot of detail, this means that database writes within a transaction will normally not corrupt the database even in the event of a system crash. This is extremely important considering the amount of time that could be consumed putting data into the database.

The number of files required to support an application such as the engine test control software could grow into the hundreds. Maintaining and organizing this number of files can be onerous. Also, there are two computers required to share the same data. Storing all of the same information in one cross platform searchable file solves this

problem. This makes sharing data between the two computers as simple as copying one file. There are free GUI editors available for managing the SQLite databases as well.

G.1 Table Definitions

Every piece of configuration data related to the engine test control software is stored in one SQLite database file. This file includes points, alarms, log file definitions, screen layouts, tests, etc. Figure G.1 shows an example of the major table design.
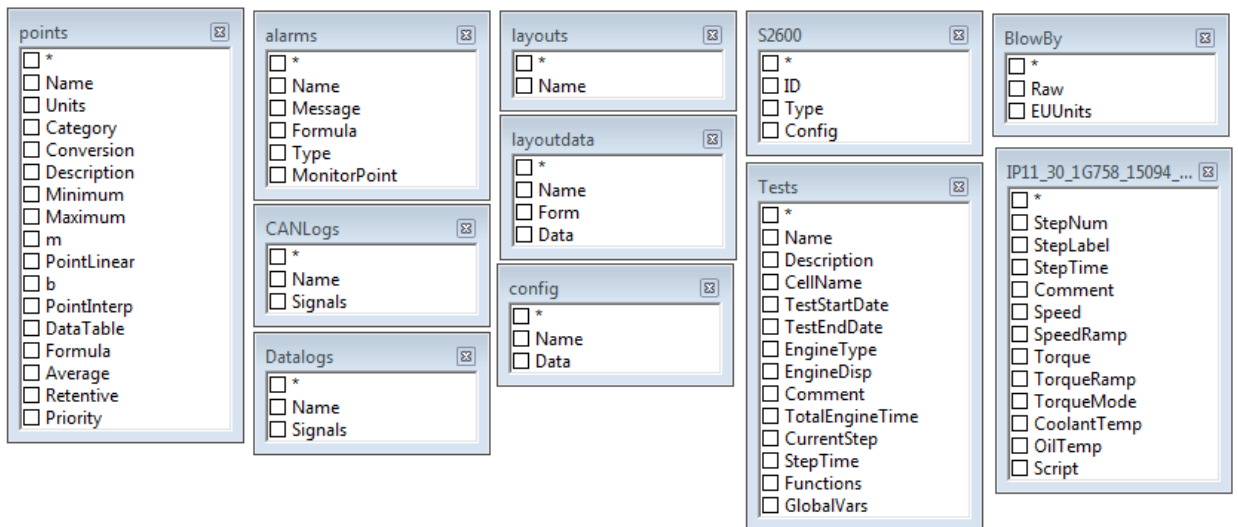

Figure G.1: Database Tables.

The points table contains a record for each point created in the point editor. The alarms table contains one record for each alarm. There is one record in the Tests table for each test created. Following this same pattern, each type of configuration data will be stored in its own table. There are also some dynamically created tables. For example, the BlowBy table is the calibration table for the interpolation point BlowBy. This table was created at runtime using the SQL statement below.

```
CREATE TABLE BlowBy (Raw REAL, EUUnits REAL  PRIMARY KEY);
```

133

When a new test is created, the test header information is stored in the Tests table. There is also a dynamically created table with the name of the test appended with the text, Seq. This table is used to store the sequence data for the test. The SQL statement used to create the sequence table for the test M1V1_FIE_3_TVSeq is shown below. This statement is executed at runtime to create the table when a new test is created.

```
CREATE TABLE 'M1V1_FIE_3_TVSeq' (StepNum INT PRIMARY KEY, StepLabel
VARCHAR(100), StepTime DOUBLE, Comment VARCHAR(255), Speed DOUBLE,
SpeedRamp DOUBLE, Torque DOUBLE, TorqueRamp DOUBLE, TorqueMode INT,
CoolantTemp DOUBLE, OilTemp DOUBLE, Script VARCHAR(1024));
```

G.2 Working with Tables

Searching for points could become tedious if the definition for each point was scattered throughout hundreds of configuration files. In SQLite, the statement to find any point in the points table is simplified. For example, finding a point that contains the text 'exh' is shown below. The percent signs are wildcards.

```
SELECT Name FROM points WHERE Name LIKE '%exh%';
```

Grep is a command line utility that can do this, but it is not as efficient when updating or deleting points. Examples of these actions in SQLite are shown below.

```
UPDATE points SET Priority = 20 WHERE Name LIKE '%exh%';
DELETE FROM points WHERE Name LIKE '%exh%';
```

G.3 Embedding SQLite

SQLite is compiled into a library that can be either statically or dynamically linked to an executable. In this application, the library was statically linked to ensure that there is no version conflict should the library be updated. The SQLite library was coded in the C language. It has with a C header file with definitions of the functions that are used to work with the database.

There are alternative wrappers that have been made to use SQLite in other languages. One of the wrappers for C++ was used in this application. An example of how data is queried using this wrapper is shown below. This is a small portion of the code used to load the point definitions from the database table points.

```cpp
void CCalc::Init(DATAPOINTS &DB)
{
        CppSQLite3DB db;
        int Conversion;
        db.open(gszFile);

        CppSQLite3Table  t  =  db.getTable("SELECT  Units,  Name,  Conversion,
        Description, Minimum, Maximum, m, PointLinear, b, PointInterp,
        DataTable, Formula, Average, Priority
        FROM points
        ORDER BY priority");

    for (int row = 0; row < t.numRows(); row++)
    {
      t.setRow(row);
      Conversion=t.getIntField(2);

      switch(Conversion)
      {
      case FORMULA:
            {
                    CFormula* myFormula;
                    myFormula = new CFormula;
                    myFormula->PointFormula = t.getStringField(FORMUL);
                    myFormula->PointName = t.getStringField(POINTNAME);
                    myFormula->min = t.getFloatField(MINFIELD);
                    myFormula->max = t.getFloatField(MAXFIELD);
                    myFormula->isAvg = t.getIntField(AVERAGE);
                    myFormula->Init();
                    points.push_back(myFormula);
                    break;
            }
      }
    }
        db.close();
}
```

The classes that are directly related to the SQLite C++ wrapper are listed in Table G.1.
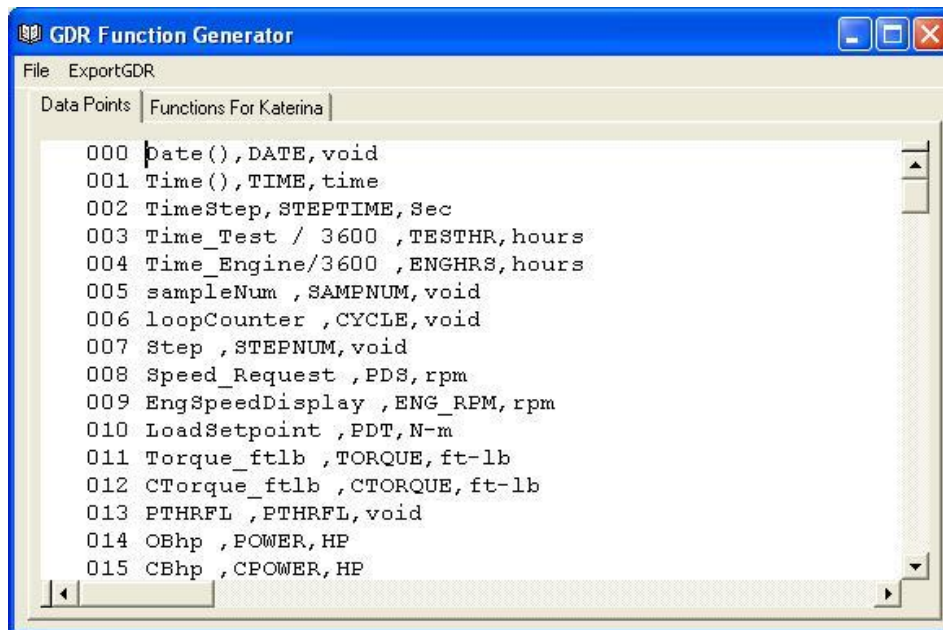
Table G.1: C++ SQLite Classes.

Class	Abstraction
CppSQLite3DB	SQLite database file
CppSQLite3Table	SQLite table

The class CppSQLite3DB provides methods to open, close, and return table data from the database. The CppSQLite3Table class provides methods to traverse records in a table and return data from the fields in standard C++ data types.

APPENDIX H: DATA LOGGING AND DATAVIEWER

One of the major purposes of testing an engine is to acquire and analyze test data. Long term data storage for engine tests are written in a Ford internal file format specification, named GDR. This is basically a modification of a delimited file format. The data for this file is typically sampled every five minutes.

In order to create this file a utility application, named GDR Function Generator, was created. This application takes a comma delimited file as input and generates a series of script functions to be used in a test procedure. Figure H.1 depicts the main screen of the application. The format for each line of input data is "Data point name, GDR point name, Units". Data point name is the local point name used at the test cell. GDR point name translates the test cell point name into a Ford global point naming convention. Finally, units are the physical units of measurement.

Figure H.1: File Format Script Generator.

The functions exported from this utility are shown in Table H.1. These functions are then inserted into the global function definitions for a test.

Table H.1: File Format Script Functions.

Function	Description
WriteDataHeader	Writes the GDR Header according to specification
WriteData	Writes a new sample of data to the file

Typically, the WriteDataHeader function is called during the initialization of a test. The WriteData function would be called at the appropriate time during each step, as shown below. Each of these functions is designed to make use of the WinDataLog function that was previously described in Appendix D on scripting.

```
// GDR Data Logging every 5 minutes and at end of step
if (TimeStep > 0)
{
    if (((intStepTime % 3000 == 0) || (intStepTime == 1)) &&
        (HOLD == 0) && (FirstScan != 1))
    {
        WriteData();
    }
}
```

The GDR files are sent by FTP to a global server within Ford, so they are accessible by other test engineers. Ford has created many macros for Excel that allows test engineers to perform standardized analysis on the data files.

In addition to long term storage requirements, it is beneficial to have data sampled at a higher frequency for diagnosing faults. Typically, this data is stored in memory using a ring buffer and dumped to a data file when a fault or alarm condition occurs. These are small files, sampled at a rate between 1 Hz and 100 Hz, and contain 10 to 30 seconds of data before and after the fault.

Rather than use this method, it was decided to try a different approach for this thesis. Since the cost of hard drive storage is about 10 cents per gigabyte, it makes sense to collect more fault data. There is no penalty for having too much data, other than the time it takes to process it. A form of a black box recorder was created. The recorder runs continuously during testing and samples data at a rate of 10Hz. All of the sampled data is stored in a SQLite database.

The points that are to be logged are added to a data log configuration. This is done using a simple dialog application, named Datalog Manager. This application is shown in Figure H.2. Multiple data log configurations can be saved and loaded, either manually or from within a test script. A specific data log configuration and file name are selected and then loaded into the real time application to start logging data.

Figure H.2: Datalog Manager.

The engine durability test cells run unattended most of the time. Therefore, the real time application will automatically create a new data log after eight hours. This helps to keep the data files within a manageable size for analysis. After eight hours of logging data, the files are normally between 200 to 500Mbytes in size, depending on the number of points logged. A durability test will run between 130 and 300 hours which would result in about 19Gbytes of data. This is equivalent to the cost of a large coffee.

The decision to use a database to store test data was met with a lot of scepticism. It was not possible to easily analyze the data without first exporting it into another format, such as a comma delimited file. A data viewer application was created that allows the database file to be viewed directly. This makes analyzing faults very easy and gives the test engineer an opportunity to see a long trace of historical data before the fault occurred. The data viewer is shown in Figure H.3.

Figure H.3: Data Viewer Application.

APPENDIX I: REMOTE MONITORING

In a medium to large engine testing facility, such as PERDC, it can be difficult to know the status of each test cell at any one point in time. A remote monitoring application was developed to assist with status checks of the cells. This would be useful to those working off site or to test engineer from another facility that may wish to view current data from a test cell.

The engine test cell control system designed for this project is installed in three of the nine test cells at PERDC. The other six test cells use an ADACS control system from Horiba. A document describing how to log data from ADACS to a MySQL database was previously written by Horiba [37]. This was used as the starting point for developing the remote monitoring web application. Since two control systems were to be remotely monitored by one application, there needed to be congruency between the data each system was populating in the MySQL database.

The Horiba document contains the basic design for how Perl was used to extract data from ADACS and insert it into a MySQL database. It also included the structure of the database tables that were used in the design. First, a MySQL database was configured with the table structure defined. The Perl script was then modified and deployed on the ADACS cells.

I.1 Configuring ADACS

The Perl script updates the MySQL database table every 10 seconds. The points that are sent to the database are stored in a table. This table is queried every 10 minutes to inspect for changes to the requested points. A very simple web administration utility was designed to allow the point definitions to be entered online as shown in Figure I.1.

There is the ability to edit cell definitions and point definitions. The table Cells, contains one entry for each engine test cell. The table Points, contains point definitions for each individual engine test cell. This is the set of points that will get inserted into the data table.
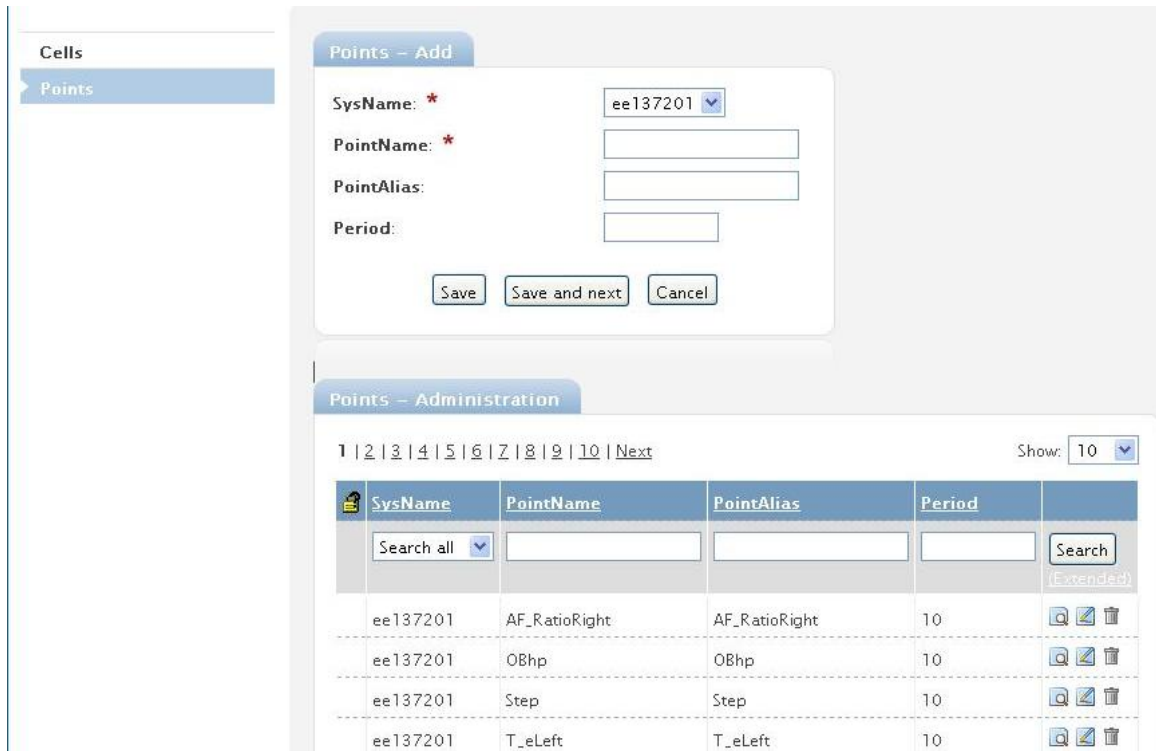

Figure I.1: Remote Monitoring Administration Site.

I.2 Merging ADACS and New System

The same concept that was laid out for the ADACS control system was also extended to this project. A simple utility was created that polls the real time database for the points requested and inserts them into the same MySQL database table. This basic utility is shown in Figure I.2.

Figure I.2: Remote Monitoring Data Collector.

I.3 RIA Application

A rich internet application (RIA) was designed to allow monitoring of the data in the MySQL database from a remote location. The application was written in Silverlight which allows it to be hosted in a web browser or installed directly on the desktop. Silverlight is a relatively new, client side, web technology designed by Microsoft and based on Dot Net. It enables creating web applications using much of the same design methods used to create desktop applications. Since it is a client side application, most of the processing is performed on the client machine, thus relieving the web server. This allows the application of rich controls and animation on the client computer and results in a much better visual experience for the end user. A competing product to Silverlight would be Adobe Flash.

The main screen of the application is shown in Figure I.3. From this screen an overview of all the test cells can be seen. The application has an internal timer that queries new data once every 10 seconds. Silverlight does not support the ability to query a database directly. A basic web service application was created that queries the MySQL

database for the points data, and returns it to Silverlight as a JavaScript Object Notation
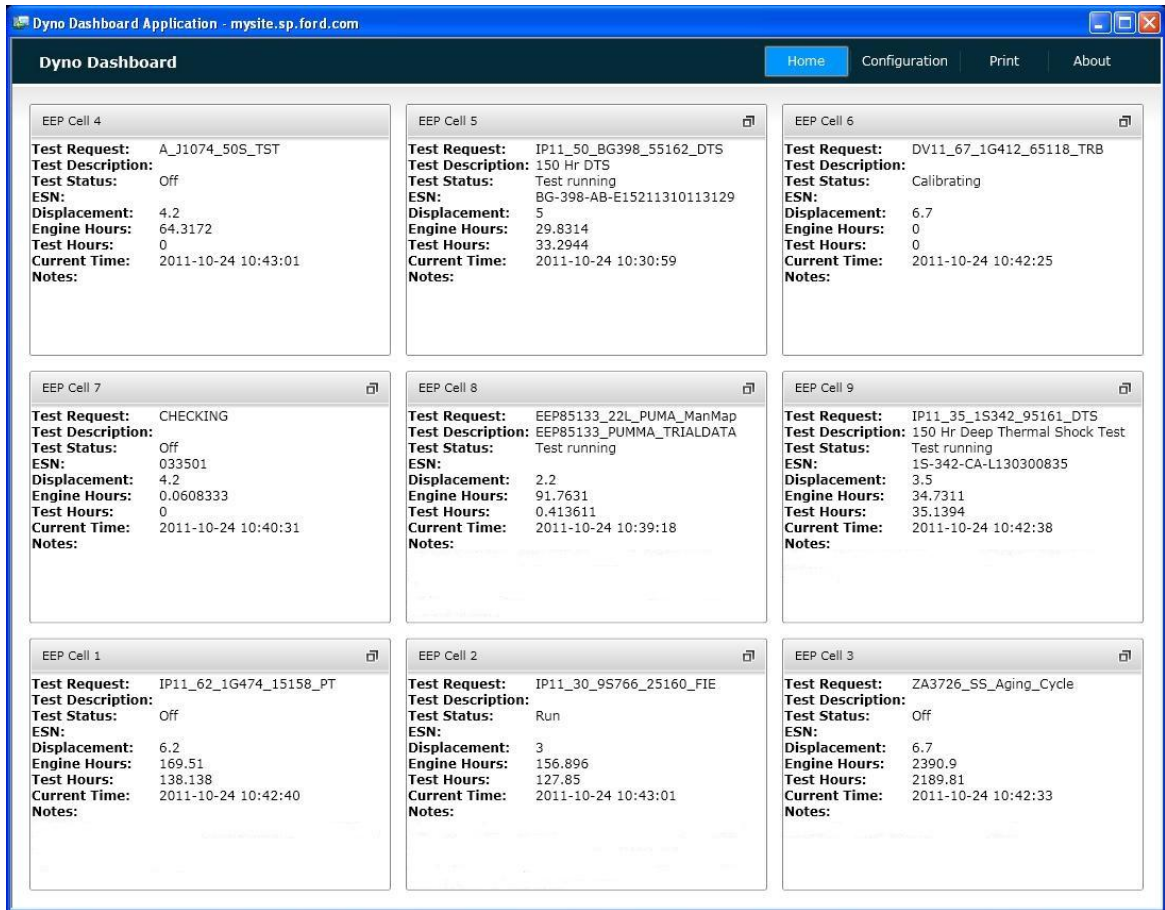
(JSON) object.

Figure I.3: Remote Monitoring Web Application.

From the main screen it is possible to view the details of each individual test cell by clicking on the header of the cell's overview window. This will display the screen shown in Figure I.4. Live engine parameter data from the selected cell are now displayed.

Figure I.4: Remote Point Monitoring.

The details of which points are visible, and the order they are displayed can be configured using the configuration menu. This is shown in Figure I.5. Each list box corresponds to the same list box in the details view. Only points that have been configured by the administrator are available for viewing. This configuration is stored on the user's local machine, along with the option to add personal notes. The personal notes are displayed on the main screen. This is a handy feature to use for reminders.

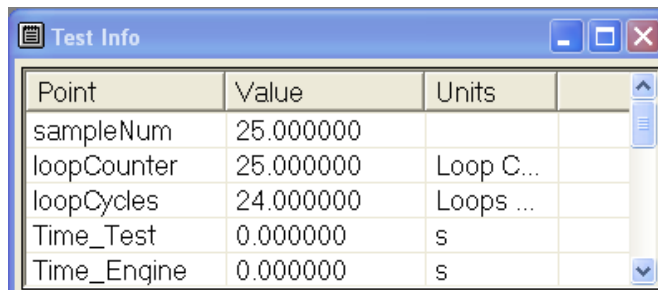Figure I.5: Remote Monitoring Configuration.

The remote monitoring application was developed using a number of simple web and database technologies. It is used on a regular basis for monitoring both the existing ADACS test cells, which had no remote monitoring ability, as well as the new system designed for this thesis. The information provided has been very brief. This was intentional, since the full details would have required a large amount of space. The idea was to show that remote monitoring was developed as part of the complete test cell control system, and to illustrate the technologies involved in implementing it.

147

APPENDIX J: GUI CONTROLS

The engine test cell control system contains a large number of controls and associated configuration dialogs. A brief overview of some of these will be discussed. The general idea of how the other controls would be used and configured is very similar. Visualization controls include tables, gauges, and charts. User input controls, like buttons and tables, are used to manipulate values in the real time database.

J.1 Tables

The table control, shown in Figure J.1, is used to display a list of points and their current values from the real time database.

Figure J.1: Table Control.

The displayed points are selected using the table configuration dialog shown in Figure J.2. The left side list box contains the names of all the points in the real time database. The right list box contains the points that have been selected. There is no limit to the number of points that can be added to a table. Double clicking a point in the table will bring up a dialog that allows the value of a point to be updated in the real time database.
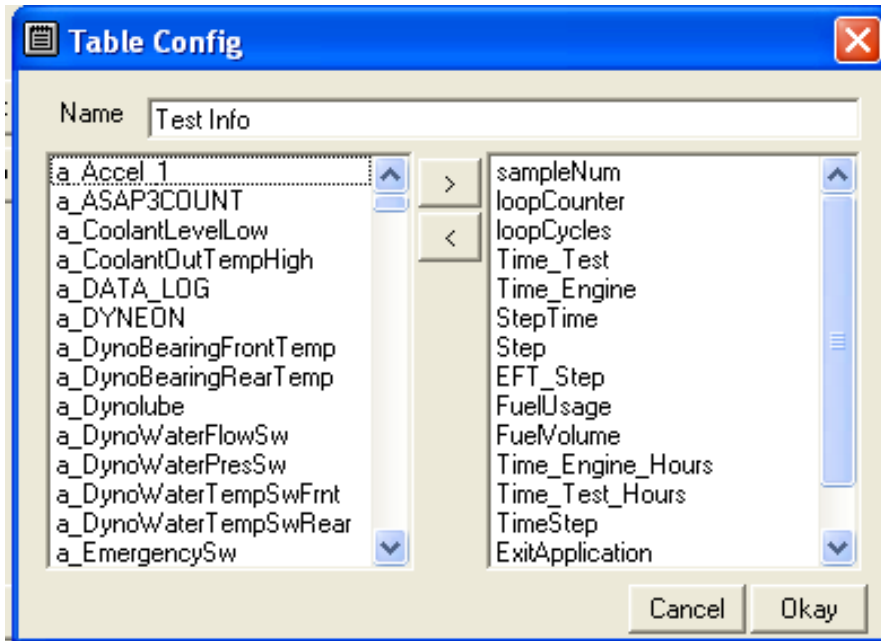
Figure J.2: Table Configuration.

J.2 Gauges

Gauges are used to provide both numerical and visual stimulus, and to attract attention to possible fault conditions. The multiple gauge control is shown in Figure J.3. To the left of the gauge controls are the point names, current values and units. The gauge controls have visual markers for low, warning and high conditions. Low is identified by the colour blue, warning is identified by yellow and high is displayed as red. The actual bar gauge will change to one of these colors if the value of the monitored point is outside the limits. This allows for a quick visual to identify fault conditions. The color green indicates no faults exist. Since color is used for the feedback, it is visible from a long distance away from the computer screen.
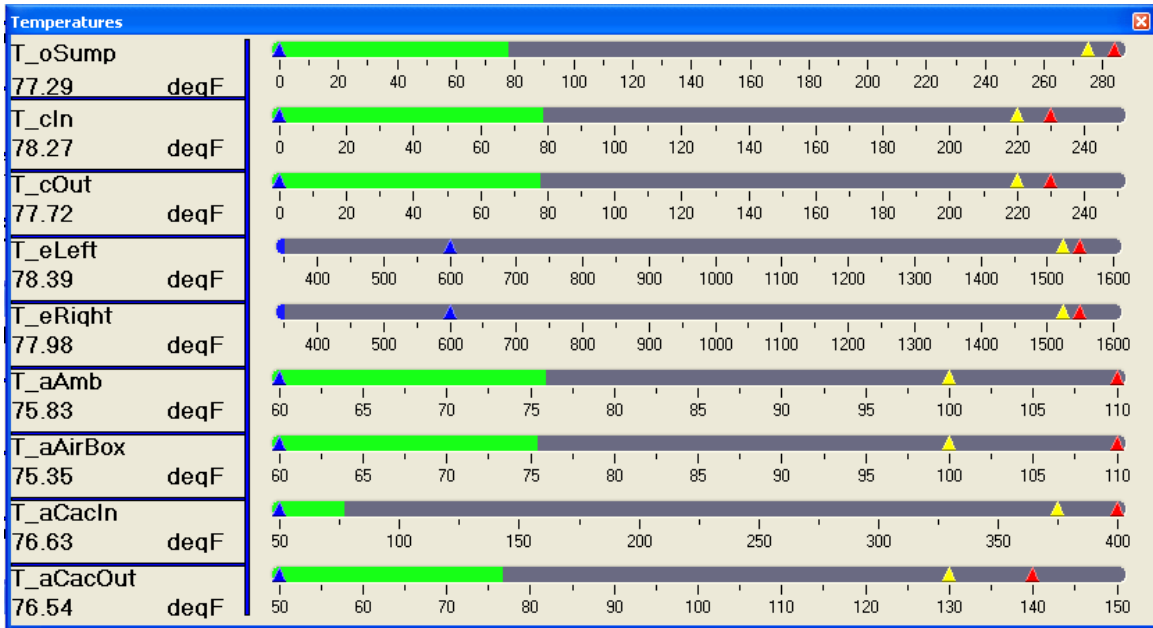
Figure J.3: Gauge Control.

This gauge control is used to group a number of related points together, such as temperatures or pressures. Any number of gauges can be added to the control using the configuration screen shown in Figure J.4. The border around the gauges has a Form Name and is assigned a text value that gives meaning to the contained information. The configurable properties are self explanatory and will not be discussed further.
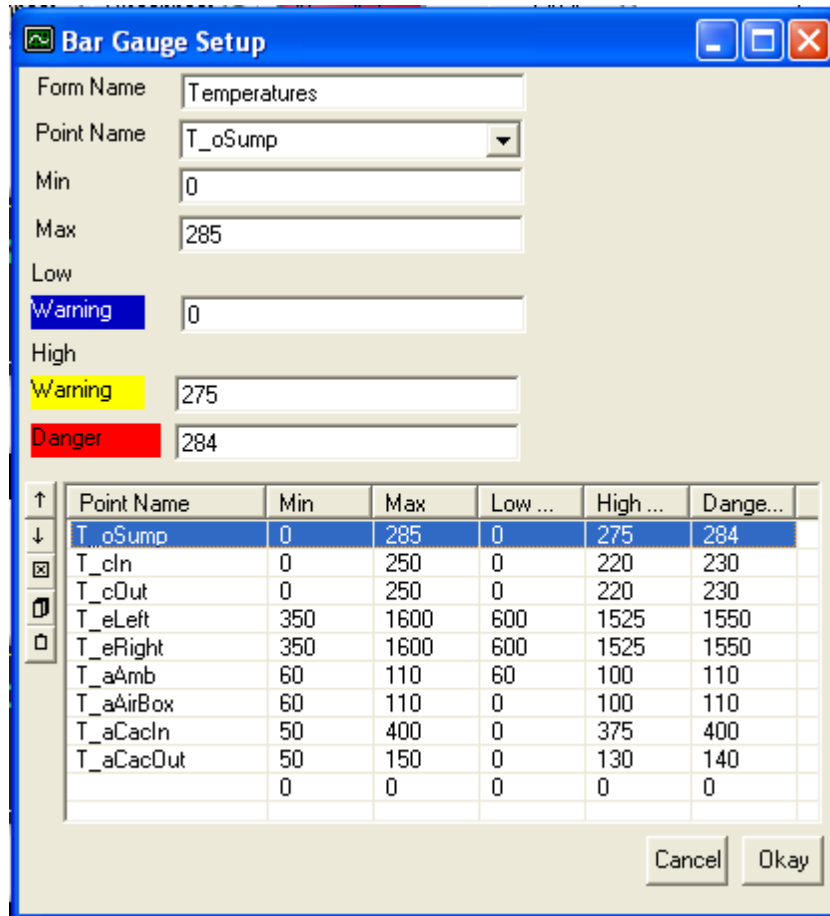
Figure J.4: Gauge Configuration.

J.3 Push Buttons

Buttons are clickable controls that have two states, on and off. A physical push button is normally classified as either a momentary or a toggle switch. A toggle switch will maintain its state when switched and a momentary switch will not. A good example of a toggle switch is the switch used to turn room lights on and off. Examples of momentary switches are ones used on a power tools, such as a drill or circular saw. In this application, the momentary switches appear as shown in Figure J.5 and toggle switches appear as shown in Figure J.6.
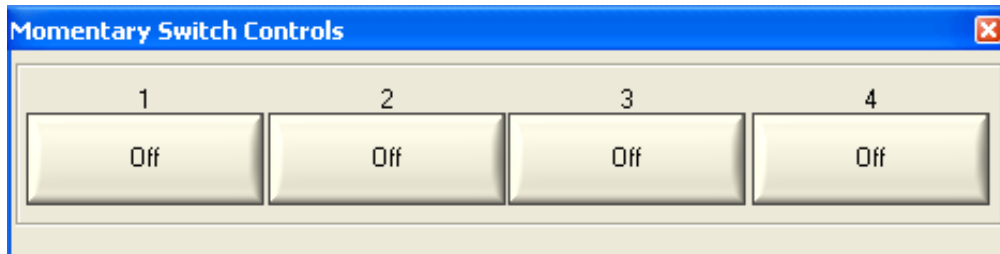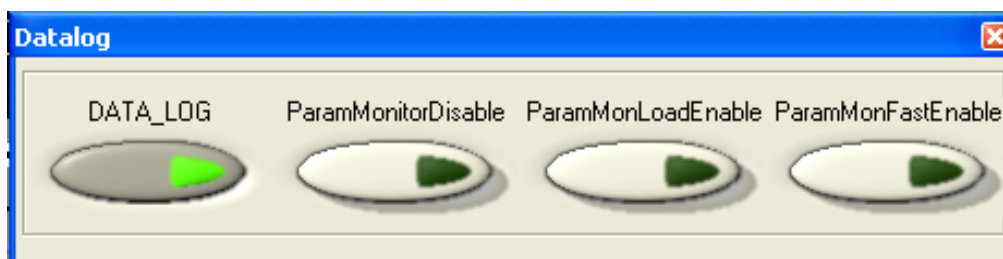
Figure J.5: Momentary Push Buttons.

Figure J.6: Toggle Switches.

The properties of the buttons are configured using the dialog shown in Figure J.7. A point name from the real time database is assigned to each switch. There is a limit of four buttons on each of these controls. The control will resize itself according to the number of points configured. The buttons have an "On" and an "Off" value associated with them. The point assigned to a momentary button will always have the "Off" value, unless the button is held down with the mouse, and assumes the "On" value. There is no restriction on the values of "On" and "Off". This allows the flexibility to create normally open and normally closed types of switches. It also enables one of two decimal values to be assigned to a point. Typically these would be 1 and 0, but there are times when an analog value may be required.
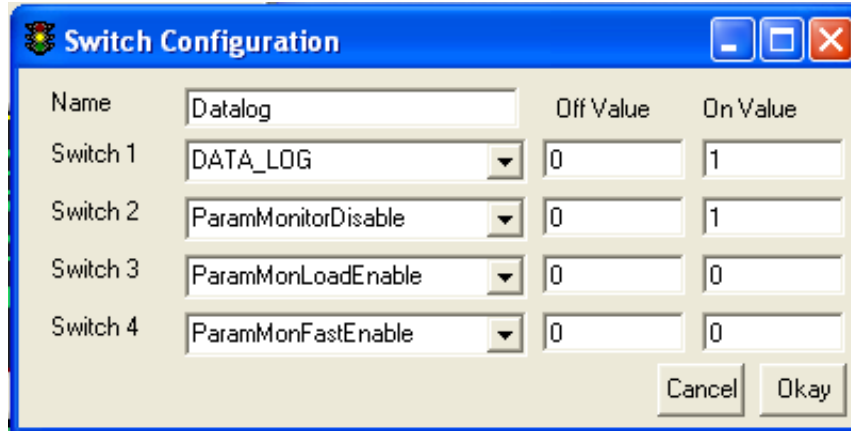
152

Figure J.7: Switch Configuration.

J.4 Line Charts

Line charts are a very powerful method to display historical data. When running an engine test cell, the current state of the engine is displayed using tables and gauges. These are very useful for obtaining the exact values of points. With so many of these variables on the screen, it would be humanly impossible to discern irregular trends using these controls. The chart control was created to store thirty minutes of historical data on the screen. This provides a long enough history to identify trends, as well as ensure that a test is proceeding correctly. It is also a great learning tool, since interactions between variables can be seen while the engine is running. The chart control is displayed in Figure J.8.
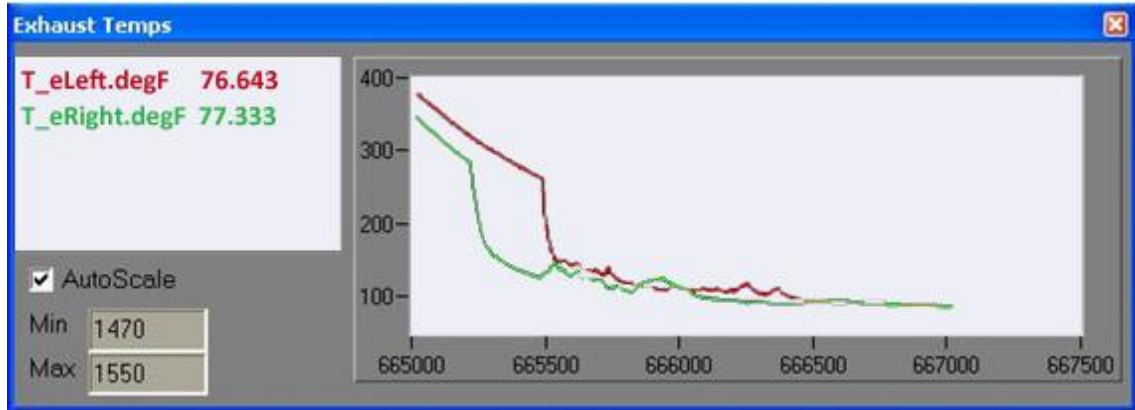
Figure J.8: Line Charts.

J.5 Fixed Controls

There are a number of fixed controls on the main window of the application. These controls are fixed because they are always required regardless of the test being run. Since screen real estate is limited, a minimum number of controls were used. The largest of these are the speed, torque and throttle controls. Only the engine speed controls are displayed in Figure J.9.
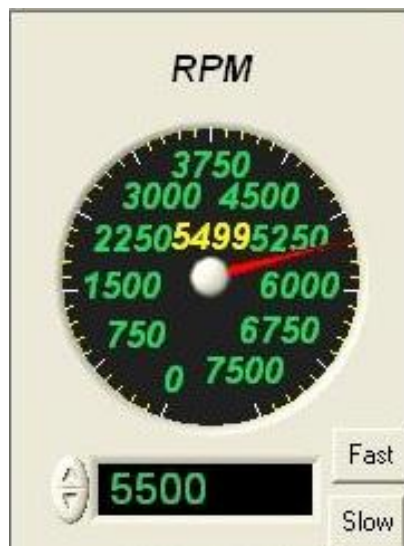

Figure J.9: Manual Speed Setpoint Control.

Each cluster of these controls contains a gauge with two indicating needles. The red needle shows the current point value and a yellow needle is used to show the requested set point. There is also a numeric input control for entering a setpoint value. Finally, two buttons are available to select a slow or fast ramp to the setpoint.

The top of the form contains a menu that provides access to all of the features in the application. Just below the main menu are buttons to start and stop communication with the real time database, as well as an alarm indicator and message window. This is shown in Figure J.10.

Figure J.10: GUI Fixed Form Controls, Top.

The bottom of the form contains the tabs, status bar and a number of fixed buttons shown in Figure J.11. The status bar displays the number of Ethernet packets sent and received, as well as the response from some of the ASCII commands.
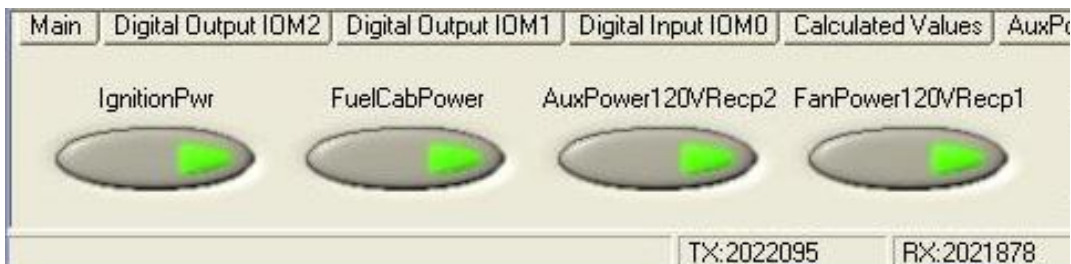

Figure J.11: GUI Fixed Form Controls, Bottom.

APPENDIX K: DAQ HARDWARE

K.1 Sensoray 2600

The Sensoray 2600 series is a low cost, Ethernet based data acquisition device used for slow speed data collection. These devices are not intended for high speed data acquisition, but have some valuable features. There are drivers for both Linux and Windows. The drivers for Linux are distributed in source code form, which was ported to QNX as part of this project. Table K.1 shows a full list of the devices that were used.

Table K.1: Sensoray 2600 Modules.

Module	Specifications	Cost
2601	Main communication module	$463
2608-8 **(3)**	16 differential inputs of voltage, thermocouple, or 4-20 mA,16-bit A/D 8 analog outputs 15 -bit D/A with remote sensing	$532
2620	Four 32-bit quadrature encoders PWM, period, frequency measurement Periodic, single-shot outputs	$345

The 2601 module, shown in Figure K.1, is the main hub which communicates over Ethernet. This module has 16 ports of serial communication for connecting the other 2600 series devices. The 2601 communication module has a watchdog timer that can be enabled to reset all of the other modules if communication to the host is lost.
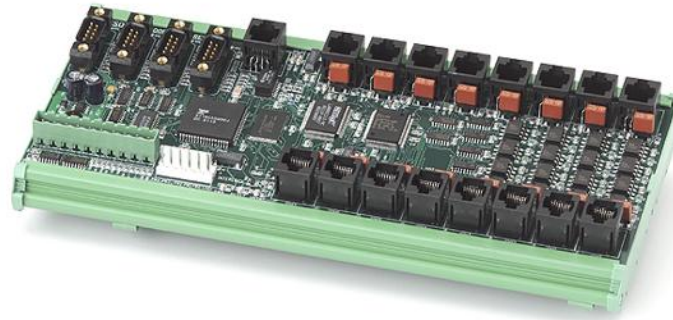
Figure K.1: Sensoray 2601 Master Module, Image Courtesy of Sensoray [38].

The 2608 analog modules, shown in Figure K.2, are primarily used for thermocouple measurements. They have cold junction temperature sensors for thermocouple measurement compensation. There is also on board circuitry to enable broken thermocouple detection. Broken thermocouples are common in engine testing and sometimes hard to detect if a circuit has heavy filtering. The device has removable terminal blocks for wiring connections directly onto the module. This reduces installation issues.
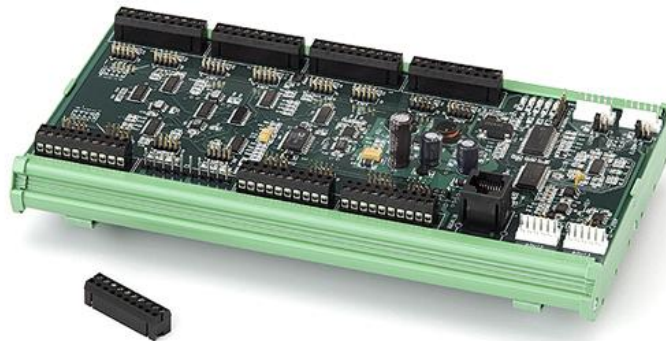

Figure K.2: Sensoray 2608 Analog I/O Module, Image Courtesy of Sensoray [38].

The 2620 module, shown in Figure K.3, is primarily used for frequency measurements. The fuel panel in the engine test cell outputs a frequency that is proportional to fuel flow. Currently, this is the primary application of this module.

Figure K.3: Sensoray 2620 Timer Counter Module, Image Courtesy of Sensoray [38].

K.2 National Instruments PCI 2630

The National Instruments PCI 2630, is an industrial M-Series data acquisition device with bank isolated inputs and outputs. This card is used for the interface with the dynamometer controller. National Instruments has a limited driver development kit for QNX that supports this board [39].

APPENDIX L: THROTTLE ACTUATOR

One of the requirements of a test is to control the load of the engine. The PCM uses the accelerator pedal to read the demanded engine load. The pedal in most engine test cells is controlled using a mechanical actuator. In some cases, the pedal signals are simulated using an analog output. The requirements for the throttle actuator are listed in Table L.1.

Table L.1: Throttle Actuator Requirements.

Requirements
Actuator is portable
Easy pedal setup
Accurate position control

The final design of the throttle actuator was based on a review of commercially available devices [40]. The design uses a combination of a servo motor and a variable frequency drive. The motor output shaft is directly connected to a stroke adjustment arm without a gear box.

Commonly available parts were purchased to complete the design and installation. The list of major components purchased for the throttle actuator is found in Table L.2. This listing does not include any of the extruded aluminum or casters used for the stand. The aluminum parts for the servo motor fixture and all the details were designed and manufactured by PERDC machinists. The stand was designed and built by PERDC tinsmiths. Finally, the electrical installation was completed by PERDC electricians.

Table L.2: Throttle Actuator Parts.

Part #	Description	Cost
1414PHM8	Hammond Electrical Enclosure 14x14x8 inches	172.40
MPL-A4540F-MJ22AA	Allen Bradley Low Inertia Servo Motor	1552.80
2098-DSD-020X	Allen Bradley Ultra 3000 Servo Drive	1713.00
2090-XXNPMP-16S03	MP Series Servo Motor Power Cable 3 Meters	86.36
2090-UXNFBMP-S03	MP Series Servo Motor Feedback cable 3 Meters	103.17
2090-U3BB2-DM44	Ultra 3000 CN1 Breakout board	87.79
	Total	3715.52

L.1 Electrical Design

The electrical design of the throttle controller had only a couple of requirements. First, the design needed an E-Stop for safety. Secondly, in order to work with the existing test cells, the position command needed to be a 0-10VDC signal. Finally, the system needed to run from a 120VAC, 15A receptacle to meet the portability requirements. The final electrical schematic is shown in Figure L.1.
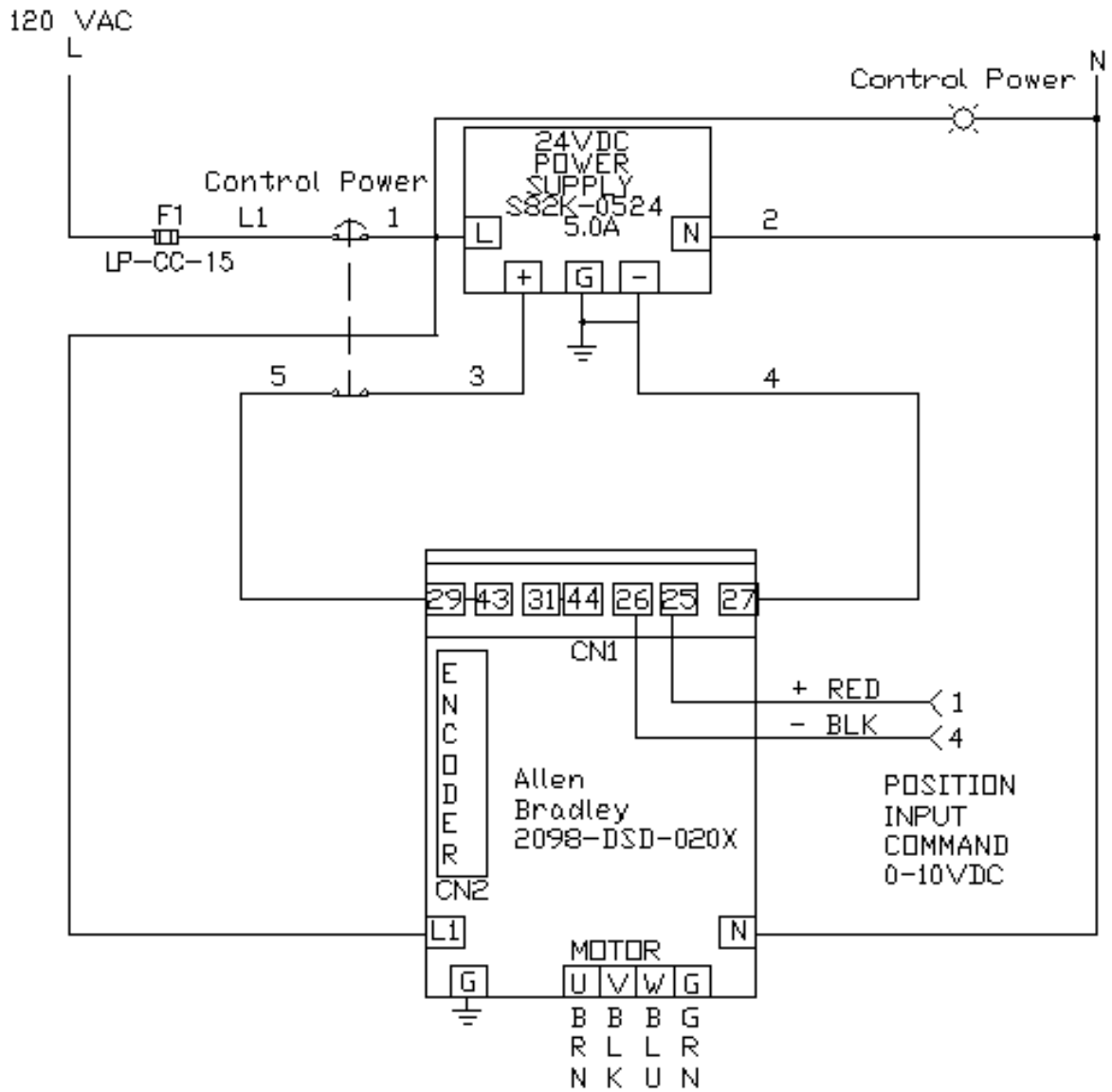
Figure L.1: Throttle Actuator Electrical Schematic.

The Allen Bradley 3000i servo drive includes a basic motor position controller [41]. Although this is not normally a component of a servo drive, it simplified the design significantly in this case. The position controller is normally another device that commands the servo drive.

L.2 Mechanical Design

The throttle actuator is shown mounted on a portable stand in Figure L.2 below.

Figure L.2: Throttle Actuator Mount.

The throttle actuator includes mechanical zero and stroke adjustment arms, as seen in Figure L.3. The zero adjustment is used to configure the pedal position to a location that registers a value of 0 in the PCM. The stroke adjustment arm is used to set the full travel position of the pedal.
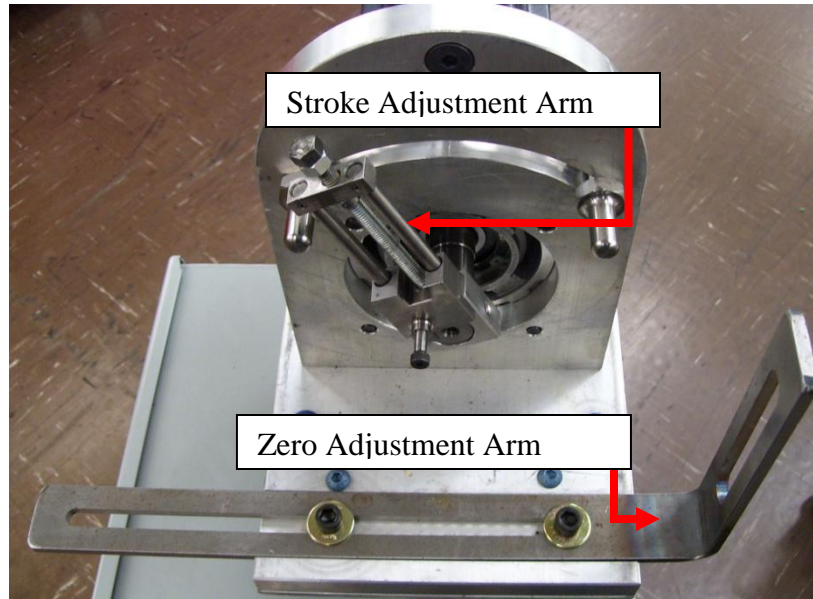
Figure L.3: Throttle Actuator Adjustment Arms.

L.3 Performance Specifications

The throttle actuator was programmed to perform automatic zero calibration when turned on. This eliminated any need for setup. The zero calibration is done by driving the motor until the zero end stop is reached. At the point of contact, the motor current increases. This indicates the zero position is reached. From this point the actuator will move 50 counts off of the zero location. This allows a small amount of over shoot during fast ramps. Table L.3 shows the full specifications for the throttle actuator.

Table L.3: Throttle Actuator Specifications.

Parameter	Description
Encoder Pulses Per Revolution	16000
Position Command Resolution	14 bit
Position Counts Per Volt	130
Homing Current	3.0 Amps
Homing Velocity	5 RPM
Angle of Rotation	30 Degrees
Range of Stroke	1.5 – 7 cm

VITA AUCTORIS

NAME: Anthony Fountaine

PLACE OF BIRTH: Ontario, Canada

YEAR OF BIRTH: 1971

EDUCATION: University of Windsor, Ontario
                               1991-1996, BASc., Electrical Engineering