IC ENGINE AIR/FUEL RATIO FEEDBACK
CONTROL DURING COLD-START

A Thesis

Presented in Partial Fulfillment of the Requirements for
the degree Master of Science in the
Graduate School of The Ohio State University

by

Kenneth C. Leisenring, B.S.E.E.

The Ohio State University

1995

Master's Examination Committee:

Giorgio Rizzoni

Vadim Utkin

Approved by

A handwritten signature in black ink, reading "Giorgio Rizzoni", is written over a horizontal line.

Adviser

Mechanical Engineering

To My Wife and Son

ACKNOWLEDGMENTS

I would like to express my appreciation to Dr. Rizzoni for his support and guidance throughout this research and to Dr. Utkin for his input and suggestions. Special thanks go to the individuals at Ford's Advanced Powertrain Research Group for all of their assistance and generosity, which greatly facilitated the completion of this work . Thanks to Joe West for his invaluable help with the experimental setup for this work. Thanks also go out to the members of the EECAD lab for their help with the many things that could not have been completed without their contributions; especially Benny Samimy, Gary Parker, and Daria Kotys.

VITA

December 22, 1969..... Born - Fullerton, California
1993..... B.S.E.E., Michigan
Technological University,
Houghton, Michigan

FIELDS OF STUDY

Major Field: Mechanical Engineering
Studies in Control Systems and System Dynamics.
Faculty adviser: Prof. Giorgio Rizzoni.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGMENTS	iii
VITA.....	iv
LIST OF TABLES	vii
LIST OF ILLUSTRATIONS.....	viii
INTRODUCTION.....	1
General.....	1
Contribution	1
Literature Review	2
CHAPTER	
I. THE AIR/FUEL RATIO CONTROL PROBLEM.....	5
Current Control Strategies	5
Proposed Solution	7
II. ALGORITHM DESCRIPTION	9
Overview	9
Equivalent Heat Release Duration (EHRD).....	9
Equivalent Torque Production Duration (ETPD)	12
Estimation of Air/Fuel Ratio	13
Estimation of Indicated Torque.....	14
III. FUZZY CONTROLLER	15
Overview.....	15
Controller Selection.....	16
Input Fuzzification.....	16
Rules	20
Output Defuzzification.....	21

IV. EXPERIMENTAL RESULTS	25
Overview	25
Instrumentation	25
Experimental Results	28
Phase One: Preliminary Analysis	30
Phase Two: Real-Time Control	41
 V. CONCLUSIONS AND RECOMMENDATIONS	 64
Conclusions	64
Contribution	64
Recommendations	65
BIBLIOGRAPHY	66
APPENDIX A: DIGITAL SIGNAL PROCESSING SYSTEM	68
Overview	68
Preliminaries	69
Program Listings for Open-Loop and Closed-Loop Experiments	75

LIST OF TABLES

Table	Page
Table 1. EHRD Error Membership Functions	17
Table 2. Time Since Start Membership Functions.....	17
Table 3. Engine Speed Error Membership Functions.....	18
Table 4. Membership Function Weights for EHRD Error Levels.....	19
Table 5. Membership Function Weights for Time Since Start Levels.....	19
Table 6. Membership Function Weights for Engine Speed Error Levels	20
Table 7. Output Membership Functions	21
Table 8. Fuzzy Controller Rules.....	23
Table 9. Fuzzy Controller Rules Continued.....	24
Table 10. Engine Specifications	26
Table 11. EHRD Data for Cold-Start Tests.....	53

LIST OF ILLUSTRATIONS

Figure	Page
Figure 1. Cylinder Pressure.....	11
Figure 2. Heat Release Rate.....	11
Figure 3. Indicated Torque	13
Figure 4. Block Diagram of Control System	16
Figure 5. Engine Instrumentation Diagram.....	27
Figure 6. Engine Speed for Phase One Tests.....	31
Figure 7. Mass Air Flow for Phase One Tests	32
Figure 8. Fuel Injector Pulse Width for Phase One Tests.....	33
Figure 9. Engine Coolant Temperature for Phase One Tests.....	34
Figure 10. EHRD for Phase One Tests.....	35
Figure 11. 10-90% Burn Duration for Phase One Tests.....	36
Figure 12. IMEP for Phase One Tests.....	37
Figure 13. 10-90% Burn Duration vs. EHRD.....	38
Figure 14. 10-90% Burn Duration vs. EHRD.....	39
Figure 15. IMEP vs. EHRD.....	40
Figure 16. Engine Speed for Phase Two Tests	42
Figure 17. Mass Air Flow for Phase Two Tests	43
Figure 18. Fuel Injector Pulse Width for Phase Two Tests.....	44
Figure 19. Desired Lambda for Phase Two Tests.....	45
Figure 20. EHRD for Phase Two Tests.....	46
Figure 21. Desired Lambda for Warm-Start Tests.....	48
Figure 22. EHRD for Warm-Start Tests.....	49

Figure 23. Desired Lambda for Desired EHRD=64 Tests.....	51
Figure 24. EHRD for Desired EHRD=64 Tests	52
Figure 25. Desired Lambda for Initial Lambda = 0.7 Tests.....	56
Figure 26. EHRD for Initial Lambda = 0.7 Tests.....	57
Figure 27. Desired Lambda for Initial Lambda = 0.9 Tests.....	58
Figure 28. EHRD for Initial Lambda = 0.9 Tests.....	59
Figure 29. Desired Lambda for -10% Flow Rate Error Tests.....	60
Figure 30. EHRD for -10% Flow Rate Error Tests	61
Figure 31. Desired Lambda for +10% Flow Rate Error Tests.....	62
Figure 32. EHRD for +10% Flow Rate Error Tests	63
Figure 33. Digital Signal Processing System	69

INTRODUCTION

General

Legislation pertaining to automobile emissions has caused an increased focus on the cold-start performance of internal combustion engines. Of particular concern is the period of time before all available sensors become active. Present engine control strategies must rely on methods other than feedback control while these sensors are not active. Without feedback control during this critical period, engine emissions performance is not optimized. These conditions cause difficulty in performing comprehensive cold-start experiments. For these reasons, some methods have been developed for feedback control during cold-start to aid in laboratory investigations of engine emissions phenomena. Combustion pressure feedback and estimated indicated torque are used in conjunction with a fuzzy logic controller to control air/fuel ratio during cold-start.

Previous work has shown that it is possible to evaluate the air/fuel ratio by the use of cylinder pressure measurements if the air/fuel ratio effects could be separated from the other effects [1]. It is proposed here to use the Equivalent Heat Release Duration (EHRD) as a means of separating these effects. In this method, the combustion pressure is used to find the heat release rate, and, indirectly, the combusted air/fuel ratio. It is proposed to use combustion pressure feedback and a fuzzy logic controller to control air/fuel ratio during cold-start.

Contribution

The primary goal of this work was to develop a method for air/fuel ratio control using combustion pressure feedback during cold-start to be used as an aid in laboratory

experiments. The effects of varying air/fuel ratio during cold-start are so profound that small differences in air/fuel ratio can create effects that will mask the effects of significant changes in other variables. The ability to control air/fuel ratio is an important aid in comprehensive emission studies during cold-start. This work will facilitate future studies of cold-start emissions.

Literature Review

A review of literature was conducted to address the state of combustion pressure based control strategies and to define the present techniques for air/fuel ratio estimation in IC engines. What follows in this section is a review of the history of cylinder pressure based control strategies. Included in this section is a review of past work that has led to and formed the basis for the present studies in the area of air/fuel ratio estimation. Presently, there are many methods being developed for the estimation of air/fuel ratio. These methods are explained in detail in Chapter I.

The first work involving combustion feedback control was performed in 1946 by Cook, Heinicke, and Haynie. [2] Their investigation involved using cylinder pressure rise feedback for spark timing control. One of their conclusions was that the maximum rate of pressure rise and the travel of the flame front were directly related. The possible benefits of cylinder pressure based feedback control for automobile engines have long been acknowledged as many and valuable. However, the economics of sensor application has prohibited the development of production control systems using cylinder pressure feedback. This has recently changed with the addition of a combustion pressure sensor on a production Toyota lean burn engine. [3] Toyota uses combustion pressure feedback to locate the lean limit in air/fuel ratio in order to run as lean as possible without encountering driveability difficulties. This implementation could quite possibly encourage

the proliferation of cylinder pressure based feedback control throughout the automotive industry in the coming years.

A very useful combustion characteristic that can be obtained with the implementation of a combustion pressure transducer is the heat release rate. [4] Research has been done on internal combustion (IC) engines using the heat release rate since the 1930's with the work done using motion pictures of engine flames by Rassweiler and Withrow. [5] This work was able to show that the heat release rate could be calculated using cylinder pressure data.

Air/fuel ratio control is important for emission performance. Catalyst efficiency is high within a very narrow range of air/fuel ratio. [6] During cold-starts, the production exhaust gas oxygen (EGO) sensors are inactive. This forces production control strategies to run open-loop and use look-up tables for air/fuel control during cold-start. These open-loop systems are unable to compensate for disturbances such as fuel quality, aging over time, manufacturing tolerances, and ambient conditions. All of these disturbances can degrade the performance of these open-loop strategies. [7] Even when the EGO sensors are active, they still only yield information of the air/fuel mixture being rich or lean of stoichiometry. This switch-like behavior does not enable closed-loop operation at conditions other than stoichiometry. [6]

Recently, a great deal of effort has been spent developing methods for estimating air/fuel ratio in IC engines. As mentioned earlier, several different methods have been developed to estimate air/fuel ratio. A major drawback with these methods is that fast, accurate results are needed to perform high bandwidth transient control. [8] The cyclic variations in combustion phenomenon make these kind of results difficult to obtain. In addition to this problem is the fact that the fuel dynamics of the engine is time-varying, event based, and strongly dependent on temperature. Therefore, this work involves low

bandwidth control designed to keep the air/fuel ratio close to a setpoint using closed-loop rather than open-loop control during cold-start.

CHAPTER I

THE AIR/FUEL RATIO CONTROL PROBLEM

Current Control Strategies

Current production control strategies use an EGO sensor to sense air/fuel ratios rich or lean of stoichiometry. This switch-like sensor is commonly fed back to a proportional-integral (PI) controller and the engine control computer calculates the desired amount of fuel to be injected. This sensor is not active during cold-start so there is a need for a method for estimating the air/fuel ratio.

Several methods have been developed for estimating air/fuel ratio. These include algorithms utilizing pressure ratios, pressure moment, molecular weight ratios, and combustion (burn) duration. All of these methods utilize a relationship of the form:

$$\phi = a_0 + a_1 N + a_2 P_{man} + a_3 F_1 + a_4 F_2 \quad (1.1)$$

where:

ϕ : air/fuel ratio

N: engine speed

P_{man} : intake manifold pressure

F_1 : estimate function one

F_2 : estimate function two

The reasons for this form is that the air/fuel ratio is affected by changes in engine speed and manifold pressure in addition to the fuel and air dynamics.

The pressure ratio algorithm uses the relationships:

$$F_1 = P_r \quad (1.2)$$

where P_r is equal to the ratio of the cylinder pressure at a crank angle location after TDC and the pressure at the same crank angle location before TDC. This is designed to

evaluate the energy added due to combustion and it is reasoned that this would vary with air/fuel ratio. [9].

The moment algorithm uses the fact that the pressure trace can be precisely described using an infinite number of moments. Using the formula:

$$M_n = \int_{\theta_o}^{\theta_f} (\theta - \bar{\theta})^n P(\theta) d\theta \quad (1.3)$$

where:

M_n : nth pressure moment

θ_o : crank angle at which integration begins

θ_f : crank angle at which integration ends

θ : crank angle location

$\bar{\theta}$: centroid of $P(\theta)$

$P(\theta)$: pressure time history

the relationships:

$$F_1 = M_2 \quad (1.4)$$

$$F_2 = M_3 \quad (1.5)$$

are used because it has been shown that the second and third moments capture the effects of varied air/fuel ratio. [10]

The molecular weight ratio algorithm uses the relationship:

$$F_1 = \frac{G_2}{G_1} = \frac{P_1 T_2}{P_2 T_1} \quad (1.6)$$

where G_2/G_1 is known as the "G-ratio" and:

P_1 : pressure at crank angle before TDC

P_2 : pressure at crank angle after TDC

T_1 : temperature at crank angle before TDC

T_2 : temperature at crank angle after TDC

This method was developed because Patrick showed in 1989 that pressure-based algorithms exhibit some error during warm-up due to sensitivities to charge temperature. It is not presently possible to measure the temperature of the charge in the cylinder, nor does it seem reasonable that an effort will be made to make this available in production vehicles.

The combustion duration algorithm uses the relationship:

$$F_1 = B_t \quad (1.7)$$

where B_t is the 0-90% burn duration. This method was developed because of the strong influence air/fuel ratio has on the heat release rate. [11]

Proposed Solution

Two methods in the previous section have desirable characteristics for a real-time control system. The moment algorithm allows calculations to be made by the use of an integral. If there is no need to save a cycle of pressure data, this would save important memory in engine control computers. A running sum can be kept and updated at an interval of a given duration of crank angle degrees. The combustion duration algorithm uses a very strong relationship between air/fuel ratio and heat release rate to perform estimation. A drawback of this method is that a portion of a cycle of data has to be saved and then calculations performed on the data twice before a final result is reached.

The proposed method described in Chapter II, the EHRD combines the best parts of both of the above methods. This algorithm uses the heat release rate and moments to develop a relationship with EHRD where:

$$F_1 = EHRD \quad (1.8)$$

This can be used to estimate air/fuel ratio during cold-start. This method is both easy to calculate and sensitive to changes in air/fuel ratio. This method ignores the effects of engine speed and intake manifold pressure on the air/fuel ratio because both remain

relatively constant during cold-start. Their effects are minimal compared to the dependency the EHRD has on the air/fuel ratio.

CHAPTER II

ALGORITHM DESCRIPTION

Overview

This chapter describes the method proposed in this work, the Equivalent Heat Release Duration, and an alternate method for use in future research, the Equivalent Torque Production Duration. The equations for all steps are given and important issues concerning implementation and methodology are discussed. The estimation of air/fuel ratio using the two methods is explained in detail. Finally, the estimation of indicated torque from crankshaft acceleration is discussed for use in the implementation of the Equivalent Torque Production Duration method.

Equivalent Heat Release Duration (EHRD)

The method of the Equivalent Heat Release Duration (EHRD) is described below. The apparent heat release rate in joules per degree ($dQ/d\theta$) can be found from equation (2.1) below. [12] The heat transfer between the cylinder gases and the cylinder walls is neglected in this calculation. The heat release calculated in equation (2.1) is assumed to be caused from the work done by the cylinder gases only.

$$\frac{dQ}{d\theta} = \left(\frac{1}{\gamma - 1} \right) V \frac{dP}{d\theta} + \left(\frac{\gamma}{\gamma - 1} \right) P \frac{dV}{d\theta} \quad (2.1)$$

where P is the cylinder pressure and V is the cylinder volume. The equation for V is derived in [13] and the equation for $dV/d\theta$ can be obtained from differentiating equation (2.2).

$$V = A_p \left(\frac{2a}{r_c - 1} + l + a - a \cos \theta - \sqrt{l^2 - a^2 \sin^2 \theta} \right) \quad (2.2)$$

where:

A_p : Piston Area

r_c : Compression Ratio

l : Connecting Rod Length

a : Crank Radius

$$\frac{dV}{d\theta} = A_p \left(a \sin \theta + \frac{a^2 \sin \theta \cos \theta}{l^2 - a^2 \sin^2 \theta} \right) \quad (2.3)$$

From this, the total apparent heat release (Q) and the Heat-Degree Moment ($MomQ$) can be calculated as follows:

$$Q = \int_{\theta_i}^{\theta_f} \frac{dQ}{d\theta} d\theta \quad (2.4)$$

$$MomQ = \int_{\theta_i}^{\theta_f} \frac{dQ}{d\theta} (\theta - \theta_s) d\theta \quad (2.5)$$

The limits of integration for these equations are from the degree location after the spark event at which the heat release becomes positive to the degree location after the start of combustion at which the heat release decreases to zero. The value θ_s is the degree location of the spark event. These two calculations are then used to find the Equivalent Heat Release Duration as shown below.

$$EHRD = \frac{MomQ}{Q} \quad (2.6)$$

the units of EHRD are degrees. EHRD can be thought of as the number of degrees after the spark event at which all of the heat would be instantly released to create a moment

equal to $MomQ$. This is analogous to the equivalent location of a distributed load in statics. Figures 1 and 2 show a typical cylinder pressure trace and the corresponding heat release rate trace.

Figure 1. Cylinder Pressure

Figure 2. Heat Release Rate

Equivalent Torque Production Duration (ETPD)

The method of the Equivalent Torque Production Duration (ETPD) is very similar to the EHRD method. In this calculation, the integrated variable is indicated torque (T_i) rather than the heat release rate. The relationship between the indicated torque and the cylinder pressure during the power stroke is derived in [14] and is shown below:

$$T_i(\theta) = g(\theta) \cdot P \quad (2.7)$$

with

$$g(\theta) = AR \cdot f(\theta) \quad (2.8)$$

and:

$$f(\theta) = \sin(\theta) + \frac{R \sin(\theta) \cos(\theta)}{\sqrt{L^2 - R^2 \sin^2(\theta)}} \quad (2.9)$$

where A is the cylinder cross-sectional area, R is the crank radius, and L is connecting rod length.

The integrated torque ($\text{Sum}T_i$) and the Torque Production Moment ($\text{Mom}T_i$) can then be calculated as follows:

$$\text{Sum}T_i = \int_{\theta_i}^{\theta_f} T_i(\theta) d\theta \quad (2.10)$$

$$\text{Mom}T_i = \int_{\theta_i}^{\theta_f} T_i(\theta) \cdot (\theta - \theta_s) d\theta \quad (2.11)$$

The limits of integration for these equations are from compression TDC to the degree location after TDC at which the torque production falls below a nominal value. The value θ_s is the degree location of the spark event. These two calculations are then used to find the Equivalent Torque Production Duration as shown below.

$$\text{ETPD} = \frac{\text{Mom}T_i}{\text{Sum}T_i} \quad (2.12)$$

The units of ETPD are also degrees. ETPD can be thought of as the number of degrees after the spark event at which all of the torque would be instantly released to create a moment equal to $MomTi$. This is similar to the discussion of EHRD. The indicated torque trace shown in Figure 3 corresponds to the cylinder pressure trace in Figure 1.

Figure 3. Indicated Torque

Estimation of Air/Fuel Ratio

It is known that the effects on combustion of varying air/fuel ratio can be observed with the aid of cylinder pressure measurements. In particular, the heat release rate is inversely related to the combusted air/fuel ratio. This means that a rich mixture will have a shorter 10-90% burn duration than a lean mixture. The results of the EHRD calculation for a rich mixture would also give a shorter duration than that for a lean mixture.

It can be shown that the value computed for EHRD is proportionally related to the combusted air/fuel ratio. However, due to difficulties involved in calculating the actual air/fuel ratio during cold-start, the calculated values for the EHRD are compared to

variables that are known to vary with air/fuel ratio, Indicated Mean Effective Pressure (IMEP) and the previously mentioned 10-90% burn duration. The difficulties arise from nonlinear fuel dynamics models described in [15] that are extremely involved to calibrate for cold-start which make it impractical to calculate air/fuel ratio from the mass air flow and known fuel pulse widths.

In this work, the variable that will be controlled will be the EHRD. Since this value will be shown to be proportional to IMEP and 10-90% burn duration, and therefore air/fuel ratio, the air/fuel ratio will be indirectly controlled. This is acceptable because the point of concern during cold-start might actually be combustion quality. There could be a strong desire to run an engine as lean as possible without encountering unacceptable combustion. If care is taken during measurement and the proper averaging is performed, it is possible to get a reasonable estimation of combustion quality using the EHRD algorithm. This work involves using this algorithm for feedback control while the usual sensors are not available. This results for ETPD are very similar to those for the EHRD. However, the ETPD method is less sensitive to changes in air/fuel ratio. This is due to the effect of the multiplication in (2.7).

Estimation of Indicated Torque

In practice, the indicated torque used in equations (2.10), (2.11), and (2.12) would not be obtained using the cylinder pressure measurements, since the goal of the ETPD method would be to estimate combustion quality in a less costly manner than that of the EHRD method. The indicated torque would instead be obtained from modeling of the engine crankshaft dynamics [16]. Knowledge of the physical dimensions of the engine, accurate engine speed measurements, and a cold-engine friction model would be needed to calculate the indicated torque, and therefore, the ETPD. This could be the topic of a future thesis.

CHAPTER III

FUZZY CONTROLLER

Overview

This chapter explains the design and function of the fuzzy logic controller. The reasons for selecting fuzzy logic as the method of control are given. All aspects of the controller are discussed, including input fuzzification, control rules, and output defuzzification. Either the EHRD or the ETPD method could be used for this work. The EHRD method was chosen because of its increased sensitivity to air/fuel ratio variations and its independence from engine dynamics and friction models.

The fuzzy controller for this system involved several inputs from the engine and provided the desired change in lambda (λ) as an output. This was then used to calculate the new desired lambda and the corresponding fuel pulse width. A discussion of the relationship between lambda and fuel pulse width can be found in the following chapter. The value of EHRD is controlled directly in this work, and therefore, the air/fuel ratio is controlled indirectly. Figure 4 on the following page shows a block diagram of the control system explained in this chapter.

Figure 4. Block Diagram of Control System

Controller Selection

The fuzzy controller was selected for this work because of the previous knowledge gained investigating difficulties controlling the nonlinear, time-varying engine system. A fuzzy controller lends itself to work in which a knowledge base exists on how to control a system. This information is often qualitative rather than quantitative. An example of this type of knowledge related to this control work is the desire to have different responses for rich and lean air/fuel ratio excursions. There are also several inputs to the controller, including cylinder pressure, engine speed, and time since start. Each separate combination of these inputs requires a different control action. A fuzzy controller also lends itself well to this type of system. The fuzzy controller was selected for these reasons.

Input Fuzzification

The inputs to the DSP system were cylinder pressure, spark location, time since engine start, and engine speed. The cylinder pressure and spark location were used to

calculate the EHRD, which was then used as one of the inputs for the fuzzy controller along with the two remaining signals. The inputs were quantized into levels so the rules could be implemented [17]. The EHRD was compared to the desired EHRD to calculate an EHRD error. The membership functions for the EHRD error are shown below in Table 1.

Table 1. EHRD Error Membership Functions

Membership Function	Description
LL	Lean Large
LM	Lean Medium
LS	Lean Small
ZE	Zero Error
RS	Rich Small
RM	Rich Medium
RL	Rich Large

The membership functions for the time since engine start are shown below in Table 2.

Table 2. Time Since Start Membership Functions

Membership Function	Description
EC	Engine Cold
EW	Engine Warm

The engine speed was compared with the desired engine speed to calculate an engine speed error. The membership functions for the engine speed error are shown in Table 3.

Table 3. Engine Speed Error Membership Functions

Membership Function	Description
RN	Running Normal
RS	Running Slow

The reason that there is no running fast condition is that this would be of no concern. The engine idle speed control will decrease the air flow if the engine is running too fast and it would be very difficult to make the engine run faster than desired by decreasing the air/fuel ratio. However, the engine running slow is of concern because this could indicate a stall condition that might be caused by poor combustion due to an undesirable air/fuel ratio. This condition should be corrected by the fuel controller, not the idle speed controller.

As mentioned earlier in this chapter, the inputs were quantized into levels and membership functions were created. A weight alpha (α) was assigned to each membership function for each level of input. This was done for all three inputs. The weights are shown in Tables 4-6. The weights were assigned using the knowledge gained during the phase one tests in this work and by considering the desired performance of the controller and engine system.

Table 4. Membership Function Weights for EHRD Error Levels

Min.	Max.	LL	LM	LS	ZE	RS	RM	RL
-Inf	-10.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
-10.0	-7.5	0.5	0.5	0.0	0.0	0.0	0.0	0.0
-7.5	-5.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
-5.0	-2.5	0.0	0.5	0.5	0.0	0.0	0.0	0.0
-2.5	-1.0	0.0	0.0	0.7	0.3	0.0	0.0	0.0
-1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
1.0	2.5	0.0	0.0	0.0	0.3	0.7	0.0	0.0
2.5	5.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0
5.0	7.5	0.0	0.0	0.0	0.0	0.0	1.0	0.0
7.5	10.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5
10.0	Inf	0.0	0.0	0.0	0.0	0.0	0.0	1.0

Table 5. Membership Function Weights for Time Since Start Levels

Min.	Max.	EC	EW
0	15	1.0	0.0
15	25	0.5	0.5
25	Inf	0.0	1.0

Table 6. Membership Function Weights for Engine Speed Error Levels

Min.	Max.	RN	RS
-Inf	150	1.0	0.0
150	300	0.7	0.3
300	Inf	0.0	1.0

Rules

The rules were formed to consider all important input combinations for the controller output. For example, a warm engine will respond better to changes in fuel pulse width than a cold engine due to the effects of fuel dynamics. Also, since the ultimate desire is to run controlled air/fuel ratios lean of stoichiometry, it is of great concern if large errors occur on the lean side because combustion stability problems will arise quickly as the air/fuel ratio approaches the lean limit. Due to this concern, the changes in desired lambda are more severe when there is an error on the lean side than when an error occurs on the rich side. In terms of system response, it would be desirable to have an over damped response to a rich error and a critically damped or under damped response to a lean error. The engine speed also gives an indication of gross problems with engine speed including stalling. Engine speed was only used to identify these types of problems because the idle speed control feedback loop was also active at this time. The controller does not, nor is it intended to control engine speed. It is intended only to control the EHRD and therefore indirectly, air/fuel ratio.

Output Defuzzification

The output of the controller, as mentioned before, is the new change in lambda. The possible fuzzy outputs and the output action or μ for each possible output is shown in Table 7. The values of μ were first determined from phase one data relating open-loop lambda values and EHRD. The μ values were then altered slightly to optimize controller performance.

Table 7. Output Membership Functions

Membership Function	Description	Action (μ)
IL	Increase Large	0.03
IM	Increase Medium	0.02
IS	Increase Small	0.01
NC	No Change	0.0
DS	Decrease Small	-0.015
DM	Decrease Medium	-0.025
DL	Decrease Large	-0.04

These actions are used in conjunction with the weights of the input membership functions to calculate a new desired delta lambda. The Center of Area (COA) method for defuzzification was used [18]. This approach calculates the control output by taking into effect all of the individual output requirements from the individual rules. The rules are shown in Table 8. The rules are in the form:

If A and B and C, then Z

where:

A: Engine Speed Error Membership Function

B: Time Since Start Membership Function

C: EHRD Error Membership Function

Z: Output Membership Function

The desired new delta lambda is calculated using the α from the three inputs and the μ from the output action for all of the rules for which A, B, and C are true. Let the weights for the engine speed error, time since start, and EHRD error be denoted α_r , α_t , and α_e , respectively. The weight for the i-th rule is given by

$$\alpha_i = \alpha_r \times \alpha_t \times \alpha_e \quad (3.1)$$

The desired change in lambda is then calculated by:

$$\Delta\lambda = \frac{\sum_{i=1}^N \alpha_i \times \mu_i}{\sum_{i=1}^N \alpha_i} \quad (3.2)$$

where:

i: index of satisfied rules

N: number of satisfied rules

Table 8. Fuzzy Controller Rules

A	B	C	Z
RN	EC	LL	DL
RN	EC	LM	DM
RN	EC	LS	DS
RN	EC	ZE	NC
RN	EC	RS	IS
RN	EC	RM	IM
RN	EC	RL	IL
RN	EW	LL	DL
RN	EW	LM	DM
RN	EW	LS	DS
RN	EW	ZE	NC
RN	EW	RS	IS
RN	EW	RM	IS
RN	EW	RL	IM

Table 9. Fuzzy Controller Rules Continued

A	B	C	Z
RS	EC	LL	DL
RS	EC	LM	DL
RS	EC	LS	DL
RS	EC	ZE	DS
RS	EC	RS	DS
RS	EC	RM	DS
RS	EC	RL	DS
RS	EW	LL	DM
RS	EW	LM	DM
RS	EW	LS	DM
RS	EW	ZE	NC
RS	EW	RS	NC
RS	EW	RM	NC
RS	EW	RL	NC

CHAPTER IV

EXPERIMENTAL RESULTS

Overview

The EHRD method described in chapter II has been tested using experimental data taken from a 4.6L V8 engine at the Engine Emissions Control and Diagnostics (EECAD) Laboratory, Mechanical Engineering Department, Center for Automotive Research, at The Ohio State University. Tests were conducted in two phases. In the first phase, the engine was run with open-loop fueling over a range of air/fuel ratios. The data was collected in 2 crank angle degree intervals and written to file. The data was post-processed to study the relationship between EHRD, IMEP and 10-90% burn duration. This data was used to demonstrate the relationships between these variables and to validate the use of the EHRD for real-time control. In the second phase, the EHRD was calculated in real-time using the DSP system. Used in conjunction with the fuzzy logic controller previously described, the fueling was controlled in closed-loop. The instrumentation for both phases was identical and is described below.

Instrumentation

The engine used to perform this work was a Ford Motor Company 4.6L V8 engine. The dimensions of this engine can be seen in Table 10. The engine was connected to a Ford AODE (Automatic Overdrive Electric) transmission and a General Electric 100hp type TLC-50 electric dynamometer. The dynamometer is controlled by a Dyne Systems DYNE-LOC IV Digital Dynamometer Controller and a Dyne Systems DTC-1

Digital Throttle Controller is used to control throttle actuation. All testing was done with the engine running in idle-neutral. A forced cooling system was installed and made it possible to cool the engine to approximately 70 degrees Fahrenheit for the beginning of each test.

Table 10. Engine Specifications

Compression Ratio	9.0
Bore	0.0902 meter
Stroke	.09 meter
Connecting Rod Length	0.1507 meter
Displacement	4.6 liters
Firing Order	1-3-7-2-6-5-4-8

The tests were conducted for approximately the first 45 seconds after engine start. The production engine controller was used to start the engine and then control was switched via relays to the laboratory Electronic Fuel Injection (EFI) controller which was driven by the Spectrum Signal Processing TMS320C30 Digital Signal Processing (DSP) system. This system was installed in a 80486 based personal computer and consisted of an analog input board, analog output board, and system board. The system was also used for data acquisition. The EFI controller was converted from two separate EFI controllers for four cylinder engines to work on the eight cylinder engine in these experiments. The spark timing was fixed to 10 degrees BTDC for the duration of the tests. A representation of the experimental setup was shown previously in Figure 4. The programs used to collect and process the data for these tests are explained in detail in Appendix A and a description of the EFI controller can be found in [19].

The signals that were used for these experiments were cylinder pressure, sync or TDC, engine coolant temperature (ECT), mass air flow (MAF), engine speed, and fuel pulse width. The ECT and MAF signals were taken directly from the Ford production EEC-IV engine controller which has inputs from sensors on the engine. These signals were sent through an isolation amplifier circuit in order to not load down the production sensors. The engine speed and fuel pulse width signals are generated by the EFI controller and also sent through the isolation amplifier circuit. The sync signal is generated by an optical encoder installed on the front of the engine which also generates a once per degree signal that is used to generate the two degree interval trigger signal for the data acquisition. The cylinder pressure is sensed in cylinder one using a Kistler model 6123 pressure transducer with 15.7 pc/Bar sensitivity. This signal was then amplified using a PCB charge amplifier to give a 5 bar/volt signal. The instrumentation of these signals is represented in Figure 5.

Figure 5. Engine Instrumentation Diagram

The fuel pulse width is calculated using the following equations.

$$Fuel_Needed = \frac{MAF * SEC_PER_MIN}{\lambda * RPM * STOICH * CYL_PER_REV} \quad (4.1)$$

where:

Fuel_Needed: Amount of fuel to be injected (pounds)

MAF: Mass Air Flow (lb/hr)

λ : Normalized Air Fuel Ratio

RPM: Engine Speed

STOICH: Air/Fuel Stoichiometric Value (14.64)

SEC_PER_MIN: Seconds Per Minute (60)

CYL_PER_REV: Cylinder Combustion Events Per Revolution (4)

$$Des_PW = MS_CONVERT * \left(\frac{Fuel_Needed}{INJ_FLOW} \right) + INJ_OFF \quad (4.2)$$

where:

Des_PW: Desired Pulse Width (ms)

INJ_FLOW: Injector Flow Rate (lb/sec)

INJ_OFF: Injector Opening Offset (ms)

MS_CONVERT: Conversion from seconds to milliseconds (1000)

Experimental Results

For the first phase of testing the data was collected every two crank angle degrees. It was shown in [20] that this was an acceptable sampling rate for the calculation of the EHRD. The values of ECT, MAF, RPM, and PW were averaged over each engine cycle (720 crank angle degrees). For each engine cycle, the cylinder pressure and sync signal were used to calculate EHRD, IMEP, and 10-90% burn duration for cylinder number one.

The EHRD was calculated as described in Chapter II. The 10-90% burn duration was calculated using equations (2.1) and (2.4). The maximum heat release was found and then the corresponding crank angles at which 10% and 90% of this maximum were reached were denoted as Bt_{10} and Bt_{90} , respectively. The 10-90% burn duration (Bt) was then calculated as:

$$Bt = Bt_{90} - Bt_{10} \quad (4.3)$$

The IMEP was calculated using the below equations from [13]:

$$IMEP = \frac{W_{c,i}}{V_d} \quad (4.4)$$

where:

V_d : Volume Displaced

$W_{c,i}$: Indicated Work Per Cycle

The indicated work per cycle is given by the following equation and is integrated over the compression and expansion strokes.

$$W_{c,i} = \oint P dV \quad (4.3)$$

where:

P : Cylinder Pressure

dV : Rate of Change of Cylinder Volume

The data for this phase can be found in the next section.

For the second phase, only the EHRD was calculated in real time. All data was averaged over eight engine cycles and written to file. This data was collected with the fuzzy controller active running closed-loop. In addition to the engine data that was collected, the output of the fuzzy controller and the levels of the three fuzzy inputs were written to file as well. The data for these experiments are shown in the section below.

Phase One: Preliminary Analysis

Tests were run with open-loop fueling with λ values of 0.7, 0.8, and 0.9 for approximately 45 seconds. Figure 6 shows the engine speed in rpm for the different tests conditions. Three tests are shown for each value of λ . The data is averaged over every eight combustion events of cylinder one (sixteen engine revolutions). It can be seen in Figure 6 that for the $\lambda=0.9$ tests, there was a problem with the engine slowing in the initial few seconds. This was due to poor combustion due to an excessively lean air/fuel ratio. This is evident in the following figures as well. Figure 7 shows the Mass Air Flow rate for the tests. An increase in air flow is seen in the $\lambda=0.9$ tests because the idle speed control system is increasing the air flow to try and increase the engine speed. Figures 8 and 9 show the Fuel Injector Pulse Width and the Engine Coolant Temperature, respectively. The pulse width in the $\lambda=0.9$ tests is high relative to the other conditions due to the increase in air flow previously discussed. Figure 9 illustrates the starting temperatures and the heating of the engine coolant throughout the tests.

Figure 10 shows the EHRD for the tests. As expected the values for EHRD decrease as the values of λ decrease. Two factors account for the change of values of EHRD throughout each test. During the first 5-10 seconds, the engine is not warm and the engine speed is higher than the rest of the test. Due to fuel dynamics, the percentage of total fuel injected that appears as vaporized fuel in the cylinder is less than later when the engine slows down and warms up. The warming of the engine can be seen by the general downward trend of EHRD versus time throughout the duration of the tests.

Figure 6. Engine Speed for Phase One Tests

Figure 7. Mass Air Flow for Phase One Tests

Figure 8. Fuel Injector Pulse Width for Phase One Tests

Figure 9. Engine Coolant Temperature for Phase One Tests

Figure 10. EHRD for Phase One Tests

Figure 11. 10-90% Burn Duration for Phase One Tests

Figure 12. IMEP for Phase One Tests

Figure 13. 10-90% Burn Duration vs. EHRD

Figure 14. 10-90% Burn Duration vs. EHRD

Figure 15. IMEP vs. EHRD

Phase Two: Real-Time Control

A wide variety of tests were run in this phase to investigate the performance of the fuzzy controller. Each test was performed by selecting a desired value for EHRD and then establishing an initial value for lambda, λ_0 . The fuzzy controller then changes the value of λ in order to keep the EHRD near the desired setpoint. As discussed before, the controller is designed to approach the setpoint from the rich side. This can be seen in all of the tests that were performed in this phase.

The first set of plots are for a desired EHRD of 50 and 60 degrees. The corresponding values for λ_0 were 0.7 and 0.8 and were taken from the knowledge gained in the phase one tests. Three tests are shown for each condition. Figures 16, 17, and 18 show engine speed, mass air flow, and injector pulse width, respectively. These plots are similar to those seen in Figures 6-8 shown previously. Figure 19 shows the desired λ values derived from the fuzzy controller. As expected, the EHRD=50 tests have a lower desired λ than those for the EHRD=60 tests. The effects of engine warming are very evident on this plot. The slopes of the lines are very steep for the first 15-20 seconds and then flatten out for the remainder of the test. This is interesting because the reason why the production strategy has the engine speed slow down at this point is because the initial warm up period is reached by the higher engine speed during the first 15-20 seconds. The plot of EHRD can be found in Figure 20. The controller does a very acceptable job at controlling the ERHD. It is important that the controller performs well at the different engine speeds throughout the tests, as these changing engine speeds change the fuel dynamics as mentioned earlier. The controller is successful in controlling this time-varying, event-based, nonlinear system.

Figure 16. Engine Speed for Phase Two Tests

Figure 17. Mass Air Flow for Phase Two Tests

Figure 18. Fuel Injector Pulse Width for Phase Two Tests

Figure 19. Desired Lambda for Phase Two Tests

Figure 20. EHRD for Phase Two Tests

An interesting observation can be made from the plots of desired lambda in Figure 19. The values of lambda throughout the test are similar to values that could be found in look-up tables common to production, open-loop control strategies. Notice that for any three plots for the same desired EHRD, that the desired lambda varies by as much as 0.07 at times and that one test for EHRD=50 has a lambda trace that is close to an EHRD=60 test during the 15-25 second range. This shows how an open-loop system would be inferior to this closed loop system. Time is saved from mapping desired lambda values for many different operating conditions because one set of fuzzy controller gains work for any range of desired EHRD values. This saves a great deal of time in laboratory testing. Time does not have to be spent to come up with a new desired lambda map for each new desired EHRD value. Also, there would be errors from test to test using the open-loop control system similar to those seen in the phase one tests.

To investigate this further, two tests were run after the conclusion of these cold-start tests. The engine was shut down and not forced cooled. The engine coolant was approximately 85 degrees at the start of the tests but engine surfaces such as intake valves were much greater than what they would be at the beginning of the cold-start tests. The same values for λ_0 and all controller gains and ranges were used for these warm-start tests as were used for the EHRD=50 tests. The results can be seen in Figures 21 and 22 which show the desired lambda and EHRD plots for the two tests. It is evident from these two plots that the controller works as well during these warmer tests as it does during the cold-start tests.

Figure 21. Desired Lambda for Warm-Start Tests

Figure 22. EHRD for Warm-Start Tests

Another important aspect of the controller is the ability to control at the lean limit. The phase one tests showed that poor combustion occurred at EHRD values at or above 65 degrees. Tests were run with the desired EHRD of 64 degrees. The performance of the controller was similar to the tests at the other conditions. The results for the EHRD=64 tests can be viewed in Figures 23 and 24 and a comparison of the three different EHRD setpoint cold-start tests is shown in Table 11. The values for normalized deviation are the percent values found by dividing the standard deviation by the mean for each condition

The results in Table 11 are encouraging since the variability in combustion is known to increase as the air/fuel ratio does. [21] It is important that the controller performs well through a wide range of air/fuel ratios and is not compromised by this variability in combustion. This performance can be evaluated by the normalized deviation, which is very similar for all of the tests shown. The reason for all of the means being slightly lower than the setpoint is by design. When the EHRD is at setpoint, no change is made in the desired lambda. As the engine warms up, more fuel is vaporized and the EHRD decreases. The controller then increases the desired lambda to bring the EHRD and air/fuel ratio higher and the cycle continues. This is why the EHRD switches between "near" setpoint and "shorter than" setpoint.

This switching behavior raises another issue, namely feed forward control. If modeling was investigated for warm-up and fuel dynamics, the effect of temperature change could be compensated for. This could be accomplished by increasing the desired lambda when the EHRD was at setpoint knowing that the air/fuel ratio would become more rich as the engine warmed up. This is a modeling and control problem that is beyond the scope of this work but could be investigated in the future.

Figure 23. Desired Lambda for Desired EHRD=64 Tests

Figure 24. EHRD for Desired EHRD=64 Tests

Table 11. EHRD Data for Cold-Start Tests

Des. EHRD	Std. Dev.	Mean	Norm. Dev.
50	2.91	48.7	5.97
50	2.91	48.8	5.96
50	2.46	48.5	5.09
60	3.52	58.1	6.06
60	3.61	58.1	6.22
60	4.02	58.4	6.89
64	3.82	61.8	6.18
64	3.28	61.9	5.29
64	4.19	61.9	6.77

Additional tests were run to examine the sensitivity of the controllers performance when values of λ_0 were different than what would be considered optimal as well as to investigate the disturbance rejection ability. For the differing λ_0 tests, the desired EHRD was 60 degrees and values of 0.7 and 0.9 were used for λ_0 . This differs from the value of 0.8 that was used for the previous tests. The results for these tests are shown in Figures 25-28. For the $\lambda_0=0.7$ tests, the EHRD begins shorter than desired. The desired lambda is increased which causes the EHRD to rise to the setpoint. For the $\lambda_0=0.9$ tests, the EHRD begins longer than desired. The desired lambda is held relatively constant rather than increased as normal. This causes the EHRD to drop to the setpoint. The response of the latter is faster than the former by design. As mentioned before, it is more desirable to err on the rich side than the lean side during these tests. This is caused by the higher output values for the fuzzy controller for lean conditions than the rich conditions.

The disturbance rejection properties of the control strategy were investigated by using an incorrect value for the INJ_FLOW parameter in equation (4.2). This is used to simulate errors in all of the variables that could be affected by incorrect values. The mass air flow and engine speed could have error due to sensor degradation or other problems and the injector flow rate changes from injector to injector due to manufacturing variability or engine-wide errors due to a faulty fuel pressure regulator. The multiplications in equations (4.1) and (4.2) would be affected in a similar manner by an error in any of the variables mentioned above. An error of +/- 10% in the INJ_FLOW parameter was chosen to run two sets of tests to see the ability of the controller to adjust for these errors. The data for these tests can be seen in Figures 29-32. The response of the -10% error is similar to the response of the $\lambda_0=0.7$ tests because the controller thinks at first that it needs a larger fuel pulse width than it actually does. This causes an initial error on the rich side. There is an equal similarity between the +10% error tests and the λ

$\phi=0.7$ tests because there is an initial error on the lean side because the controller thinks the injector is flowing at a higher rate than it actually is.

The previous tests are related to the problem of individual cylinder air/fuel ratio control. Each cylinder receives slightly different air/flow, has slightly different fuel dynamics during warm-up, and slightly different injector flow rates. With a pressure transducer on each cylinder, it is conceivable that individual cylinder air/fuel ratio could be performed. This is also beyond the scope of this work and could be investigated for future work.

Figure 25. Desired Lambda for Initial Lambda = 0.7 Tests

Figure 26. EHRD for Initial Lambda = 0.7 Tests

Figure 27. Desired Lambda for Initial Lambda = 0.9 Tests

Figure 28. EHRD for Initial Lambda = 0.9 Tests

Figure 29. Desired Lambda for -10% Flow Rate Error Tests

Figure 30. EHRD for -10% Flow Rate Error Tests

Figure 31. Desired Lambda for +10% Flow Rate Error Tests

Figure 32. EHRD for +10% Flow Rate Error Tests

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

It has been shown in this work that it is possible to use combustion pressure feedback to control air/fuel ratio during cold-start. The controller is able to control air/fuel ratio at all levels at which combustion quality would be acceptable. The controller proved to be insensitive to initial errors in λ and was able to compensate for errors caused by sensor aging as well as injector flow rate variability. The controller performed well through a range of gains. Tests were performed with gains ranging across an order of magnitude, in the middle of which were the final gains selected. These gains appear to have the best combination of transient response, disturbance rejection, and overshoot. The optimum choice for gains is a subjective issue at present as there are no guidelines for performance requirements.

Contribution

This controller can be used as a valuable tool for emission studies in the future. The primary goal of this work to develop a method for air/fuel ratio control using combustion pressure feedback during cold-start has been achieved. The ability to control air/fuel ratio is an important aid in comprehensive emission studies during cold-start. This work will facilitate future studies of cold-start emissions.

Recommendations

There are many possible areas for future research in this area. The first involves the alternative method for inferring air/fuel ratio mentioned in this paper, Equivalent Torque Production Duration (ETPD). By using crankshaft acceleration and engine dynamics models, it is possible to estimate the indicated torque and then calculate the ETPD. It was shown in [20] that this value was linearly related to the EHRD. This work would also need a warm-up engine model including friction and temperatures that could be used for feed forward control that was mentioned in the previous section. Individual cylinder air/fuel ratio control could be developed with the addition of combustion pressure sensors on the remaining cylinders. A supervisory strategy could be developed to enable the controller to "learn". This would allow the controller to update controller gains and values for λ_0 over time as the engine goes through changes such as accumulating valve deposits. Finally, different methods such as sliding mode control could be implemented to study the performance of this controller relative to other control methods.

BIBLIOGRAPHY

- [1] Gassenfeit, E.H., and J.D. Powell, "Algorithms for Air-Fuel Ratio Estimation Using Internal Combustion Engine Cylinder Pressure," SAE Paper No. 890300.
- [2] Cook, H.A., O.H. Heinicke, and W.H. Haynie, "Spark-Timing Control Based on Correlation of Maximum-Economy Spark Timing, Flame-Front Travel, and Cylinder-Pressure Rise," NACA TN No. 886, 1946.
- [3] Sugitani et al., "Combustion Pressure Sensor for Toyota Lean Burn Engine Control," SAE Paper No. 930882.
- [4] Grimm, B.M., and R.T. Johnson, "Review of Simple Heat Release Computations," SAE Paper No. 900445.
- [5] Rassweiler, G.M., and L. Withrow, "Motion Pictures of Engine Flames Correlated with Pressure Cards," SAE Paper No. 800131.
- [6] Anderson, M.J., "A Feedback A/F Control System for Low Emission Vehicles," SAE Paper No. 930388.
- [7] Pestana, G.W., "Engine Control Methods Using Combustion Pressure Feedback," SAE Paper No. 890758.
- [8] Powell, J.D., "Engine Control Using Cylinder Pressure: Past, Present, and Future," *Journal of dynamic Systems, Measurement, and Control*, vol. 115, pp. 343-350, June 1993.
- [9] Matekunas, F.A., 1986, "Engine Combustion Control with Ignition Timing by Pressure Ratio Management," U.S. Patent No. 4,622,939, Nov. 18.
- [10] Gilkey, J.G., 1984, "Fuel-Air Ratio Estimation from Cylinder Pressure in an IC Engine," Ph.D. thesis, Stanford University, Stanford, CA, Aug.
- [11] Hogh, G., B.K. Powell, and G.P. Lawson, 1988, "Real Time Engine Dynamic Analysis and Control," SAE paper No. 885104.

- [12] Amann, C.A., "Cylinder-Pressure Measurements and Its Use in Engine Research," SAE Paper No. 852067.
- [13] Heywood, J.B., *Internal Combustion Engine Fundamentals*, McGraw-Hill, 1988.
- [14] Rizzoni, G., "A Dynamic Model for the Internal Combustion Engine," Ph.D. dissertation, Dept. Elec. Eng. Comput. Sci., Univ. Michigan, Ann Arbor, Jan. 1986.
- [15] Hendricks et al., "Nonlinear Transient Fuel Film Compensation (NTFC)," SAE Paper No. 930767.
- [16] Rizzoni, G., "Estimate of Indicated Torque from Crankshaft Speed Fluctuations: A Model for the Dynamics of the IC Engine," *IEEE Trans. Veh. Tech.*, vol. 38, no. 3, pp. 168-179, August 1989.
- [17] Lee, C.C., "Fuzzy Logic in Control Systems: Fuzzy Logic Controller-Part I," *IEEE Trans. Syst. Man Cybern.*, vol. 20, no. 2, pp. 404-418, Mar./Apr. 1990.
- [18] Lee, C.C., "Fuzzy Logic in Control Systems: Fuzzy Logic Controller-Part II," *IEEE Trans. Syst. Man Cybern.*, vol. 20, no. 2, pp. 419-435, Mar./Apr. 1990.
- [19] Li, C.W., and D. Kotys, "Design and Analysis of an IC Engine Fuel Injection Controller," Masters Project Report, Mechanical Engineering Department, The Ohio State University, Columbus, Nov. 1994.
- [20] Leisenring, K.C., G. Rizzoni, and B. Samimy, "Methods for Internal Combustion Engine Feedback Control During Cold-Start," SAE Paper No. 950842.
- [21] Powell, B.K. and G.P. Lawson, "Advanced Real Time Powertrain Systems Analysis," Presented at the Energy-Sources Technology Conference and Exhibition, ASME, Dallas, Texas, Feb. 15-20, 1987.

APPENDIX A

DIGITAL SIGNAL PROCESSING SYSTEM

Overview

The digital signal processing system used in this work consists of three computer boards that reside in a 80486 personal computer, a signal interface panel, and computer programs that run on the DSP system board and the pc itself. The DSP system board has dual access memory that allows data to be read to and from the PC memory as well as communication to take place between the PC and DSP system board via flag registers. The DSP system can be run transparent from the PC and only draw power from the PC if data is not needed to be transferred between the two. If data is to be transferred, two programs need to be written; one compiled and run on the PC and the second which is compiled for the DSP system board and loaded into that board's memory by the PC program. A graphical depiction of this DSP system is shown below.

Figure 33. Digital Signal Processing System

The remainder of this appendix is divided into two sections. The first discusses preliminaries that are needed to be understood in order to use the DSP system. It shows things such as how to sample an input and how to generate an output. It describes the program structure and essentials. The second section describes the programs used for this research.

Preliminaries

There are certain items that are required in programs written to do the event based sampling and real-time DSP that is performed by the programs used in this test. They will be described in this section. The following descriptions are for generic cases and can be used "cookbook" style to create programs. There have been many macro commands that have been setup to make the programming more simple and descriptive. All programs were written in ANSI C. There are two types of listings on the following pages; header files (filename.h) which contain macros and constants, and program listings (filename.c)

that are the actual programs compiled and loaded into the DSP board or run on the PC.

The commands shown below are required to sample a channel.

```

1      adMultSet(ADC_X,ADC_cont_reg);
2      adSample(ADC_X,Mode,ADC_cont_reg);
3      adTrigger(ADC_time_reg);
4      while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
5      data_value_ADC=adRead(ADC_result);
6      voltage=(float)(data_value_ADC>>20)*DataConvertADC;
```

In line 1, the desired multiplexer is set. There are four eight-channel multiplexers on the input board. This command selects which channel on the multiplexers are set to the signal analog-to-digital converter. The value ADC_X is any channel from ADC_0 to ADC_31 as can be seen in the header file input.h on the following pages. The value of ADC_X will cause the multiplexer to send the corresponding channel and the other three channels in the same multiplexer position to the 4-by-1 multiplexer on the input of the a/d converter. All of the registers shown are actually pointers to the addresses of the registers that are initialized in the DSP programs. These registers are ADC_cont_reg, ADC_time_reg, ADC_stat_reg, and ADC_result. Similar register pointers are found throughout these programs. Any macro parameter that is not described can be assumed to be a register pointer. In line 2, the channel selected by ADC_X is sampled. The Mode variable can either be TRACK or HOLD. TRACK causes the values at the input of the 4-by-1 multiplexer to follow the analog signals input to them. HOLD will cause the multiplexer to hold of the input signals at their present state. This enables simultaneous sampling of up to four channels even though the channels must be sampled sequentially by the single a/d converter. Line 3 generates a software trigger and Line 4 waits for the a/d converter to finish its conversion. Line 5 reads the integer result from the a/d converter result register and stores it into the upper 12 bits of the 32 bit integer data_value_ADC. Line 6 converts this integer into a floating point value named voltage. DataConvertADC is

shown in the header files and is the conversion for volts per a/d converter count. The following lines show how to generate a voltage on the output board.

```

7    daSetChannel(DAC_X,DAC_cont_reg);
8    daWrite(((int)(volt*DataConvertDAC))<<20,DAC_input_reg);
9    daRead(dummy_value,DAC_input_reg);

```

DAC_X is the channel that the voltage is to be generated on and can range between DAC_0 to DAC_15 as can be seen in output.h on the following pages. Line 7 sets this channel as the channel to be written to. Line 8 causes the correct integer value to be written to the output board's input register to generate the voltage given by the variable volt. Line 9 causes a software trigger that causes the voltage of volt to be written on channel DAC_X.

Communication between the PC and DSP system board is performed by communication flags that are setup as shown below. A .bss memory section is set aside for these functions on the DSP system board. An integer flag can be setup as shown below in lines 10-12 where a pointer to a floating point variable or array can be setup as shown in line 13-15. The values for commX can be from comm0 to comm9. An example of these can be seen in the program listing fuzzy.c shown later in this appendix. Lines 10,11,13,14 are in the variable declaration section and lines 12 and 15 are in the main program listing. The communication flag Integer_flag and floating point variable Data can have any name.

```

10    extern long CommX;
11    long *Integer_flag
12    Integer_flag=&CommX;

13    extern float *CommX;
14    float Data;
15    Comm5=Data;

```


Once these locations are setup, they can be written to as variables on the DSP board by the program resident on the system board. Special functions exist to read and write to these memory locations from the PC. These can be found in the documentation from Spectrum Signal Processing. Lines 16-18 show the procedure for downloading a program onto the DSP system board. The #define statement must be in the variable declaration section and lines 17 and 18 are found in the main program. The file filename.out is created by compiling the filename.c program listing using the compiler for the TMS320C30 supplied by Texas Instruments. There are several other functions that are needed for successful operation of the DSP system. These are explained in the documentation from Spectrum Signal Processing and a concerted effort has been made to comment the following programs as to make the code very explanatory.

```
16    #define BOARDADR    0x290
17    SelectBoard(BOARDADR);
18    loadstat=coffLoad("filename.out");
```

```

/*****
/* INPUT.H : Header file for A/D input board */
*****/

/* define A/D input board base address and register locations */

#define adConvert 0x03
#define ADC_BASE 0x800006
#define ADC_CONT_REG ADC_BASE
#define ADC_STAT_REG ADC_BASE
#define ADC_TIME_REG ADC_BASE + 0x1
#define ADC_RESULT ADC_BASE + 0x1

/* define macros for reading A/D channels */

#define ADC_BUSY 0
#define ADC_CONVERSION_DONE 1
#define ADC_EOC_FLAG_SET(a) (*a & 0x00800000)
#define ADC_HOLD 0xf0f0f0
#define ADC_CALIBRATION 0x00200000
#define ADC_CALIBRATION_MASK 0x00200000
#define HOLD 0x00
#define TRACK 0x400000
#define adTrigger(x) (*x = ADC_HOLD)
#define adSample(x,y,z) (*z = (x + y))
#define adMultSet(x,y) (*y = (x + TRACK))
#define adRead(x) (*x)

/* define each channel write */

#define ADC_0 0x00000000
#define ADC_1 0x00100000
#define ADC_2 0x00200000
#define ADC_3 0x00300000
#define ADC_4 0x00400000
#define ADC_5 0x00500000
#define ADC_6 0x00600000
#define ADC_7 0x00700000
#define ADC_8 0x00800000
#define ADC_9 0x00900000
#define ADC_10 0x00a00000
#define ADC_11 0x00b00000
#define ADC_12 0x00c00000
#define ADC_13 0x00d00000
#define ADC_14 0x00e00000
#define ADC_15 0x00f00000
#define ADC_16 0x01000000
#define ADC_17 0x01100000
#define ADC_18 0x01200000
#define ADC_19 0x01300000
#define ADC_20 0x01400000
#define ADC_21 0x01500000
#define ADC_22 0x01600000

```

```

/*****
/* OUTPUT.H : Header file for D/A output board */
*****/

/* define D/A output board base address and register locations*/

#define DAC_BASE 0x800008
#define DAC_CONT_REG DAC_BASE
#define DAC_INPUT_REG DAC_BASE + 0x1

/* define output statements */

#define daWrite(x,y) (*y = (x))
#define daSetChannel(x,y) (*y = x)
#define daRead(x,y) (x = (*y))

/* define each channel write */
/* note: nonzero bit (4,5,6, or 7) to enable software load control */

#define DAC_0 0x00100000
#define DAC_1 0x00110000
#define DAC_2 0x00120000
#define DAC_3 0x00130000
#define DAC_4 0x00240000
#define DAC_5 0x00250000
#define DAC_6 0x00260000
#define DAC_7 0x00270000
#define DAC_8 0x00480000
#define DAC_9 0x00490000
#define DAC_10 0x004a0000
#define DAC_11 0x004b0000
#define DAC_12 0x008c0000
#define DAC_13 0x008d0000
#define DAC_14 0x008e0000
#define DAC_15 0x008f0000

/* define zero offsets for output channels */

#define DRIFT_COMP_0 .062
#define DRIFT_COMP_8 -.032

```

Program Listings for Open-Loop and Closed-Loop Experiments

The programs `efic30.c` and `efitest.c` were used to run the open loop tests. The command to type to run the open-loop tests was "efitest" which runs the executable file `efitest.exe` on the PC. The parameters for the program were `EFI_ACTIVE` and `DES_LAMBDA` found in the header file `efidata.h` which is shown on the following pages. `EFI_ACTIVE` could be either YES or NO which controls whether the relays are fired to switch control to the EFI console. The `DES_LAMBDA` is the desired normalized air/fuel ratio to run open-loop during the test. The program `eficalc.c` processes the degree sampled data and writes the values to separate files. The raw data is saved in binary format in the `testname.bin` file. The once-per-cylinder averaged data is saved in the `testname.tdc` file as ascii data. The degree data for pressure related computations is written in ascii format to the `testname.tmp` file. The combustion data for each combustion event is then written in ascii format to the `testname.mep` file. There are a great deal of header files that are called by these programs and an effort was made to do in-line documentation. These programs are not listed here due to space concerns.

The programs that ran the closed-loop tests were `fuzzy.c` and `runfuzzy.c` and are shown in this appendix. The parameters for these programs are `FUZZY_ACTIVE`, `EFI_ACTIVE`, `DES_LAMBDA`, `DES_EHRD`, `PRINT_UPDATE`, `UPDATE_INTERVAL` and are found in the `fuzzdata.h` header file. `FUZZY_ACTIVE` and `EFI_ACTIVE` can either be YES or NO. The latter serves the same function as mentioned above and `FUZZY_ACTIVE` decides whether the program runs open-loop or closed-loop. If degree data is not required, this program can run all of the desired tests by changing these parameters. `DES_LAMBDA` now controls the initial value for λ . `DES_EHRD` is the setpoint for the EHRD throughout the tests. `PRINT_UPDATE` is either YES or NO and decides whether the averaged data is output to the PC screen. The integer value `UPDATE_INTERVAL` controls how often the values are written to the

screen. The data for the tests using these programs are stored in ascii format in the file `testname.fuz`. This explanation is not intended to encompass all that is needed to know about this DSP system in order for someone to start modifying programs. It should however explain the programs enough for someone to use these programs. For program modification, it is understood that a fair amount of time would have to be spent to read the documentation for the Spectrum Signal Processing system. If an individual has a strong background in C programming, the development or modification of programs is very simple. Everything in these programs could be used in "cookbook" fashion as mentioned earlier.

```

/*
FUZZY.C - DSP Data Acquisition
           Works in conjunction with RUNFUZZY.EXE
*/
asm (" .length 58");
asm (" .width 120");

#include <math.h>
#include <input.h>
#include <output.h>
#include <kcfuzzy.h>
#include <fuzzdata.h>
#include <engspecs.h>

/* Set up interrupt */

asm(" .sect \" .int02\");
asm(" .word _c_int02");
asm(" .text");

/* define global variables ( these go in the .bss memory section ) */

extern long Comm0,Comm1,Comm2,Comm3,Comm4;
extern float *Comm5,*Comm6;

long *PCproceedflag,*DSPproceedflag,*DATAAcquitflag,*Buffreadflag,*Datareadflag;

/* define necessary global variables */

unsigned int
*ADC_cont_reg,*ADC_stat_reg,*ADC_time_reg,*ADC_result,*DAC_cont_reg,*DAC_input_reg;

int output_value=0x0;int buffer_index=1;int deg,dummy_value,data_value_ADC;
double data_value,pw_volt;
float maftemp,mafvoltage,pressure,sync_voltage,rpm,fuel_needed,ect,ecttemp,lambda,rpmtemp;
float des_pw,actual_pw,pwtemp,pw_ave,eprd,eprd_sum,eprd_ave;float maf_ave,ect_ave,
float time=0.0;float rpm_sum,rpm_ave,pw_sum,offset,Momdelta,Sumdelta;
float maf_sum=0.0;float rpm_sum=0.0;float ect_sum=0.0;float eprd_sum=0.0;
int samples=0;int buffer_samples=0;int stop_dbuff=0;int channel_count=0;
float buffer[ARRAY_SIZE],data[DATA_COLUMNS];
int turn_off_efi=NO;int efi_status=OFF;int Synced_Up=NO;int Initialized=NO;int Sync=NO;
int TDC_found=YES;int quit_buffer_updating=NO;
float V[SAMPLES_PER_CYCLE],dV[SAMPLES_PER_CYCLE];
double OdRadians,coef,g,e;
float dP,last_pressure,dQ,tempdata;float MomQ=0.0;float SumQ=0.0;
int i,j,k,OdCy,w,x,y;int writes_to_pc=0;int index=0;int cycles=0;
float
Ehrd_Error[NUM_EHRD_LEVELS][NUM_EHRD_STATES],Ehrd_States[NUM_EHRD_STATES];
float Time_Since_Start[NUM_TIME_LEVELS][NUM_TIME_STATES]=TIME_ARRAY;
float Time_States[NUM_TIME_STATES];
float Rpm_Error[NUM_RPM_LEVELS][NUM_RPM_STATES]=RPM_ARRAY;
Rpm_States[NUM_RPM_STATES];

```

```

float ehrrd_min[NUM_EHRD_LEVELS]=MIN_EHRD;float
ehrrd_max[NUM_EHRD_LEVELS]=MAX_EHRD;
float time_min[NUM_TIME_LEVELS]=MIN_TIME;float
time_max[NUM_TIME_LEVELS]=MAX_TIME;
float rpm_min[NUM_RPM_LEVELS]=MIN_RPM;float rpm_max[NUM_RPM_LEVELS]=MAX_RPM;
float Rules_Alpha[NUM_RPM_STATES][NUM_TIME_STATES][NUM_EHRD_STATES];
float Rules_Mu[NUM_RPM_STATES][NUM_TIME_STATES][NUM_EHRD_STATES];
float Mu_Rpm_0[NUM_TIME_STATES][NUM_EHRD_STATES]=MU_RPM_ERROR_0;
float Mu_Rpm_1[NUM_TIME_STATES][NUM_EHRD_STATES]=MU_RPM_ERROR_1;
float Ehrrd_Column0[NUM_EHRD_LEVELS]=EHRD_COLUMN_0;
float Ehrrd_Column1[NUM_EHRD_LEVELS]=EHRD_COLUMN_1;
float Ehrrd_Column2[NUM_EHRD_LEVELS]=EHRD_COLUMN_2;
float Ehrrd_Column3[NUM_EHRD_LEVELS]=EHRD_COLUMN_3;
float Ehrrd_Column4[NUM_EHRD_LEVELS]=EHRD_COLUMN_4;
float Ehrrd_Column5[NUM_EHRD_LEVELS]=EHRD_COLUMN_5;
float Ehrrd_Column6[NUM_EHRD_LEVELS]=EHRD_COLUMN_6;
int level,state,ehrrd_level,time_level,rpm_level;int test_over=NO;
float sum_mu_alpha=0.0;float delta_mu_alpha=0.0;float sum_alpha=0.0;float delta_alpha=0.0;
float delta_lambda,feedback_error,rpm_error,desired_rpm;

main()
{
    ADC_cont_reg=(unsigned int*)ADC_CONT_REG;
    ADC_stat_reg=(unsigned int*)ADC_STAT_REG;
    ADC_time_reg=(unsigned int*)ADC_TIME_REG;
    ADC_result=(unsigned int*)ADC_RESULT;
    DAC_cont_reg=(unsigned int*)DAC_CONT_REG;
    DAC_input_reg=(unsigned int*)DAC_INPUT_REG;

    *ADC_cont_reg=0;
    *ADC_cont_reg=ADC_CALIBRATION;
    *ADC_cont_reg=0;
    while((*ADC_stat_reg&ADC_CALIBRATION_MASK)==0);

    output_value=adRead(ADC_result); /* Reset EOC Flag */

    PCproceedflag=&Comm0;
    DSPproceedflag=&Comm1;
    Buffreadflag=&Comm2;
    Datareadflag=&Comm3;
    DATAacquitflag=&Comm4;

    Comm5=buffer;
    Comm6=data;

    /* Reset EFI Relay and MAP Voltage Signal */

    data_value=RESET+DRIFT_COMP_8;
    daSetChannel(DAC_8,DAC_cont_reg);
    daWrite(((int)(data_value*DataConvertDAC))<<20,DAC_input_reg);
    daRead(dummy_value,DAC_input_reg);

    pw_volt=DRIFT_COMP_0+RESET;

```

```

daSetChannel(DAC_0,DAC_cont_reg);
daWrite(((int)(pw_volt*DataConvertDAC))<<20,DAC_input_reg);
daRead(dummy_value.DAC_input_reg);

for(i=0;i<=720;i+=RES)
{
    OdCy=i;
    OdRadians=OdCy*PIE/180.0;
    coef=sqrt(1.0-pow((Ratio*sin(OdRadians)),2));
    g=0.5*sin(OdRadians)*(1+(Ratio*cos(OdRadians))/(coef));
    e=0.5*((1.0-cos(OdRadians))+(1.0-coef)/Ratio);
    dV[i/RES]=Stroke*g*Ap*PIE/180.0;
    V[i/RES]=(1/(CompRatio-1.0)+e)*Stroke*Ap;
}

state=0;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column0[k];
state=1;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column1[k];
state=2;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column2[k];
state=3;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column3[k];
state=4;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column4[k];
state=5;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column5[k];
state=6;
for(k=0;k<NUM_EHRD_LEVELS;k++)
    Ehrd_Error[k][state]=Ehrd_Column6[k];

w=0;
for(x=0;x<NUM_TIME_STATES;x++)
{
    for(y=0;y<NUM_EHRD_STATES;y++)
        Rules_Mu[w][x][y]=Mu_Rpm_0[x][y];
}
w=1;
for(x=0;x<NUM_TIME_STATES;x++)
{
    for(y=0;y<NUM_EHRD_STATES;y++)
        Rules_Mu[w][x][y]=Mu_Rpm_1[x][y];
}

*PCproceedflag=1; /* Tell PC that all pointers are initialized */

while(*DSPproceedflag==0)    /* Wait for PC to get ready */

```



```

;

*DSPproceedflag=0;    /* Reset flag */

    /* enable interrupt */

asm(" OR    2h, IE");
asm(" OR    2000h, ST");

while(1);

} /* End of main() */

/* Listing of program that runs every trigger event */

void c_int02(void)
{
    if (Synced_Up==NO)
    {
        adMultSet(ADC_PRESS,ADC_cont_reg);
        adSample(ADC_PRESS,TRACK,ADC_cont_reg);
        adTrigger(ADC_time_reg);
        while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
        data_value_ADC=adRead(ADC_result);
        voltage=(float)(data_value_ADC>>20)*DataConvertADC;

        adMultSet(ADC_SYNC,ADC_cont_reg);
        adSample(ADC_SYNC,TRACK,ADC_cont_reg);
        adTrigger(ADC_time_reg);
        while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
        data_value_ADC=adRead(ADC_result);
        sync_voltage=(float)(data_value_ADC>>20)*DataConvertADC;

        if (voltage>=PRESS_PK && sync_voltage>=SYNC_HI)
        {
            Synced_Up=YES;
            deg=TDC_CYL_ONE;
            *Datareadflag=1;
        }
    }

    else if (Initialized==NO) /* Synced_Up==YES */
    {
        deg+=RES;
        if (deg==DEG_REF_PRESS)
        {
            adMultSet(ADC_PRESS,ADC_cont_reg);
            adSample(ADC_PRESS,TRACK,ADC_cont_reg);
            adTrigger(ADC_time_reg);
            while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
            data_value_ADC=adRead(ADC_result);
            voltage=(float)(data_value_ADC>>20)*DataConvertADC;

```

```

        offset=(voltage*BAR_PER_VOLT)-BAR_PER_ATM;
        lambda=DES_LAMBDA;
        Initialized=YES;
        *Datareadflag=1;
    }
}

else if (deg==DEG_REF_PRESS)
{
    deg+=RES;
    adMultSet(ADC_PRESS,ADC_cont_reg);
    adSample(ADC_PRESS,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);
    while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
    data_value_ADC=adRead(ADC_result);
    voltage=(float)(data_value_ADC>>20)*DataConvertADC;
    offset=(voltage*BAR_PER_VOLT)-BAR_PER_ATM;
}

else if (deg>DEG_REF_PRESS && deg<=DEG_NEW_CYLINDER)
{
    deg+=RES;
    if (deg==DEG_END_OF_CYLINDER)
    {
        deg=DEG_START_OF_CYLINDER;
        TDC_found=NO;
    }
}

else if (deg>=CHECK_FOR_TDC && TDC_found==NO)
{
    deg+=RES;
    adMultSet(ADC_PRESS,ADC_cont_reg);
    adSample(ADC_PRESS,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);
    while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
    data_value_ADC=adRead(ADC_result);
    voltage=(float)(data_value_ADC>>20)*DataConvertADC;

    adMultSet(ADC_SYNC,ADC_cont_reg);
    adSample(ADC_SYNC,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);
    while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
    data_value_ADC=adRead(ADC_result);
    sync_voltage=(float)(data_value_ADC>>20)*DataConvertADC;

    if (voltage>=PRESS_PK && sync_voltage>=SYNC_HI)
    {
        TDC_found=YES;
        deg=TDC_CYL_ONE;
        cycles+=1;
        last_pressure=NM_PER_BAR*((voltage*BAR_PER_VOLT)-offset);
        tempdata=last_pressure;
    }
}

else if (TDC_found==YES && deg>=TDC_CYL_ONE && deg<=EXHAUST_OPEN)
{
    deg+=RES;
    adMultSet(ADC_PRESS,ADC_cont_reg);

```

```

adSample(ADC_PRESS,TRACK,ADC_cont_reg);
adTrigger(ADC_time_reg);
while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
data_value_ADC=adRead(ADC_result);
voltage=(float)(data_value_ADC>>20)*DataConvertADC;

pressure=NM_PER_BAR*((voltage*BAR_PER_VOLT)-offset);
dP=(pressure-last_pressure)/RES;
last_pressure=pressure;
dQ=((1/(GAMMA-1))*V[deg/RES]*dP)+((GAMMA/(GAMMA-
1))*pressure*dV[deg/RES]);

    if (dQ>=0.0)
    {
        Momdelta=((deg-SPARK_DEGREE)*dQ*RES);
        MomQ+=Momdelta;
        Sumdelta=(dQ*RES);
        SumQ+=Sumdelta;
    }

else if (deg==DEG_CALC_EHRD)
{
    deg+=RES;
    ehrd=MomQ/SumQ;
    MomQ=0.0;
    SumQ=0.0;
    ehrd_sum+=(ehrd);
    if (cycles==NUM_AVE)
    {
        ehrd_ave=ehrd_sum/NUM_AVE;
        ehrd_sum=0.0;
        data[7]=ehrd_ave;
    }
}

else if (deg==DEG_READ_ECT)
{
    deg+=RES;
    adMultSet(ADC_ECT,ADC_cont_reg);
    adSample(ADC_ECT,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);
    while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
    data_value_ADC=adRead(ADC_result);
    ecttemp=(float)(data_value_ADC>>20)*DataConvertADC;

    ect=(ECT_OFFSET)+(ECT_SLOPE*ecttemp*2.0);
    ect_sum+=ect;
    if (cycles==NUM_AVE)
    {
        ect_ave=ect_sum/NUM_AVE;
        ect_sum=0.0;
        data[1]=ect_ave;
    }
}

else if (deg==DEG_READ_MAF)
{
    deg+=RES;
    adMultSet(ADC_MAF,ADC_cont_reg);
    adSample(ADC_MAF,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);

```

```

while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
data_value_ADC=adRead(ADC_result);
maftemp=(float)(data_value_ADC>>20)*DataConvertADC;

maf=(MAF_A0)+(MAF_A1*maftemp)+(MAF_A2*(maftemp*maftemp));
maf_sum+=maf;
if (cycles==(NUM_AVE-1))
{
    maf_ave=maf_sum/NUM_AVE;
    maf_sum=0.0;
    data[2]=maf_ave;
}

}

else if (deg==DEG_READ_RPM)
{
    deg+=RES;
    adMultSet(ADC_RPM,ADC_cont_reg);
    adSample(ADC_RPM,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);
    while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
    data_value_ADC=adRead(ADC_result);
    rpmtemp=(float)(data_value_ADC>>20)*DataConvertADC;

    rpm=RPM_GAIN*(rpmtemp-RPM_BIAS);
    rpm_sum+=rpm;
    if (cycles==(NUM_AVE-1))
    {
        rpm_ave=rpm_sum/NUM_AVE;
        rpm_sum=0.0;
        data[3]=rpm_ave;
    }

}

else if (deg==DEG_READ_PW)
{
    deg+=RES;
    adMultSet(ADC_PW,ADC_cont_reg);
    adSample(ADC_PW,TRACK,ADC_cont_reg);
    adTrigger(ADC_time_reg);
    while(ADC_EOC_FLAG_SET(ADC_stat_reg)==0);
    data_value_ADC=adRead(ADC_result);
    pwtemp=(float)(data_value_ADC>>20)*DataConvertADC;

    actual_pw=(pwtemp-EFI_PW_OFFSET)/EFI_PW_SLOPE;
    pw_sum+=actual_pw;
    if (cycles==(NUM_AVE-1))
    {
        pw_ave=pw_sum/NUM_AVE;
        pw_sum=0.0;
        data[6]=pw_ave;
    }

}

else if (deg==DEG_CALC_TIME)
{
    deg+=RES;
    if (rpm>RPM_MIN)
        time+=((DEG_PER_CYCLE)/(rpm*RPM_SEC_PER_DEG));
    data[0]=time;
    if (time<TIME_RPM_CHANGE)
        desired_rpm=RPM_COLD_DESIRE;
}

```

```

else
    desired_rpm=RPM_WARM_DESIRED;
}

else if (deg==DEG_FUZZIFY_INPUTS && cycles==NUM_AVE)
{
    deg+=RES;
    feedback_error=DES_EHRD-ehrd_ave;
    for(y=0;y<NUM_EHRD_LEVELS;y++)
    {
        if (feedback_error>ehrd_min[y] && feedback_error<=ehrd_max[y])
        {
            ehrd_level=y;
            data[8]=ehrd_level;
            for(state=0;state<NUM_EHRD_STATES;state++)
                Ehrd_States[state]=Ehrd_Error[ehrd_level][state];
        }
    }
    for(y=0;y<NUM_TIME_LEVELS;y++)
    {
        if (time>time_min[y] && time<=time_max[y])
        {
            time_level=y;
            data[9]=time_level;
            for(state=0;state<NUM_TIME_STATES;state++)
                Time_States[state]=Time_Since_Start[time_level][state];
        }
    }

    for(y=0;y<NUM_RPM_LEVELS;y++)
    {
        rpm_error=desired_rpm-rpm_ave;
        if (rpm_error>rpm_min[y] && rpm_error<=rpm_max[y])
        {
            rpm_level=y;
            data[10]=rpm_level;
            for(state=0;state<NUM_RPM_STATES;state++)
                Rpm_States[state]=Rpm_Error[rpm_level][state];
        }
    }
}

else if (deg==DEG_FUZZY_RULES && cycles==NUM_AVE)
{
    deg+=RES;
    for(w=0;w<NUM_RPM_STATES;w++)
        for(x=0;x<NUM_TIME_STATES;x++)
            for(y=0;y<NUM_EHRD_STATES;y++)
                Rules_Alpha[w][x][y]=Rpm_States[w]*Time_States[x]*E
hrd_States[y];
}

else if (deg==DEG_FUZZIFY_OUTPUTS && cycles==NUM_AVE)
{
    deg+=RES;
    for(w=0;w<NUM_RPM_STATES;w++)
        for(x=0;x<NUM_TIME_STATES;x++)
            for(y=0;y<NUM_EHRD_STATES;y++)
            {
                delta_mu_alpha=Rules_Alpha[w][x][y]*Rules_Mu[w][x][y];
                delta_alpha=Rules_Alpha[w][x][y];
                sum_mu_alpha+=delta_mu_alpha;
                sum_alpha+=delta_alpha;
            }
}

```

```

else if (deg==DEG_CALC_LAMBDA && cycles==NUM_AVE)
{
    deg+=RES;
    if (FUZZY_ACTIVE==YES)
    {
        delta_lambda=sum_mu_alpha/sum_alpha;
        sum_mu_alpha=0.0;
        sum_alpha=0.0;
        lambda+=delta_lambda;
        data[4]=lambda;
        if (lambda>LAMBDA_MAX)
            lambda=LAMBDA_MAX;
        if(lambda<LAMBDA_MIN)
            lambda=LAMBDA_MIN;
        if(ehrd_ave>EHRD_MAX || ehrd_ave<EHRD_MIN)
            lambda=LAMBDA_DEF;
    }
    else
    {
        lambda=DES_LAMBDA;
        data[4]=lambda;
    }
}

else if (deg==DEG_CALC_PW && cycles==NUM_AVE)
{
    deg+=RES;
    fuel_needed=(maf*SEC_PER_MIN)/(rpm*STOICH*lambda*CYL_PER_REV);
    des_pw=(MS_CONVERT*(fuel_needed/SLOPE))+PW_OFF;
    data[5]=des_pw;

    pw_volt=DRIFT_COMP_0+(des_pw-EFI_OFFSET)/EFI_SLOPE;
    daSetChannel(DAC_0,DAC_cont_reg);
    daWrite(((int)(pw_volt*DataConvertDAC))<<20,DAC_input_reg);
    daRead(dummy_value,DAC_input_reg);

    if (efi_status==OFF && EFI_ACTIVE==YES && cycles==NUM_AVE)
    {
        data_value=5.0+DRIFT_COMP_8;
        daSetChannel(DAC_8,DAC_cont_reg);
        daWrite(((int)(data_value*DataConvertDAC))<<20,DAC_input_reg);
        daRead(dummy_value,DAC_input_reg);
        efi_status==ON;
    }
}

else if (deg==DEG_START_PW && test_over==NO && efi_status==OFF &&
EFI_ACTIVE==YES && cycles==NUM_CYCLES_PW)
{
    deg+=RES;
    fuel_needed=(maf*SEC_PER_MIN)/(rpm*STOICH*lambda*CYL_PER_REV);
    des_pw=(MS_CONVERT*(fuel_needed/SLOPE))+PW_OFF;
    data[5]=des_pw;

    pw_volt=DRIFT_COMP_0+(des_pw-EFI_OFFSET)/EFI_SLOPE;
    daSetChannel(DAC_0,DAC_cont_reg);
    daWrite(((int)(pw_volt*DataConvertDAC))<<20,DAC_input_reg);
    daRead(dummy_value,DAC_input_reg);

```

```

        data_value=5.0+DRIFT_COMP_8;
        daSetChannel(DAC_8,DAC_cont_reg);
        daWrite(((int)(data_value*DataConvertDAC))<<20,DAC_input_reg);
        daRead(dummy_value,DAC_input_reg);
        efi_status==ON;
    }

    else if (deg==DEG_BUFFER_DATA && quit_buffer_updating==NO)
    {
        deg+=RES;
        if (cycles==NUM_AVE)
        {
            for(j=0;j<DATA_COLUMNS;j++)
            {
                buffer[index]=data[j];
                index+=1;
            }
        }
    }

    else if (deg==DEG_DATA_TO_PC && cycles==NUM_AVE)
    {
        deg+=RES;
        *Datareadflag=1;
        writes_to_pc+=1;
        cycles=0;
        if (writes_to_pc>=SEC_APPROX)
        {
            *DATAacquitflag=YES;
            quit_buffer_updating=YES;
            test_over=YES;
            data_value=RESET+DRIFT_COMP_8;
            daSetChannel(DAC_8,DAC_cont_reg);
            daWrite(((int)(data_value*DataConvertDAC))<<20,DAC_input_reg);
            daRead(dummy_value,DAC_input_reg);

            pw_volt=DRIFT_COMP_0+RESET;
            daSetChannel(DAC_0,DAC_cont_reg);
            daWrite(((int)(pw_volt*DataConvertDAC))<<20,DAC_input_reg);
            daRead(dummy_value,DAC_input_reg);
        }
    }

    else
        deg+=RES;
} /* End of int02 */

```

```

/*
RUNFUZZY.C - This is the PC program that runs the fuzzy controller.
               This is designed to run with the EFI controller closed-loop.
               This program calls the DSP program FUZZY.OUT*/

#include "tms30.h"
#include <stdio.h>
#include <stdlib.h>
#include <kcfuzzy.h>
#include <fuzzdata.h>

#define BOARDADR    0x290 /* Factory default I/O Address */
#define COMM0    0x30000 /* Start of .bss memory area */
#define PC_PROCEED_FLAG    COMM0+0 /* Absolute memory locations */
#define DSP_PROCEED_FLAG    COMM0+1 /* reserved in LSICMAP.CMD */
#define BUFFER_READ_FLAG    COMM0+2
#define DATA_READ_FLAG    COMM0+3
#define DATA_ACQ_QUIT_FLAG    COMM0+4
#define BUFFER    COMM0+5
#define DATA_COMM    COMM0+6
#define MAXCOUNT 80000
#define MAXCNT 20

void main(void)
{
    unsigned short loadstat;
    int iterations=0,int update_count=0;int i,j,k,l,index,dot_location;
    long num_of_bytes=0;long a;
    int offset1,offset2,bytes_written;
    unsigned long buffloc,dataloc;
    DATATYPE buffer[ARRAY_SIZE],data[DATA_COLUMNS];DATATYPE *ptr1,*ptr2;
    FILE *outfile,*headfile;
    char binfile[11],headerfile[11],datatype[8],dummy[8];

    /* Prompt user for output filename */

    printf("\n\nType output filename (Must be mmmddyyx.fuz) : ");
    gets( binfile);
    for(index=0;index<=12;index++)
        if (binfile[index]=='.')
            dot_location=index;

    for(index=0;index<=dot_location;index++)
        headerfile[index]=binfile[index];
        headerfile[dot_location+1]='h';
        headerfile[dot_location+2]='e';
        headerfile[dot_location+3]='d';
        for(index=dot_location+4;index<=12;index++)
            headerfile[index]=binfile[index];

    /* Initialize Board */

    SelectBoard(BOARDADR);
    loadstat=coffLoad("fuzzy.out");

```



```

if (loadstat!=0)
{
    printf("\n\nError During Program Load!!!!\n");
    printf("coeffLoad() returned %x\n",loadstat);
    exit(0);
}

/* Initialize Flags */

Put32Bit(PC_PROCEED_FLAG,DUAL,0x0L);Put32Bit(DSP_PROCEED_FLAG,DUAL,0x0L);
Put32Bit(BUFFER_READ_FLAG,DUAL,0x0L);Put32Bit(DATA_READ_FLAG,DUAL,0x0L);
Put32Bit(DATA_ACQ_QUIT_FLAG,DUAL,0x0L);

outfile=fopen( binfile,"wb");
if (outfile==NULL)
{
    printf("\n\nUnable to open binary file. Program terminated.\n\n");
    exit(0);
}
headfile=fopen( headerfile,"w");
if (headfile==NULL)
{
    printf("\n\nUnable to open header file. Program terminated.\n\n");
    exit(0);
}

fprintf( headfile, "Filename : %s", binfile);
fprintf( headfile, "\nApproximate Time (sec) : %d ",SEC_APPROX);
fprintf( headfile, "\nTotal Cylinders : %d ",TEST_CYL_LENGTH);
fprintf( headfile, "\nNumber of Data Columns : %d ",DATA_COLUMNS);
fprintf( headfile, "\nFuzzy Controller Active : %d ",FUZZY_ACTIVE);
fprintf( headfile, "\nEFI Controller Active : %d ",EFI_ACTIVE);
fprintf( headfile, "\nDesired EHRD : %3.1lf\n",DES_EHRD);

printf("\n\nFilename : %s", binfile);
printf("\nApproximate Time (sec) : %d ",SEC_APPROX);
printf("\nTotal Cylinders : %d ",TEST_CYL_LENGTH);
printf("\nNumber of Data Columns : %d ",DATA_COLUMNS);
if (FUZZY_ACTIVE==NO)
    printf("\nFuzzy Controller Active : NO ");
else if (FUZZY_ACTIVE==YES)
    printf("\nFuzzy Controller Active : YES ");
else
{
    printf("\nFuzzy Controller Active : INVALID!!! ");
    exit(0);
}
if (EFI_ACTIVE==NO)
    printf("\nEFI Controller Active : NO ");
else if (EFI_ACTIVE==YES)
    printf("\nEFI Controller Active : YES ");
else
{
    printf("\nEFI Controller Active : INVALID!!! ");
    exit(0);
}
printf("\nDesired EHRD : %3.1lf\n",DES_EHRD);

```

```

/* Start the DSP Program Running */

Reset();

/* Wait till the DSP has initialized the absolute locations that contain */
/* the pointers to the buffer arrays. */

for(a=0;a<MAXCOUNT && (Get32Bit(PC_PROCEED_FLAG,DUAL)!=0x1L);a++);
if (a==MAXCOUNT)
{   printf("\n\nTimeout waiting for array pointers!!!\n");
    exit(0);
}

Put32Bit(PC_PROCEED_FLAG,DUAL,0x0L); /* Reset Flag */

/* Get actual starting addresses of arrays */

buffloc=Get32Bit(BUFFER,DUAL);
dataloc=Get32Bit(DATA_COMM,DUAL);

/* Begin Double-Buffered Data Acquisition */

printf("\n\nPress ENTER to Begin Data Acquisition.\n");
gets( dummy);
printf("Data Acquisition Underway.\n");

/* Tell DSP that it is OK two start */

Put32Bit(DSP_PROCEED_FLAG,DUAL,0x1L);

while(Get32Bit(DATA_READ_FLAG,DUAL)==0x0L)
;
printf("\n\nSynced UP!!");
Put32Bit(DATA_READ_FLAG,DUAL,0x0L);

while(Get32Bit(DATA_READ_FLAG,DUAL)==0x0L)
;
printf("\n\nInitialized!!\n\n");
Put32Bit(DATA_READ_FLAG,DUAL,0x0L);

while((Get32Bit(DATA_ACQ_QUIT_FLAG,DUAL)!=0x1L)&&(iterations<SEC_APPROX))
{   /* Check if data is ready to copy */
    while(Get32Bit(DATA_READ_FLAG,DUAL)==0x0L)
    ;
    /* Copy Data */
    RdBlkFlt(dataloc,DUAL,DATA_COLUMNS,data);
    Put32Bit(DATA_READ_FLAG,DUAL,0x0L);
    iterations+=1;
    update_count+=1;

    if (PRINT_UPDATE==YES && update_count==UPDATE_INTERVAL)
    {   update_count=0;
        printf("\n\nTime : %3.1lf sec.\t\tEHRD Average : %3.1lf

```

```

degrees",data[0],data[7]);
    printf("\nDesired Lambda : %3.2lf\tDesired PW : %2.1lf ms",data[4],data[5]);
    printf("\nEngine Speed : %4.0lf rpm\tMAF : %lf lb/s",data[3],data[2]);
}
} /* End of While Loop */

RdBlkFlt(buffloc,DUAL,ARRAY_SIZE,buffer);
printf("\n\nProgram Finished!!");
printf("\n\nIterations = %d",iterations);
for(index=0;index<(DATA_COLUMNS*SEC_APPROX);index+=DATA_COLUMNS)
{
    fprintf(outfile,"%3.1lf", buffer[index]);
    fprintf(outfile,"\t%3.1lf", buffer[index+1]);
    fprintf(outfile,"\t%lf", buffer[index+2]);
    fprintf(outfile,"\t%4.0lf", buffer[index+3]);
    fprintf(outfile,"\t%3.2lf", buffer[index+4]);
    fprintf(outfile,"\t%3.1lf", buffer[index+5]);
    fprintf(outfile,"\t%3.1lf", buffer[index+6]);
    fprintf(outfile,"\t%3.1lf", buffer[index+7]);
    fprintf(outfile,"\t%2.0lf", buffer[index+8]);
    fprintf(outfile,"\t%2.0lf", buffer[index+9]);
    fprintf(outfile,"\t%2.0lf\n", buffer[index+10]);
}

fclose(outfile);
fclose(headfile);

} /* End of Main */

```

```
/* FUZZDATA.H */
```

```
#define FUZZY_ACTIVE YES
#define EFI_ACTIVE YES
#define DES_LAMBDA 0.8
#define DES_EHRD 60.0
#define PRINT_UPDATE YES
#define UPDATE_INTERVAL 2

#define LAMBDA_MIN 0.6
#define LAMBDA_MAX 1.1
#define LAMBDA_DEF 0.75
#define EHRD_MAX 95.0
#define EHRD_MIN 25.0
#define NUM_EHRD_LEVELS 11
#define NUM_EHRD_STATES 7
#define NUM_TIME_LEVELS 3
#define NUM_TIME_STATES 2
#define TIME_ARRAY {{1.0,0.0},{0.5,0.5},{0.0,1.0}}
#define NUM_RPM_LEVELS 3
#define NUM_RPM_STATES 2
#define RPM_ARRAY {{1.0,0.0},{0.7,0.3},{0.0,1.0}}
#define MIN_EHRD {-1000,-10,-7.5,-5.0,-2.5,-1.0,1.0,2.5,5.0,7.5,10}
#define MAX_EHRD {-10,-7.5,-5.0,-2.5,-1.0,1.0,2.5,5.0,7.5,10,1000}
#define MIN_TIME {0,15,25}
#define MAX_TIME {15,25,10000}
#define MIN_RPM {-10000,150,300}
#define MAX_RPM {150,300,10000}
#define MU_RPM_ERROR_0 {{-0.04,-0.025,-0.015,0.0,0.01,0.02,0.03},{-0.04,-0.025,-0.015,0.0,0.01,0.01,0.02}}
#define MU_RPM_ERROR_1 {{-0.04,-0.04,-0.04,-0.015,-0.015,-0.015,-0.015},{-0.025,-0.025,-0.025,0.0,0.0,0.0,0.0}}
#define EHRD_COLUMN_0 {1.0,0.5,0.0,0.0,0.0,0.0,0.0}
#define EHRD_COLUMN_1 {0.0,0.5,1.0,0.5,0.0,0.0,0.0}
#define EHRD_COLUMN_2 {0.0,0.0,0.5,0.7,0.0,0.0,0.0}
#define EHRD_COLUMN_3 {0.0,0.0,0.0,0.3,1.0,0.3,0.0}
#define EHRD_COLUMN_4 {0.0,0.0,0.0,0.0,0.7,0.5,0.0}
#define EHRD_COLUMN_5 {0.0,0.0,0.0,0.0,0.0,0.5,1.0}
#define EHRD_COLUMN_6 {0.0,0.0,0.0,0.0,0.0,0.5,1.0}

#define TIME_RPM_CHANGE 15.0
#define RPM_COLD_DESIRED 1050.0
#define RPM_WARM_DESIRED 900.0
```

```

/* KCFUZZY.H */
/* Header file for Fuzzy Controller tests*/

#define DATA_COLUMNS 11
#define SEC_APPROX 45
#define NUM_AVE 8
#define TEST_CYL_LENGTH (NUM_AVE*SEC_APPROX)
#define SAMPLES_PER_CYCLE 361
#define ARRAY_SIZE (DATA_COLUMNS*TEST_CYL_LENGTH)
#define DataConvertADC 0.001220703 /* volts per count */
#define DataConvertDAC 250.0610501 /* counts per volt */
#define SysBrdConvertADC 0.000091553 /* volts per count */

#define MAF_A2 0.0062288
#define MAF_A1 0.0114725
#define MAF_A0 -0.0001053
#define RESET 0.0
#define CYL_PER_REV 4
#define SEC_PER_MIN 60
#define MS_CONVERT 1000.0

#define EFI_SLOPE 1.9481 /* Gain for PW voltage to EFI Controller */
#define EFI_OFFSET 0.1938 /* Offset for PW voltage to EFI Controller */
#define STOICH 14.64
#define SLOPE 0.00528 /* injector flow (lb/sec) */
#define PW_OFF 0.2
#define ECT_SLOPE -39.8607
#define ECT_OFFSET 189.4428
#define EFI_PW_SLOPE .0975 /* Gain for PW voltage from EFI Controller */
#define EFI_PW_OFFSET -.1155 /* Offset for PW voltage from EFI Controller */
#define GAMMA 1.2

#define DATATYPE float
#define ADC_PRESS 0x00000000
#define ADC_SYNC 0x0010000
#define ADC_ECT 0x0020000
#define ADC_MAF 0x0030000
#define ADC_RPM 0x0040000
#define ADC_PW 0x0050000

#define RPM_GAIN 1000.0
#define RPM_MIN 200.0
#define RPM_BIAS 0.035
#define PRESS_PK 0.2
#define SYNC_HI 2.0
#define PRESS_LO -0.1
#define SYNC_HYS 20
#define RPM_SEC_PER_DEG (6) /* Actually 1/6, this is inverted */
#define TDC_PER_REV 4
#define TDC_CYL_ONE 360
#define DEG_PER_CYCLE 720
#define TDC_CYL_EIGHT 270
#define DEG_REF_PRESS 704

```

```

#define DEG_NEW_CYLINDER 730
#define DEG_END_OF_CYLINDER 720
#define DEG_START_OF_CYLINDER 0
#define CHECK_FOR_TDC 340
#define DEG_READ_ECT 490
#define DEG_READ_MAF 242
#define DEG_READ_RPM 240
#define DEG_READ_PW 238
#define DEG_CALC_EHRD 480
#define DEG_CALC_TIME 500
#define DEG_CALC_LAMBDA 550
#define DEG_CALC_PW 560
#define DEG_BUFFER_DATA 570
#define DEG_DATA_TO_PC 580
#define DEG_START_PW 600
#define NUM_CYCLES_PW 2

#define DEG_FUZZIFY_INPUTS 510
#define DEG_FUZZY_RULES 520
#define DEG_FUZZIFY_OUTPUTS 530

#define NO 0
#define YES 1
#define ON 1
#define OFF 0

#define RES 2
#define BAR_PER_VOLT 5.0
#define BAR_PER_ATM 1.0135
#define NM_PER_BAR 100000

/* ENGSPECS.H */

#define Bore .0902
#define Stroke .09
#define R (0.5*Stroke)
#define L .1507
#define Ratio (R/L)
#define Ap (pi*Bore*Bore*0.25)
#define CompRatio 9.0
#define pi 3.14159
#define gamma 1.4
#define INTAKE_CLOSED 244
#define EXHAUST_OPEN (477)
#define SPARK_DEGREE 350

```