DESIGN OF AN ALTERNATE FUEL INJECTION CONTROLLER

DESIGN OF A CONFIGURABLE ALTERNATE FUEL INJECTION CONTROLLER

By
KEVIN DAGENAIS, B. ENG.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Applied Science
Department of Computing and Software
McMaster University

© Copyright by Kevin Dagenais, May 3, 2005

MASTER OF APPLIED SCIENCE(2005) McMaster University
(Software) Hamilton, Ontario

TITLE: Design of a Configurable Alternate Fuel Injection Controller

AUTHOR: Kevin Dagenais, B. Eng.(McMaster University, Canada)

SUPERVISOR: Dr. M. v. Mohrenschildt

NUMBER OF PAGES: x, 149

Abstract

This thesis presents a strategy for documenting real-time control systems, and the work products that result from its application to the development of an alternate fuel injection controller. In doing so, this document contributes technically to the areas of automotive control, and control systems documentation. The strategy was not developed independently of the control system, but in a manner which reflects its size and complexity.

The controller is used to generate and transmit appropriately timed and sized pulses to an alternative fuel injector array, and switch auxiliary devices including a fuel heater, and an injector lock-off. Such a controller, when used to inject natural gas or propane into a gasoline burning engine, provides a reduction in both engine operating costs and harmful engine emissions.

The controller stores a fuel map that relates the amount of energy released by the combustion of petroleum to that released by the combustion of an alternate fuel, over a range of varying environmental conditions. The fuel map is used to calculate the length of alternate injection pulses. These maps have been designed by, and are the property of Cosimo's Garage Ltd. and thus will not appear in this document.

At present, nearly all large engine car conversion technology is more rigid than the solution provided here. Conversion costs are often prohibitive and problems requiring professional service are frequent. Should the controller described here, help to curb conversion costs and reduce the need for frequent service as is expected, the controller will be a viable candidate for production and sale.

Contents

Abstract iii

1 Introduction 1

1.1 Document Overview . . . 1

1.2 Fuel Injection . . . 2

1.3 Injection and Firing Diagrams . . . 3

1.4 Conversion Motivation . . . 4

1.5 The Primary Control Task . . . 4

2 Software Requirements and Design Documentation Strategy 6

2.1 IEE Guideline . . . 6

2.1.1 Feasibility Study . . . 7

2.1.2 User Requirements Specification . . . 7

2.1.3 Functional Specification . . . 7

2.1.4 Software System Specification . . . 7

2.1.5 Test Documents . . . 8

2.2 The Hoffman and Strooper Method . . . 8

2.2.1 Requirements Specification . . . 9

2.2.2 Module Guide . . . 9

2.2.3 Module Interface Specification . . . 10

2.2.4 Module Internal Design . . . 10

2.3 Software Cost Reduction . . . 11

2.3.1 SCR Requirements . . . 11

2.3.2 The Four Variable Model . . . 12

2.4 Injection Controller Software Documentation Strategy . . . 12

2.4.1 Requirements Specification 12
2.4.2 High Level Design 13
2.4.3 Implementation Description 14
2.4.4 Testing, Inspection and Verification 14

3 Alternate Fuel Injection Controller Requirements Specification 15
3.1 Overview 15
3.2 State Space 16
3.2.1 Monitored Quantities 16
3.2.2 Controlled Quantities 17
3.2.3 State Variables 17
3.2.4 Mode Invariants 18
3.2.5 Event Description 19
3.3 Requirements 19
3.3.1 Safety Requirements 19
3.3.2 Operating Environment 20
3.3.3 Performance and Timing Requirements 20
3.3.4 Control Mode Switching Requirements 21
3.3.5 Injection Status Sub Modes 21
3.3.6 Trouble 22
3.3.7 Injector Grounding 22
3.3.8 Fuel Maps 24
3.3.9 Granularity and Numerical Representation 26
3.3.10 Driver Interface 27
3.3.11 Serial Connection 27
3.3.12 OBC diagnostics 28

4 Alternate Fuel Injection Controller PC Application Requirements Specification 29
4.1 Overview 29
4.2 Requirements 30
4.2.1 Display Requirements 30
4.2.2 Logging Requirements 30
4.2.3 Fuel Map Requirements 31

4.2.4 Safety Requirements . . . 31
4.2.5 Performance and timing . . . 32
4.2.6 Platform . . . 32
4.2.7 Communication . . . 32

5 Hardware Description **34**
5.1 Port and Pin Assignments . . . 35
5.2 Pin Diagram . . . 36
5.3 Hardware Overview . . . 36
5.3.1 Hardware Components in the Prototype . . . 39
5.4 Hardware Circuit Diagrams . . . 40

6 High Level Design **43**
6.1 Overview . . . 43
6.2 Flow Chart Conventions . . . 43
6.3 General Program Structure . . . 45
6.4 Initialization Program Segment . . . 46
6.5 Mode Update and Data Acquisition Main-Loop . . . 48
6.5.1 External Variable Update and Transmission Sub-Chart . . . 50
6.5.2 Enable/Disable Native Injectors Sub Chart . . . 51
6.6 Interrupt Service Request Handler . . . 51
6.6.1 TIMER 1 Overflow . . . 52
6.6.2 Change on Port B . . . 54
6.6.3 Timer 2 Match . . . 55
6.6.4 Interrupt Race Conditions . . . 55
6.7 Programming Program Segment . . . 56
6.8 PC Programming Segment . . . 56

7 Implementation Description **60**
7.1 Overview . . . 60
7.2 PIC Configuration . . . 61
7.2.1 Configuration Bits . . . 61
7.2.2 Timers . . . 62
7.3 Coding Conventions . . . 64

7.4 Requirement Variable PIC Representations 65
7.5 Initialization 68
7.5.1 Assumptions 68
7.5.2 Variables 68
7.5.3 Initialization Code 71
7.6 Interrupt Service Routine (ISR) 78
7.6.1 Assumptions 78
7.6.2 Variables 78
7.6.3 ISR Code 80
7.7 Main Loop 89
7.7.1 Assumptions 89
7.7.2 Variables 89
7.7.3 Main Loop Code 90
7.8 Programming Loop 98
7.8.1 Assumptions 98
7.8.2 Variables 98
7.8.3 Programming Loop Code 99
7.9 Serial Port 100
7.9.1 Assumptions 100
7.9.2 Variables 101
7.9.3 Serial Port Code 101
7.10 A/D Conversion 103
7.10.1 Assumptions 103
7.10.2 Variables 103
7.10.3 A/D Conversion Code 104
7.11 Flash 106
7.11.1 Assumptions 106
7.11.2 Variables 106
7.11.3 Flash Code 107

8 Testing, Inspection, and Verification 112
8.1 Overview 112
8.2 Laboratory Testing 112

8.2.1 Laboratory Testing Tools 112
8.2.2 Lab Tests and Results 115
8.3 Field Testing 127
8.3.1 Field Tests and Results - Modified Lab Tests 128
8.3.2 Field Tests and Results - New Tests 131
8.4 Inspection 133
8.4.1 Explicit Banking 133
8.4.2 Variable Addressing 133
8.4.3 GOTO ISR Instructions 134
8.4.4 Call Stack Size 134
8.4.5 Flash Write Instruction Sequence 134
8.4.6 Variable Naming and Code Casing 134
8.5 Verification 135
8.5.1 Verification of Safety Requirements 135
8.5.2 Verification of Timing Requirements 140

9 Conclusion **142**
9.1 Controller Evaluation 142
9.2 Documentation Strategy Evaluation 142
9.2.1 The Requirements Specification 143
9.2.2 High Level Design Document 143
9.2.3 Implementation Description 144
9.3 The Evolution of the Development Process 144
9.4 Reuse 145
9.5 Future Work 145

List of Figures

1.1 Native Injector Signals 3
1.2 Oscilloscope connected to Ford Police Interceptor 4

3.1 Pulse Waveform 24
3.2 Post Look-up Multiplication Regions 26

5.1 Controller Hardware Prototype 34
5.2 PIC Pin Diagram 37
5.3 Hardware Overview 38
5.4 Hardware Overview 40
5.5 NIRB - Native Injector Resistor Bank 40
5.6 NDB - Native Diode Bank 41
5.7 NITB - Native Injector Transistor Bank 41
5.8 AITB - Alternate Injector Transistor Bank 42

6.1 Flow Chart Legend 44
6.2 Program Structure Flowchart 46
6.3 Initialization Flowchart 47
6.4 Main-Loop Flowchart 49
6.5 External Variable Update and Transmit 51
6.6 Enable/Disable Native Injectors 52
6.7 Interrupt Flowchart 53
6.8 Compute Elongation - Sub-Chart 54
6.9 Programming Flowchart - PIC Side 57
6.10 Programming Flowchart - PC Side 58

8.1 Picture of Lab Testing Environment 113
8.2 OBC Simulator Waveforms 114

Chapter 1

Introduction

Recent years have seen a dramatic decrease in the price of micro-controller technology. So much so, that the successful design and implementation of embedded control systems, once limited to large and specialized organizations, is now well within the reach of small business. In keeping with the common technological trend, this decrease in price was accompanied by a tremendous increase in performance. This has allowed general purpose micro-controllers to move into the real time arena, where previously, timing constraints dictated that control be performed by faster, custom hardware solutions. This combination of increased accessibility and utility has fueled an increase in micro-controller software development. Unfortunately, while micro-controller technology now enjoys widespread use, there are few available examples of complete micro-controller software project documentation. It is among the goals of this thesis to provide one such example. More specifically, it is the primary goal of this thesis to provide complete software documentation, including implementation code, for a configurable automotive alternate fuel injection controller.

1.1 Document Overview

The alternate fuel injection control system described in this document consists of three components.

- Hardware - The electronics used to support the micro-controller.

- Micro-controller Software - The software responsible for completing the control tasks.

- Application Software - The Personal Computer application software used to monitor and configure the controller.

This document is divided into chapters in a manner that reflects the division of the software design process into largely independent stages and in some cases the division of the controller into these three components. Included are chapters pertaining to controller requirements, PC application requirements, hardware design, controller software design, controller software implementation and finally system testing, inspection and verification.

1.2 Fuel Injection

Port fuel injection, as described in [6], is the process by which fuel is forced under pressure into the engine intake manifold. In the manifold, fuel mixes with air before entering the cylinder to be compressed and finally ignited. The amount of fuel in the air-fuel mixture is what determines the force exerted on the piston and the composition of the exhaust gases. There are two factors that determine how much fuel enters the manifold during an engine cycle. The first of these is the amount of time that the fuel injectors remain open, the second is the pressure in the fuel lines leading to the injectors. The pressure however, generally remains constant, and injection timings alone are used to control the amount of fuel being injected. It is important to note that fuel injectors do not receive a 12 volt signal to cause them to open. Rather, when the car is on, the injectors receive a constant 12 volt signal but not always a ground to allow for the flow of current. The car computer, or the controller described in this document, fires the injectors by providing them with a ground. Therefore, when the signal from the controller to a fuel injector is 12 volts, the injector is closed. When the signal is ground, the injector is open.

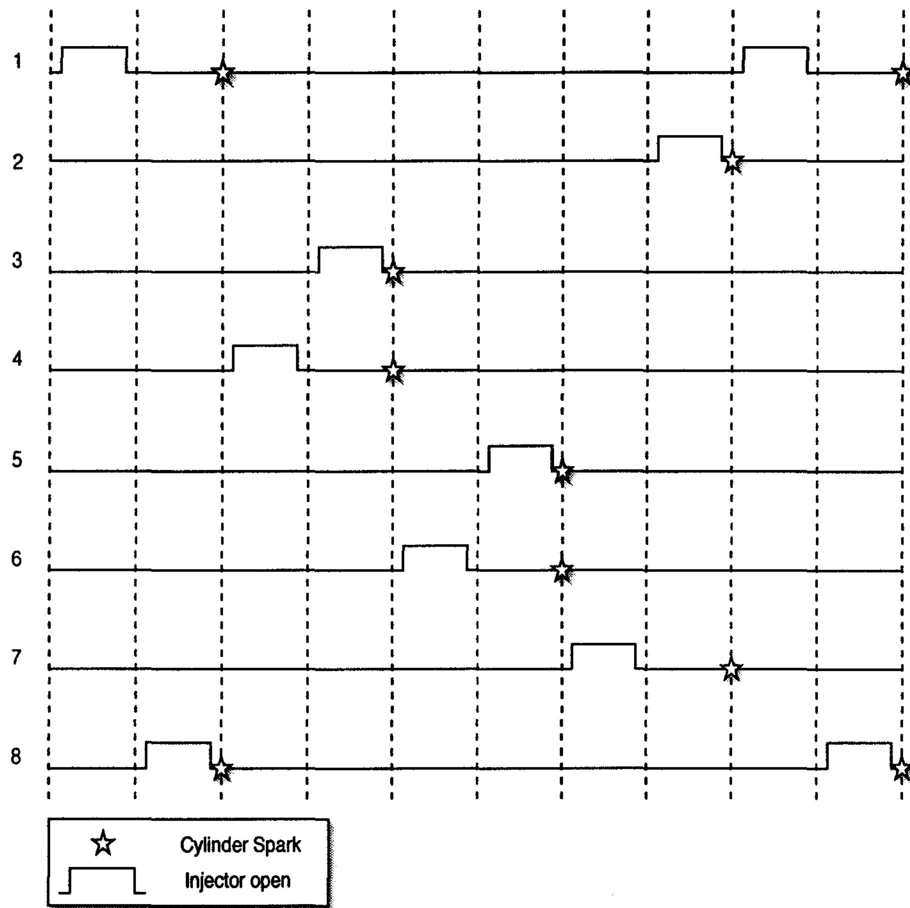


Figure 1.1: Native Injector Signals

1.3 Injection and Firing Diagrams

Figure 1.1 illustrates the relative timing of fuel injection pulses and spark plug firings. The injection ordering is consistent with figure 1.2 which is a photo taken of an oscilloscope while it was connected to the native fuel injector lines of an 8 cylinder "Ford Police Interceptor".

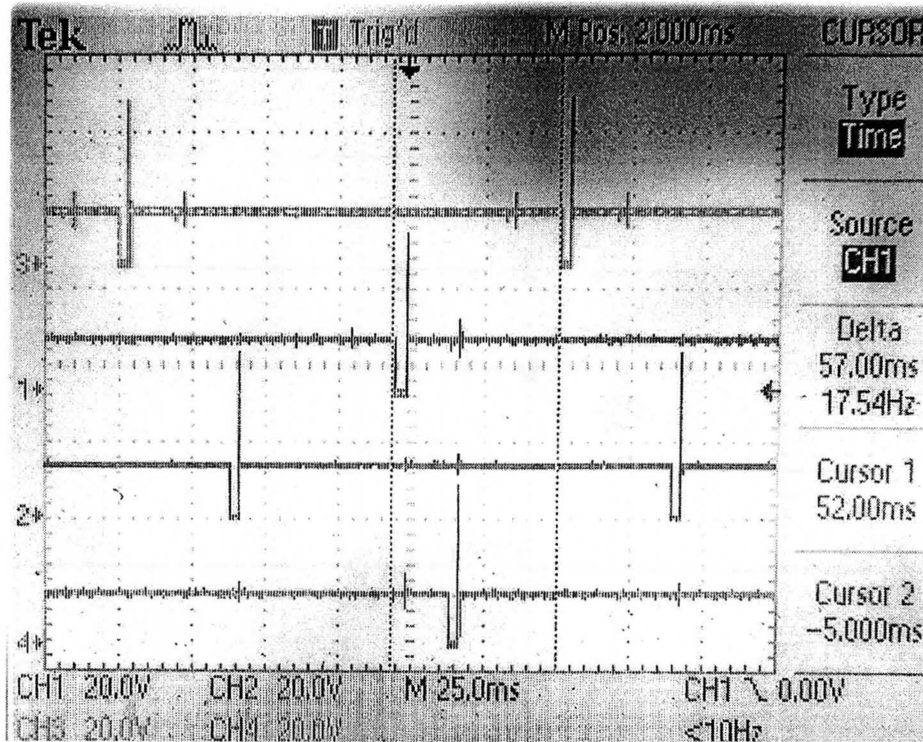


Figure 1.2: Oscilloscope connected to Ford Police Interceptor

1.4 Conversion Motivation

The motivation for converting a gasoline burning automobile to one that also burns an alternate fuel is usually cost reduction. The prices of fuels like propane and natural gas, the two most common alternatives, are significantly less than the price of petroleum. The other major advantage of powering an automobile with one of these fuels is a drastic reduction in the amount of harmful engine emissions.

1.5 The Primary Control Task

For an engine powered by one of the above mentioned alternate fuels to produce an output comparable to an otherwise identical gasoline burning engine, requires an increased mass of fuel per injection. Therefore, injection pulses must be lengthened if comparable engine output is desired. Thus, the job of an alternate fuel injection

controller, is to open an alternate fuel injector at the time when the car computer is intending to open a native injector. Then, keep that alternate injector open for the precise amount of time after the native injector would have been closed to achieve equivalent engine output.

Chapter 2

Software Requirements and Design Documentation Strategy

This chapter provides a literature review which summarizes software documentation guidelines and philosophies outlined by the institute of Electrical Engineers in [11], Daniel Hoffman and Paul Strooper in [4], and Constance Heitmeyer in [3]. These sources were selected for their applicability to real time systems documentation. This review is followed by a description and justification of the documentation strategy employed in this document.

2.1 IEE Guideline

This section reviews the real time software documentation standard proposed in [11]. Documents, or sets of documents proposed therein, are described in terms of their purpose and content. The guideline frequently refers to those in need of system development or selection as *The User*, and refers to those who's task it is to develop a system as *The Supplier*. The guideline has an intended primary readership of small to medium size companies and software providers but is also relevant for larger companies and academics. Reviewed documentation includes the feasibility study, user requirements specification, functional specification, software system specification, and test documentation.

2.1.1 Feasibility Study

The feasibility study is an evaluation of possible solutions to a particular problem. Solutions are evaluated based on criteria including operational and financial benefit, possible hazards, system lifetime, and operational safety. Among the most important results of the feasibility study is a recommendation endorsing the selection of a system already in existence, or the development of a new system. The onus of producing such a document falls upon the user.

2.1.2 User Requirements Specification

The user requirements specification defines the functionality of the system, and lists all associated constraints and factors. Requirements should be precise, complete, consistent and free from language that may prejudice the design. The document is written by the user in language that the supplier can understand and is often considered a contract between the two.

2.1.3 Functional Specification

The functional specification, written by the supplier, is used to augment contractual documentation and outlines how the supplier plans to satisfy the user requirements specification. The document should provide the user with a summary of what the system will do, how it will be operated, and how it will be maintained. It is common for this document to include a description of the proposed solution presented in the form of a block diagram expressing the relationships between functional components. The process of writing such a document is generally iterative, producing several versions as negotiations between the supplier and user advance. The supplier should notify the user of any aspect of the functional specification that is inconsistent with the user requirements specification. Also, the two documents should be similar in structure so that these inconsistencies are more easily uncovered.

2.1.4 Software System Specification

The software system specification is a supplier written document that describes the software in terms of content, structure, and function. The document begins as a

statement of design intent, and is developed in parallel with the software, eventually becoming a comprehensive description of the operational software system. The software system specification is considered both a development and support document, as it facilitates software maintenance as well as design and implementation. When written properly, this document enables qualified developers, lacking prior knowledge of the system, to perform software maintenance tasks.

2.1.5 Test Documents

The IEE Guideline outlines a number of documents that should be produced before and during the testing process. The first of these is a document describing the supplier's test philosophy. Its purpose is to explain how the testing procedures demonstrate the correctness of the system. The test philosophy is also used to explain the principles that guide test case selection. The next document described in the guideline is the test plan. The test plan may restate the test philosophy, but is more concerned with test scheduling and procedure. This document is followed by the test specification. The test specification is a highly detailed description of all tests to be performed. Each test is described in terms of its objective, location, test conditions, configuration, input and output, operational procedure, etc. Finally, test logs record the results of tests, and the test summary lists test failures and unexplained incidents.

2.2 The Hoffman and Strooper Method

This section reviews work products specified by Hoffman and Strooper in [4]. Work products are discussed in terms of their purpose, intended readership and content. Hoffman and Strooper refer to four classes of concerned parties, these being *Users*, *Designers*, *Developers*, and *Verifiers*. The people who fill these roles, however, differ between work products. The documents reviewed are the Requirements Specification, the Module Guide, the Module Interface Specification, and the Module Internal Design.

2.2.1 Requirements Specification

The Requirements Specification is the document in which the behaviour of a software system is explicitly defined. The requirements specification supports all four groups mentioned in the text. The users, in this case the end users, provide information to the designers who write the document. When completed, the requirements specification becomes a contract between these two parties. Developers work from the requirements specification rather than deciding on behalf of the users how the system should operate. Finally, verifiers, in this case primarily testers, use the requirements specification as a basis for their testing and verification.

Listed and described below are a number of sections commonly found in requirements specifications.

- **Environment Variables** - Defines how aspects of the system's environment are to be modeled by the software system as inputs and outputs.
- **State Machine** - Provides possibly several Finite State Machines which describe the desired behavior of the system.
- **Functions** - Defines constants, types, and functions used throughout the document.
- **Expected Changes** - Used to identify likely future changes in the requirements so that the design can be produced in such a way that makes these changes as simple and inexpensive as possible.

2.2.2 Module Guide

Software modularization is a process by which large and complex software systems are broken down into smaller more manageable components. These components, or modules, are essentially programming work assignments. The product of a good modular decomposition is a set of modules that are manageable in size and complexity, and are largely independent. Further to this, if the decomposition is driven by information hiding, each module should encapsulate a likely change. In this way, the details likely to change are kept secret to a single module, and if the change becomes necessary, effects are for the most part limited to that module. The module guide is

the work product used to present module decompositions. The two sections which make up a module guide are the **Module Summary** section and the **Module Service and Secret** section. The module summary section lists the modules and groups them in terms of the type of secret they keep. The module service and secret section describes the modules in terms of the likely change they encapsulate and the service they provide.

2.2.3 Module Interface Specification

The module interface specification, or MIS, is the document where the assumptions about module behavior and use that comprise module interfaces are provided. With respect to the four groups previously mentioned, the designer writes an interface specification to reflect the desired observable behavior of a module. The developer creates an implementation to satisfy that specification. The verifier determines whether or not the implementation satisfies the specification. Lastly the user, generally a programmer, uses the interface specification to command the services provided by the module. When compared to source code, which is often used to specify interfaces, an MIS allows for more parallel development, reduces errors that arise from incorrect assumptions, and provides more guidance for test case development.

The MIS for each module is divided into two sections, syntax and semantics. The syntax section deals with naming and typing of parameters and return values, as well as exceptions, constants and exported types. The semantics section lists assumptions, and describes how the state variables are modified by calls to the access routines. Also included are a state invariant, local functions, local constants and local types.

2.2.4 Module Internal Design

The abstract state variables of the MIS are chosen to be clear and expressive. It is therefore not uncommon for the types of those variables to be either unsupported or operationally/spatially inefficient in the implementation language. In these situations, the concrete state of the implementation often differs from the abstract state of the MIS. The Module Internal Design, or MID, is a document which provides a link between the abstract and concrete state spaces when such differences exist. The users of this document consist of the module designers and implementers.

The MID lists the concrete state variables, and describes how they are affected by calls to the access routines. The MID also provides a state invariant which restricts the legal state space of the concrete variables before and after access routine calls. Finally, the MID provides an abstraction function which associates each legal concrete state to a corresponding abstract state.

2.3 Software Cost Reduction

This section reviews software cost reduction (SCR) as described by Constance Heitmeyer in [3]. Unlike the previous sections that were primarily concerned with the content and structure of work products, the discussion of SCR is limited to its principles and techniques. More specifically, this section describes the SCR approach to requirements and the four variable model.

2.3.1 SCR Requirements

This section outlines three fundamental aspects of the SCR requirements approach. These are:

- A focus on outputs
- Specifying outputs using a tabular notation
- Requirements evaluation criteria

The SCR approach to requirements is considered output focused in that the value of each output that the software system is required to produce is specified by a function of the environment's past and current states. These functions are generally presented in SCR tables.

SCR tables, which include Mode Transition, Event, and Condition tables, facilitate the writing, comprehension, and verification of the functions which they encapsulate. Mode transition tables express functions from a mode and an event to a new mode, and only consider events which cause transitions. Event and condition tables express the values of controlled variables or terms. These values depend on events and conditions respectively as well as modes.

SCR requirements documents must satisfy a number of evaluation criteria. These include completeness, implementation independence, and the organizational constraint that they be reference documents.

2.3.2 The Four Variable Model

The four variable model expresses required software system behaviour using four variable types and four relations. The four types of variables are monitored, controlled, input and output. Monitored and controlled variables are representations of quantities external to the system that the software must monitor and control. Input and output variables correspond to the interfaces of input and output devices to which the system has direct access.

The four relations are NAT, REQ, IN, and OUT. Together, NAT and REQ express ideal system behavior by providing valid assumptions about the environment and defining the required relationships between monitored and controlled variables. IN and OUT define mappings from monitored to input variables and from output to controlled variables respectively, that reflect the properties of the hardware input and output devices.

2.4 Injection Controller Software Documentation Strategy

This section describes and justifies the software documentation strategy that guided the development of this document. Reasons are also provided for the omittance of documents, and the exclusion of particular design philosophies.

2.4.1 Requirements Specification

The requirements specification for the fuel injection controller was developed with the intent of completely specifying the behavior of the system and listing all constraints outlined by the end user. These goals are common to each of the literature sources examined in this review [11, 4, 3]. Contractually, this document would have served as an agreement between the system developers and Cosimo's Garage ltd. if in fact

the system were being developed for compensation. This is also consistent with the reviewed sources.

In contrast to the IEE guideline [11], the requirements specification was not written by the end user but by the system developer, and was not preceded by a formally documented feasibility study. The requirements document was authored by the system developers as no person among the end users was sufficiently qualified to write a document that would meet the objectives listed above. The content of the document, however, was elicited from the end user by the author. Development of a feasibility study, according to the guideline, is the responsibility of the user and is therefore omitted.

The requirements specification partially employs the SCR approach [3] by mathematically specifying outputs using functions presented in SCR tables. Also, the terms monitored and controlled variables are used to represent environmental quantities. The benefits of this approach are outlined in the SCR section of this chapter.

The organization of the specification, is such that requirements pertaining to a specific environmental quantity or control task may appear in more than one place. This is a direct result of a deliberate effort to first specify the necessary behavior and properties of the controller, before specifying the negation of all unacceptable behavior and properties. The structure of the resulting requirements specification is deemed helpful for the industrial partner as it closely resembles the way in which he verbally specified the requirements.

2.4.2 High Level Design

The high level design document serves to describe the decomposition and sequencing of the software system. The system decomposition identifies program segments that are manageable in size and complexity, while the program sequence ensures that time bounds can be successfully met. The design is presented using flowcharts as they constitute a minimal, well defined tool, for characterizing the system in terms of both of these properties. While state charts currently enjoy wide spread use, their semantics are not as easily defined, and they provide no advantage over flowcharts in expressing sequencing and decomposition information.

The decomposition of the software system into program segments does provide

information hiding, however a one to one correspondence between secrets and modules is not present in all cases. To reflect this deviation from the Hoffman and Strooper method, the word module has not been used to describe the software units resulting from the decomposition. The benefits of such a modular decomposition are only fully realized when applied to multi-version, multi-person system development. Seeing as the development of the system described in this document was neither, the additional benefits would have been marginal.

While the format of the high level design differs from the Software System Specification template provided in the IEE guideline, the two documents are quite similar in purpose and content.

2.4.3 Implementation Description

The implementation document provides code, a mapping from state space to concrete variables, and a listing of the variables that are read and changed by each programming segment of the software system. The listings of changed and read variables effectively constitute an interface specification, while the mapping is used to show the relationship between the implementation variables, and the input, output, and state variables of the requirements document. This information is included in keeping with the rational for creating an MIS and MID outlined by Hoffman and Strooper in [4].

2.4.4 Testing, Inspection and Verification

The last software document deals with the testing, inspection, and verification of the software system. The chapter includes discussions of testing tools, laboratory tests, field tests, inspection criteria and verifiable software properties. The sections that deal with testing convey the information normally found in test plans and specifications as described by the IEE guideline. The inspection and verification sections mirror the inspection and verification processes that were employed to increase confidence in the controller's software component.

Chapter 3

Alternate Fuel Injection Controller Requirements Specification

This chapter is a requirements specification for a multi-fuel, multi-engine format, alternate fuel injection controller.

3.1 Overview

The controller specified in this chapter is responsible for generating and transmitting appropriately timed and sized pulses to an alternate fuel injector array. The lengths of these pulses are determined primarily by the lengths of pulses sent from the car computer and bound for the stock injectors, the intake air temperature, and the barometric pressure. Precise adjustments to these lengths are made depending on other engine parameters including but not limited to the fuel trim and the throttle position. The controller is responsible for switching peripheral hardware such as a fuel heater and an injector lock-off. Also, the device must provide a user interface to inform the operator of the status of fuel reserves and the presence of trouble while receiving user input used to determine which fuel to inject. Another interface will be used to export machine readable information and to update fuel maps.

3.2 State Space

This section provides a listing and description of quantities to be monitored, controlled and internally maintained by the system. Monitored or input variables are prefixed by i_, controlled or output variables are prefixed by o_ , and maintained or state variables are prefixed by s_.

3.2.1 Monitored Quantities

Table 3.1 lists the environment variables that are monitored by the system. Each variable is characterized in terms of its unit of measure, and the range of values it may exhibit. In the cases where the unit is specified as an enumerated set, the value range includes every element of that set. A short description of each variable is also provided.

Table 3.1: Monitored Quantities

Name	Type	Range	Description
i_injector$_{k,k=1\cdots8}$	$\{open, closed\}$		Injection grounded
i_fuelSelector	$\{on, off\}$		User fuel selection
i_programingSwitch	$\{up, down\}$		Depress to program
i_coolantTemp	volts	0 - 5	Engine Coolant temperature
i_o2Sensor	volts	0 - 5	Fuel trim sensor
i_throttlePos	volts	0 - 5	Throttle position sensor
i_baroPres	volts	0 - 5	Barometric pressure
i_airInTemp	volts	0 - 5	Air intake temperature
i_altFuelLevel	volts	0 - 5	Alternate fuel reserve level
i_rpm	volts	0 - 5	Revolutions per minute
i_o2Thresh$_{k,k=1\cdots4}$	volts	0 - 5	O2 region boundaries
i_tpThresh$_{k,k=1\cdots4}$	volts	0 - 5	TPS region boundaries
i_o2Mult$_{k,k=1\cdots4}$	real	0.5 - 1.5	O2 region multipliers
i_tpMult$_{k,k=1\cdots4}$	real	1 - 1.5	TPS region multipliers

3.2.2 Controlled Quantities

Table 3.2 lists the environment variables that are controlled by the system. Each variable is characterized in terms of its unit of measure, and the range of values it may be assigned. In the cases where the unit is specified as an enumerated set, the value range includes every element of that set. A short description of each variable is also provided.

Table 3.2: Controlled Quantities

Name	Type	Range	Description
$\mathsf{o_natInjector}_{k,k=1\cdots8}$	{*open*, *closed*}		Injection grounded native
$\mathsf{o_altInjector}_{k,k=1\cdots8}$	{*open*, *closed*}		Injection grounded alternate
o_heaterRelay	{*active*, *inactive*}		Alternative fuel heater
o_lockOffRelay	{*active*, *inactive*}		Solenoid lock off
o_altFuelLevelDisp	quarter tanks	0 - 4	Fuel reserve level indicator
o_nativeLED	{*on*, *off*}		User interface mode indicator
o_altLED	{*on*, *off*}		User interface mode indicator
o_stbyLED	{*on*, *off*}		User interface mode indicator

3.2.3 State Variables

Table 3.3 lists the state variables that are internally maintained by the system. Each variable is characterized in terms of its unit of measure, which is specified as an enumerated set. A short description of each state variable is provided.

Table 3.3: State Variables

Name	Type	Description
s_controlMode	{init, stbyNat, altFuel, native, program}	System control mode
s_injectionStatus	{pulse, elongation, interpulse}	Injector bank status
s_trouble	{*on*, *off*}	System trouble state

3.2.4 Mode Invariants

Provided here are lists of invariants associated with specific values of s_controlMode and s_trouble.

Table 3.4, presents invariants for s_controlMode, but makes no mention of the value init since values of several environment variables are unknown at the time of system start-up.

Table 3.4: s_controlMode Invariant Table

Mode	Invariant
program	$\forall k(\text{o_natInjector}_k = \text{i_injector}_k)$
	$\forall k(\text{o_altInjector}_k = \mathit{closed})$
	o_nativeLED = *on*
	o_altLED = *on*
	o_lockOffRelay = *inactive*
	o_heaterRelay = *inactive*
native	$\forall k(\text{o_natInjector}_k = \text{i_injector}_k)$
	$\forall k(\text{o_altInjector}_k = \mathit{closed})$
	o_nativeLED = *on*
	o_lockOffRelay = *inactive*
	o_heaterRelay = *inactive*
stbyNat	$\forall k(\text{o_natInjector}_k = \text{i_injector}_k)$
	$\forall k(\text{o_altInjector}_k = \mathit{closed})$
	o_nativeLED = *off*
	o_altLED = *off*
	o_stbyLED = *on*
	o_lockOffRelay = *active*
	o_heaterRelay = *active*
altFuel	$\forall k(\text{i_injector}_k = \mathit{open} \rightarrow \text{o_altInjector}_k = \mathit{open})$
	$\forall k(\text{o_natInjector}_k = \mathit{closed})$
	o_nativeLED = *off*
	o_altLED = *on*
	o_stbyLED = *off*
	o_lockOffRelay = *active*
	o_heaterRelay = *active*

Table 3.5 provides an invariant for the situation when s_trouble has the value *on*. The invariant is presented in terms of a restriction on the value of s_controlMode.

Table 3.5: s_trouble Invariant Table

Mode	Invariant
on	s_controlMode = native

3.2.5 Event Description

Table 3.6 introduces events that are used later in this chapter when defining mode transitions.

Table 3.6: Event Table

Event Name	Description
reset	A cycling of power to the system, or a restarting of the software for any reason
non_term_pulse	The detection of an injection pulse with a duration greater than 26.2132 ms
elong_over_pulse	The detection of an overlapping native pulse and an alternate pulse elongation

3.3 Requirements

This section lists both functional and non-functional requirements that must be satisfied by any successful controller design or implementation.

3.3.1 Safety Requirements

Requirements listed here are those that ensure the safety of the driver and the engine insofar as that safety depends on the controller.

1. Once the system is initialized, exactly one fuel injector shall fire for every incoming injection pulse.
2. Fuel injectors fired must be associated with the same cylinders as incoming injection pulses.
3. In all situations where s_trouble = *on*, fuel injection control shall default to the car computer.
4. The lock off relay must remain inactive in the native and program modes.
5. Pulse elongations must not overlap with subsequent native injection pulses.

3.3.2 Operating Environment

This section deals with the requirements of the controller after an installation that places it sufficiently far from all sources of heat, moisture and electromagnetic interference.

1. The system must operate normally at all temperatures between −30°C and 60°C.
2. The system must sustain no damage when exposed to temperatures between −40°C and 70°C.
3. The system must operate normally under all conditions generally realized at installation suitable areas of a running automobile's engine compartment.

3.3.3 Performance and Timing Requirements

This section deals with injection timings and acceptable delays as well as performance requirements determined by projected maximum workloads.

1. The controller shall be capable of detecting and generating pulses to keep an 8 cylinder engine running at up to 6500 RPM.
2. The controller shall be capable of detecting and generating pulses to keep an 8 cylinder engine running at as few as 500 RPM.

3. Every injection pulse generated by the car computer must arrive at the destination cylinder(s) within $5\mu s$.
4. i_coolantTemp must be sampled at a frequency of at least 1 Hz
5. i_o2Sensor must be sampled at a frequency of at least 10 Hz
6. i_baroPres must be sampled at a frequency of at least 1 Hz
7. i_airInTemp must be sampled at a frequency of at least 1 Hz
8. i_throttlePos must be sampled at a frequency of at least 10 Hz
9. i_altFuelLevel must be sampled at a frequency of at least 1 Hz
10. i_fuelSelector must be sampled at a frequency of at least 10 Hz
11. s_controlMode must be updated at a frequency of at least 10 Hz

3.3.4 Control Mode Switching Requirements

Table 3.7 is a mode transition table that describes the behavior of s_controlMode. Initially, the value of s_controlMode is init.

3.3.5 Injection Status Sub Modes

Tables 3.8 and 3.9 present the transitions of the state variable s_injectionStatus. When the value of s_injectionStatus is interpulse, the input signals for all cylinders are at 12 volts. When the value of s_injectionStatus is pulse, at least one of the input signals is at ground. Finally, a value of elongation indicates that the system is in a state where the signal to an alternate injector is ground but there is no grounded input injector signal.

1. The state variable s_injectionStatus initially has the value interpulse.
2. In the event that the state variable s_injectionStatus has the value elongation at the time that s_controlMode transitions away from the altfuel mode, $@F$(s_controlMode = altFuel), s_injectionStatus is assigned the value interpulse, s_injectionStatus := interpulse.

Table 3.7: Mode Transition Table of s_controlMode

Mode	Event	New Mode
init	$@T(true)$ **WHEN** i_programmingSwitch = *down*	program
	$@T(true)$ **WHEN** i_programmingSwitch = *up*	native
program	*reset*	init
native	*reset*	init
	$@T$(i_fuelSelector = *on* $\wedge$ s_trouble = off)	stbyNat
stbyNat	*reset*	init
	$@T$(i_fuelSelector = *off* $\vee$ s_trouble = on)	native
	$@T$(i_coolantTemp > 2 volts $\wedge$ i_altFuelLevel > 0.6 volts $\wedge$ i_rpm > 1 volt $\wedge$ s_injectionStatus = *interpulse*)	altFuel
altFuel	*reset*	init
	$@T$(i_fuelSelector = *off* $\vee$ s_trouble = on)	native
	$@T$(i_altFuelLevel < 0.4 volts)	stbyNat

3.3.6 Trouble

Table 3.10 defines the transitions of the state variable s_trouble. s_trouble initially has the value off.

3.3.7 Injector Grounding

This section describes how injection grounding durations are measured and calculated. As the controller is digital, injector signals are sampled and the amount of time that current flows through the native injectors is approximated.

1. Leading and trailing edges of native injection pulses must be detected within 240 ns of their arrival to the controller.

2. The output to alternate injectors shall be updated with a frequency of at least 200,000 Hz. This value is represented in Figure 3.1 as $\frac{1}{T}$.

Table 3.8: State Transition Table of s_injectionStatus when (s_controlMode = init ∨ s_controlMode = program ∨ s_controlMode = native ∨ s_controlMode = stbyNat)

Mode	Event	Action	New Mode
interpulse	$@T(\exists k.\text{i_injector}_k = open)$	$\text{o_natInjector}_k := \text{i_injector}_k$ $k = 1 \cdots 8$	pulse
pulse	$@T(\neg\exists k.\text{i_injector}_k = open)$	$\text{o_natInjector}_k := \text{i_injector}_k$ $k = 1 \cdots 8$	interpulse

Table 3.9: State Transition Table of s_injectionStatus when s_controlMode = altFuel

Mode	Event	Action	New Mode
interpulse	$@T(\exists k.\text{i_injector}_k = open)$	Reset *clock*; $\text{o_altInjector}_k := \text{i_injector}_k$ $k = 1 \cdots 8$	pulse
pulse	$@T(\neg\exists k.\text{i_injector}_k = open)$	$pulselength := clock$; Reset *clock*	elongation
elongation	$@T(clock > ElLength())$	$\text{o_altInjector}_k := closed$ $k = 1 \cdots 8$	interpulse

3. Pulse length refers to the number of periods of length T that a signal is below 6 volts.

4. The width of an incoming pulse must be measured to within an accuracy of $\pm T$.

   The measured width of an incoming pulse jT is equal to

$$jT|(j-1)T \leq \Delta t \leq (j+1)T \tag{3.1}$$

   Outgoing injection pulses must remain grounded for mT

$$mT||mT - \text{ideal}| \leq \frac{T}{2} \wedge \neg\exists n|n > m \wedge |nT - \text{ideal}| \leq \frac{T}{2} \tag{3.2}$$

   where ideal is the sum of the incoming pulse length and the elongation calculated using the fuel maps, the incoming pulse length, the fuel trim and the throttle position, unless such a pulse would cause injector overlap.

Table 3.10: State Transition Table of s_trouble

Mode	Event	New Mode
off	*non_term_pulse*	on
off	*elong_over_pulse*	on
on	*reset*	off

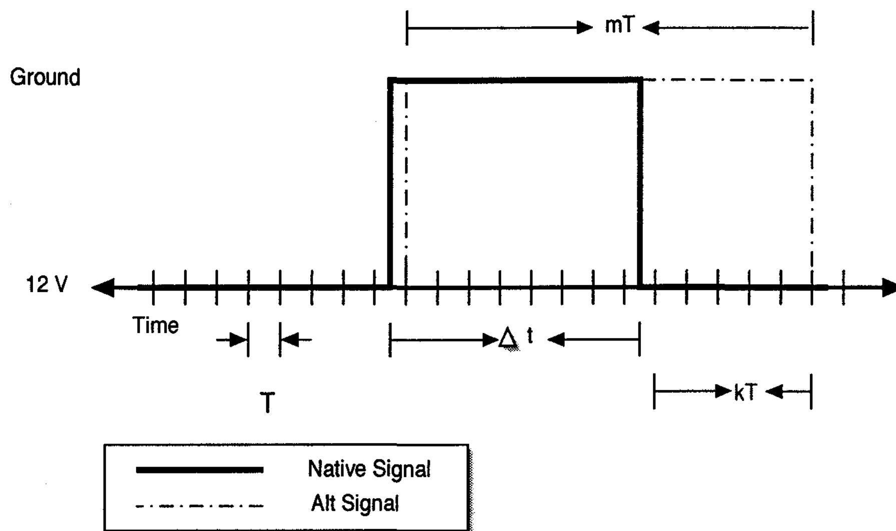


Figure 3.1: Pulse Waveform

$kT = mT - jT$

3.3.8 Fuel Maps

Computing the amount of time to keep an alternate injector open, based on all relevant engine operating parameters, would consume more time than can be spared by the controller. For this reason, the system will incorporate a fuel map used to relate ambient pressure and temperature, to an elongation factor. Besides reducing computational complexity at runtime, a fuel map is also useful to practitioners as it allows them to make small changes to very specific areas of the operating range without changing the continuous model of the system. The function *FuelMap*() takes

no parameters but has access to all state variables defined in this document. The variables that are used by the function are i_airInTemp and i_baroPres. The Output is a fixed point number between 0 and 0.5 that is used to compute the length of pulse elongations. The fuel map values shall be pre-computed using a PC and stored in the flash memory of the controller.

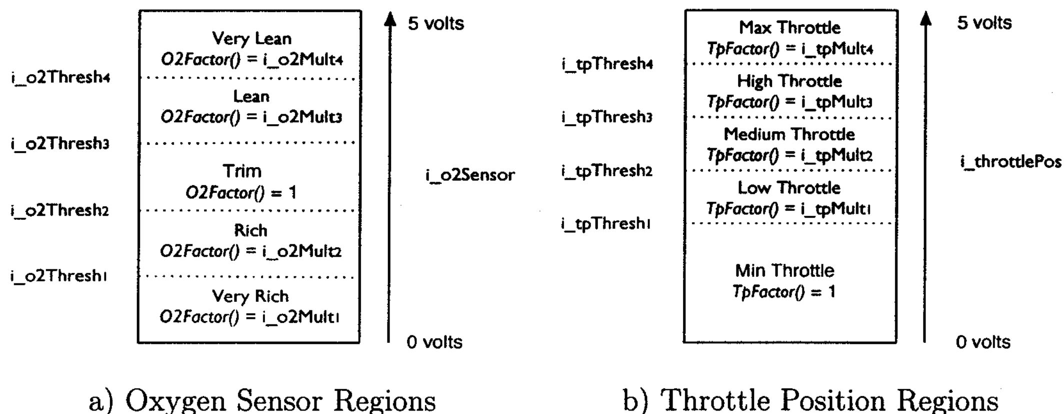
The length of pulse elongation also depends on fuel trim and the amount that the gas pedal is depressed. The state variables i_o2Sensor and i_throttlePos are used to reason about these properties. The values from both of these sensors are compared against thresholds which divide their sensor ranges into 5 distinct regions. In both cases, one of the five regions represents optimal operation and therefor corresponds to a multiplier of 1. The other 4 regions are associated with possibly non-one multipliers which are used to adjust pulse lengths. Figures 3.2 illustrates how the ranges of i_o2Sensor and i_throttlePos are divided into regions and what multipliers they are associated with.

Here the functions are described explicitly. The following can be assumed to be a property of the state variables:

$\mathsf{i_o2Thresh}_1 < \mathsf{i_o2Thresh}_2 < \mathsf{i_o2Thresh}_3 < \mathsf{i_o2Thresh}_4$ and
$\mathsf{i_tpThresh}_1 < \mathsf{i_tpThresh}_2 < \mathsf{i_tpThresh}_3 < \mathsf{i_tpThresh}_4$.

$$O2Factor() = \begin{cases} \mathsf{i_o2Mult}_1 & \text{if } \mathsf{i_o2Sensor} \leq \mathsf{i_o2Thresh}_1, \\ \mathsf{i_o2Mult}_2 & \text{if } \mathsf{i_o2Thresh}_1 < \mathsf{i_o2Sensor} \leq \mathsf{i_o2Thresh}_2, \\ 1 & \text{if } \mathsf{i_o2Thresh}_2 < \mathsf{i_o2Sensor} \leq \mathsf{i_o2Thresh}_3, \\ \mathsf{i_o2Mult}_3 & \text{if } \mathsf{i_o2Thresh}_3 < \mathsf{i_o2Sensor} \leq \mathsf{i_o2Thresh}_4, \\ \mathsf{i_o2Mult}_4 & \text{if } \mathsf{i_o2Sensor} > \mathsf{i_o2Thresh}_4. \end{cases} \tag{3.3}$$

$$TpFactor() = \begin{cases} 1 & \text{if } \mathsf{i_throttlePos} \leq \mathsf{i_tpThresh}_1, \\ \mathsf{i_tpMult}_1 & \text{if } \mathsf{i_tpThresh}_1 < \mathsf{i_throttlePos} \leq \mathsf{i_tpThresh}_2, \\ \mathsf{i_tpMult}_2 & \text{if } \mathsf{i_tpThresh}_2 < \mathsf{i_throttlePos} \leq \mathsf{i_tpThresh}_3, \\ \mathsf{i_tpMult}_3 & \text{if } \mathsf{i_tpThresh}_3 < \mathsf{i_throttlePos} \leq \mathsf{i_tpThresh}_4, \\ \mathsf{i_tpMult}_4 & \text{if } \mathsf{i_throttlePos} > \mathsf{i_tpThresh}_4. \end{cases} \tag{3.4}$$



a) Oxygen Sensor Regions b) Throttle Position Regions

Figure 3.2: Post Look-up Multiplication Regions

Now that the types of the functions *FuelMap*(), *O2Factor*(), and *TpFactor*() have been specified, a formula for computing pulse elongations can be given as well.

$$ElLength = \text{incoming pulse length} * FuelMap() * O2Factor() * TpFactor() \quad (3.5)$$

1. Pulses may not be contracted since pulse elongation calculations require the value of the duration of the native pulse.

2. Fuel map elongation lengths shall be chosen with the intent of achieving a clean, efficient, and driver oriented combustion stroke.

3.3.9 Granularity and Numerical Representation

This section deals with the precision of data representation inside the controller.

1. All analogue values being monitored shall be represented by no fewer than 8 bits.

2. Durations of inbound pulses shall be represented by no fewer than 16 bits.

3. Durations of pulse elongations shall be represented by no fewer than 8 bits.

3.3.10 Driver Interface

This section describes the heads up driver display.

1. The alternate fuel reserve level shall be displayed to the driver on a 0 - 4 scale.
2. An alternate fuel reserve level indicator value of 4 represents a tank between full and 7-8ths full
3. An alternate fuel reserve level indicator value of 3 represents a tank less than 7-8ths full and at least 5-8ths full.
4. An alternate fuel reserve level indicator value of 2 represents a tank less than 5/8th full and at least 3-8ths full.
5. An alternate fuel reserve level indicator value of 1 represents a tank less than 3/8th full and at least 1-8th full.f
6. An alternate fuel reserve level indicator value of 0 represents a tank less than 1/8th full.
7. The driver interface shall indicate when the system is in a state of trouble.
8. The driver interface shall provide the driver a means of indicating the fuel desired for powering the car.

3.3.11 Serial Connection

This section describes the functionality provided by the serial connection.

1. The Controller shall provide a serial connection to a personal computer.
2. The serial connection shall adhere to the RS232C serial protocol.
3. Serial communication shall employ 8 data bits, 1 stop bit, no parity and no handshake.
4. The serial connection shall provide a means to transfer fuel maps from the Personal computer to the controller.

5. The serial connection shall export engine operating parameters from the controller to the personal computer.

3.3.12 OBC diagnostics

The automobile's on-board computer performs several diagnostics to verify that the car is operating correctly. One of these checks that there is resistance on the lines leading to the fuel injectors. If this resistance is not present, the computer assumes that there is a short and lights the check engine signal on the dash. As the controller is intercepting this signal it must ensure that the OBC diagnostic does not come to the above conclusion.

1. The controller must provide a resistance of no less than 10Ω and no more than $15\ \mathrm{K}\Omega$ on each native injector input line.

Chapter 4

Alternate Fuel Injection Controller PC Application Requirements Specification

This chapter contains requirements for the system used to configure and monitor the alternate fuel injection controller.

4.1 Overview

The computer application specified in this chapter is responsible for providing on-line communication with the fuel injection controller. This communication is bidirectional and serves two fundamental purposes. The first of these is providing the servicing technician with information pertaining to the operation of the controller. The second is providing the servicing technician a means to configure the controller by uploading fuel maps, threshold boundaries, and threshold multipliers. As will be discussed in more detail later in this chapter, communication will adhere to the RS232C serial protocol. It should be noted that this application is being specified to interface with a controller that has already been designed. This has resulted in a situation where several issues that would usually be considered during the software design phase have been dealt with in this document to ensure a compatible interface.

4.2 Requirements

This section lists both functional and non-functional requirements that must be satisfied by any successful application design or implementation.

4.2.1 Display Requirements

Requirements listed here are those that describe what data must be displayed by the system and in what ways it must be displayed.

1. The system shall display, both numerically and graphically, a subset of the data transmitted by the controller over the serial port.

2. Membership in the displayed subset implies that a data element has or will be displayed for some amount of time by the application.

3. The displayed subset shall include at least one data element from each uniquely defined value type received each second.

4. When a value element is displayed it shall replace the previously displayed element of the same value type.

5. Data shall be displayed in the order that it arrives.

6. The system shall display a descriptive label for each value type.

7. Descriptive labels shall be terms from the domain language.

8. The system shall continue to display the most recently received data in the case that data reception ends.

9. The system shall indicate to the user when the data being displayed is "Stale". Stale data is any data displayed for 2 or more seconds after being received.

4.2.2 Logging Requirements

Requirements listed here are those dealing with what data needs to be logged by the system, as well as how logged data needs to be organized.

1. The system shall log a subset of the data received on the serial port.
2. The system shall log all data elements that are members of the displayed subset.
3. Logged data shall be organized in a way that preserves the order of arrival.
4. Logged data shall be organized in a way that is compatible with third party statistical, graphing, and archiving systems.
5. The system shall not destroy logs previously produced by the system.

4.2.3 Fuel Map Requirements

Requirements listed here deal with the creation, retrieval, verification and transmission of fuel maps.

1. The system shall provide a means to select a fuel map.
2. The system shall verify that fuel maps satisfy safety requirements listed in this chapter.
3. The system shall transfer fuel maps to the controller over the serial port.
4. The system shall adhere to the transfer protocol presented in the controller high level design chapter.
5. Fuel maps shall be editable by any standard ASCII text editor.

4.2.4 Safety Requirements

Requirements listed here are those that ensure the safety of the driver, passengers, and the engine insofar as that safety depends on the application. As the controller will always be burning native fuel when it is accepting input from this system, this discussion can be limited to fuel map validation and transmission verification.

1. The application shall NOT transmit to the controller any fuel map with one or more values greater than 255, or less than 0.
2. The application shall NOT transmit to the controller any fuel map with a number of entries not equal to 16,384.

4.2.5 Performance and timing

Although this application is not considered a real-time system, it is important to consider certain timing issues. Among these issues are communication baud rates and sampling for display and data logging. As the platform for this application is not a real time operating system, it is important to realize that values listed here are guidelines, and occasional failure in meeting these requirements does not necessarily constitute a system failure.

1. The system shall display data coming from the controller at a frequency of 1Hz.
2. The system shall log data coming from the controller at a frequency of 3 Hz.

4.2.6 Platform

This section contains requirements pertaining to the operating system and computer hardware that will support the application. These requirements come directly from the client and thus could not be left to the designers.

1. The application shall operate under a version of the Microsoft Windows operating system with the .NET Framework installed.
2. The application shall run on both lap-top and desktop computers.

4.2.7 Communication

This section deals with the protocols and communication standards that must be employed by the application in order to successfully communicate with the controller.

1. Communication with the controller shall be conducted over the serial port.
2. Communication shall employ the RS232C serial protocol.
3. The serial baud rate shall be 9600 bps.
4. The serial data format will use 8 data bits.
5. The serial data format will use 1 stop bit.

6. Parity shall not be used during serial communication.
7. There shall be no handshake to initiate serial communication.
8. The system shall transmit in pairs of bytes.
9. The second byte shall have the value 0 in every instance that the first byte represents a fuel map value being transmitted for the first time.
10. The second byte shall have the value 255 in every instance that the first byte represents a fuel map value being retransmitted.
11. The system shall transmit a new fuel map value as the first byte of a communication block after each completely correct echo.
12. The first byte of a block shall be retransmitted every time the controller fails to properly echo the first byte of the previous byte pair, but correctly echoes the second.
13. The system will discontinue communication, and inform the user to restart the transmission in the event the controller fails to correctly echo the second byte of the previous byte pair.
14. A completely correct echo is a two-byte block sent from the controller to the application that consists of the same information that was sent from the application during its last transmission.
15. The system shall consider an echo where the second byte differs by four or more bits from the first byte of the last transmission as a failure to echo the second byte.
16. The system shall consider an echo where the first byte differs in any way from the first byte of the last transmission as a failure to echo the first byte.

Chapter 5

Hardware Description

This chapter provides a general description of the controller hardware design proposed by Dr. M. v. Mohrenschildt. Included are a listing of port and pin assignments to state space variables, an overview of hardware components, and circuit diagrams of the input and output electronics. The computing platform employed by the controller is the PICMicro PIC18F452. This micro controller is ofter referred to as the "PIC" in the remainder of this document.

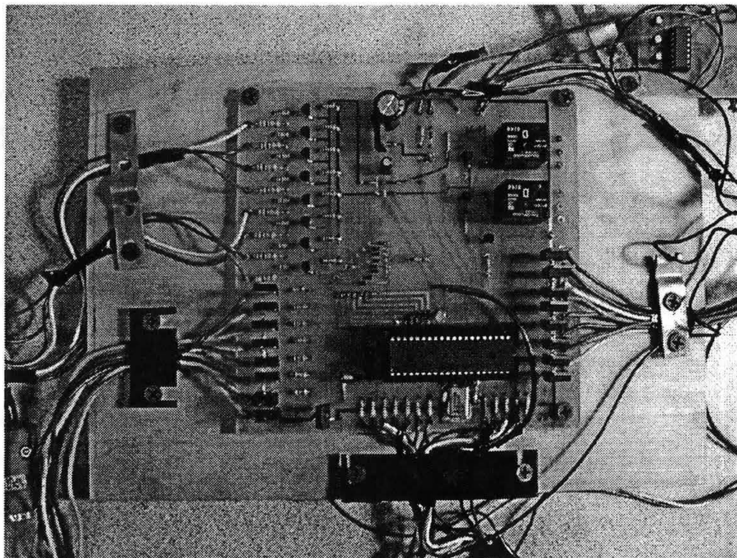


Figure 5.1: Controller Hardware Prototype

5.1 Port and Pin Assignments

This section provides the mapping of state space variables to ports and pins of the PIC. An "NA" in the first column indicates that the given row is not associated with a PIC port. In cases where a pin is not associated with a state space variable, the pin function is italicized.

Port Identifier	Pin Function	Pin Number
PORTA[0]	i_airInTemp	Pin 2
PORTA[1]	i_baroPres	Pin 3
PORTA[2]	i_o2Sensor	Pin 4
PORTA[3]	not used	Pin 5
PORTA[4]	i_fuelSelector	Pin 6
PORTA[5]	i_coolantTemp	Pin 7
PORTB[0]	$i_injector_1$	Pin 33
PORTB[1]	$i_injector_2$	Pin 34
PORTB[2]	$i_injector_3$	Pin 35
PORTB[3]	$i_injector_4$	Pin 36
PORTB[4]	$i_injector_5$	Pin 37
PORTB[5]	$i_injector_6$	Pin 38
PORTB[6]	$i_injector_7$	Pin 39
PORTB[7]	$i_injector_8$	Pin 40
PORTC[0]	o_nativeLED	Pin 15
PORTC[1]	o_stbyLED	Pin 16
PORTC[2]	o_altLED	Pin 17
PORTC[3]	i_programmingSwitch	Pin 18
PORTC[4]	o_heaterRelay	Pin 23
PORTC[5]	o_lockOffRelay	Pin 24
PORTC[6]	*serial transmit*	Pin 25
PORTC[7]	*serial Receive*	Pin 26
PORTD[0]	$o_altInjector_1$	Pin 19
PORTD[1]	$o_altInjector_2$	Pin 20

PORTD [2]	o_altInjector$_3$	Pin 21
PORTD [3]	o_altInjector$_4$	Pin 22
PORTD [4]	o_altInjector$_5$	Pin 27
PORTD [5]	o_altInjector$_6$	Pin 28
PORTD [6]	o_altInjector$_7$	Pin 29
PORTD [7]	o_altInjector$_8$	Pin 30
PORTE [0]	i_altFuelLevel	Pin 8
PORTE [1]	i_throttlePos	Pin 9
PORTE [2]	*native injectors ON/OFF*	Pin 10
NA	*reset*	Pin 1
NA	*power*	Pin 11
NA	*ground*	Pin 12
NA	*crystal*	Pin 13
NA	*crystal*	Pin 14
NA	*ground*	Pin 31
NA	*power*	Pin 32

5.2 Pin Diagram

The pin diagram, figure 5.2, presents much of the information previously provided by the pin assignment table. This time, the information is presented graphically. This figure was necessary for designing a controller hardware layout and traces as it provides information relating to the geography of PIC.

5.3 Hardware Overview

This section provides a graphical representation, figure 5.3, of the controller hardware design. The figure consists of hardware components represented as blocks, wires represented as thin black arrows, and buses represented as slashed and numbered

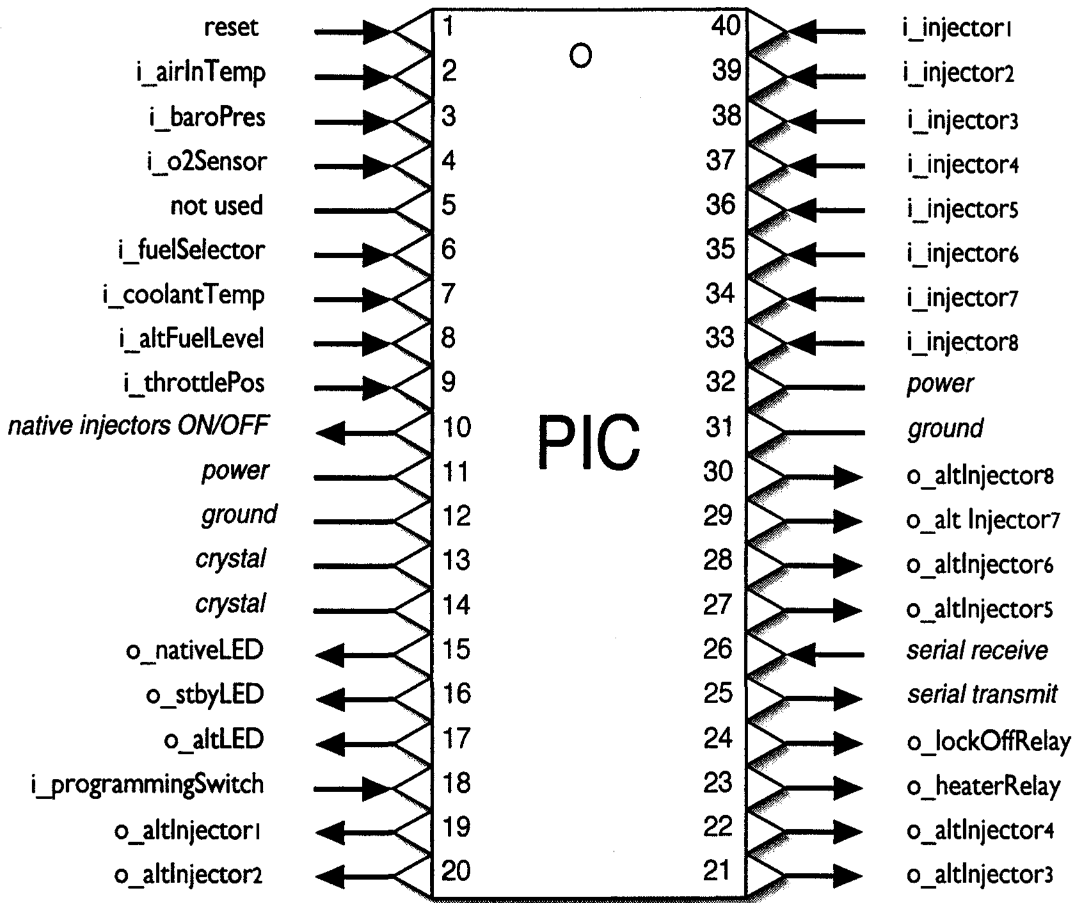


Figure 5.2: PIC Pin Diagram

arrows. The number that accompanies a bus correspond to the number of wires within that bus. Each hardware components is described below.

ACLA - Analogue Converter Level Adjustment

The PIC supports 12 bit analogue to digital conversion with predefined limits on voltage and amperage. It is the task of this hardware component to ensure the analogue signals are within these limits before arriving at the PIC.

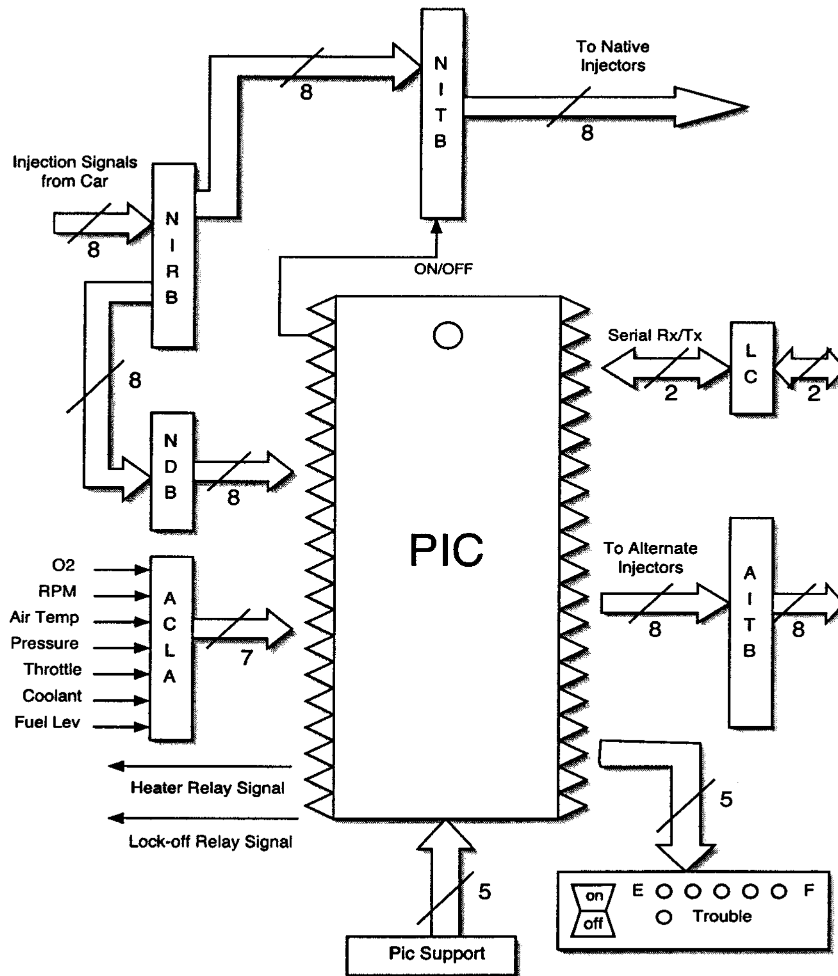


Figure 5.3: Hardware Overview

NIRB - Native Injector Resistor Bank

This hardware component serves two purposes. The first of these it to protect the PIC from the high current and voltage of incoming fuel injection pulses. Its second task, is to create a resistance on the input injection lines that is necessary to prevent an error from being registered by most automotive On-Board Computers.

NDB - Native Diode Bank

The Native Diode bank ands all incoming injector signals producing the $i_injector_8$ signal.

NITB - Native Injector Transistor Bank

This hardware component is a switchable array of transistors that when enabled allows incoming injection pulses to arrive unaltered to the native fuel injectors.

AITB - Alternate Injector Transistor Bank

This hardware component is an array of individually controlled transistors that provides fuel injection pulses to the alternate fuel injectors.

LC - Serial Level Converter

This hardware component adjusts the input and output signals of the PIC serial port as to make them consistent with the RS232C serial protocol.

PIC Support

The PIC support hardware component represents all hardware required for PIC operation. This includes access to 5 volt power, ground, and an oscillating crystal.

Driver Interface

The remaining hardware block is the driver interface. This hardware component consists of the fuel selection switch, the programming switch, and all LED's.

5.3.1 Hardware Components in the Prototype

Figure 5.4 is a modified version of the hardware prototype photo presented at the start of this chapter. Several hardware components discussed in this chapter have been labeled in the image to illustrate the correspondence between the the prototype and the overview.

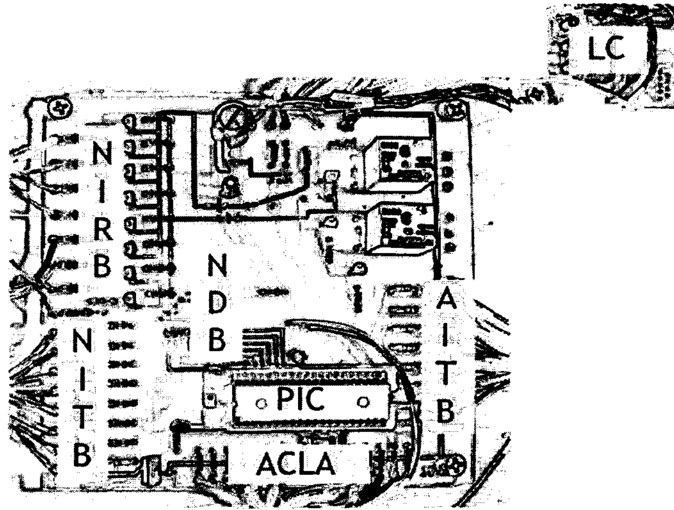


Figure 5.4: Hardware Overview

5.4 Hardware Circuit Diagrams

This section consists of four circuit diagrams. The first of these, figure 5.5, depicts one of the 8 elements of the native injector resistor bank (NIRB).[1] Figure 5.6 depicts

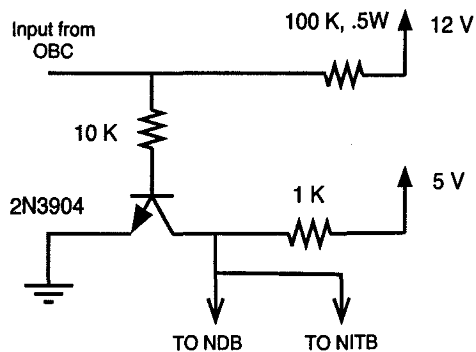


Figure 5.5: NIRB - Native Injector Resistor Bank

[1]The transistors labeled 2N3904 are general purpose PNP transistors. There are 8 such transistors used in the controller.

the native diode bank (NDB). The third, figure 5.7, depicts one of the 8 elements

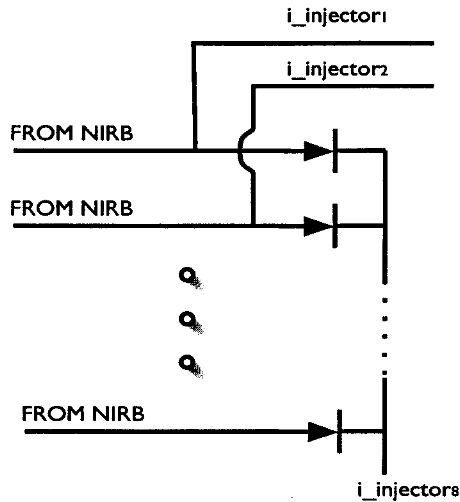


Figure 5.6: NDB - Native Diode Bank

of the native injector transistor bank. It also shows the PIC switched connection to ground that all 8 of these elements have in common. [2] The last circuit diagram,

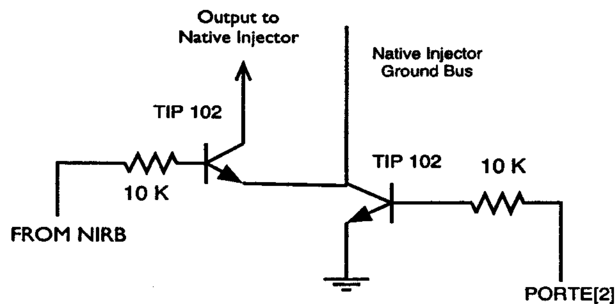


Figure 5.7: NITB - Native Injector Transistor Bank

figure 5.8, depicts one element of the alternate injector transistor bank.

[2]The transistors labeled TIP 102 are NPN Darlington Pair transistors capable of handling 100 volts at 15 amps for short periods of time. There are 8 + 8 + 1 such transistors used in the controller.

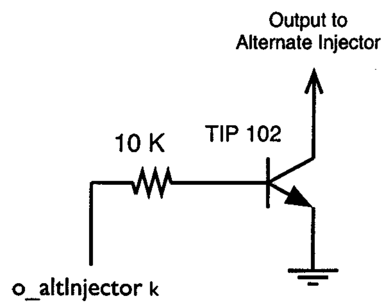


Figure 5.8: AITB - Alternate Injector Transistor Bank

Chapter 6

High Level Design

This chapter provides a structured software design for the alternate fuel injection controller. The design is presented using flowcharts.

6.1 Overview

The chapter begins with a description of the flow charting convention used, and continues with a series of flowcharts that comprise the design. Each flow chart is accompanied by a description and a list of results pertaining to safety, timing and functionality that can be inferred from it. The texts by Alan Shaw [1] and Frank Vahid and Tony Givargis [5] offered relavant discussions of real-time, embedded software design.

6.2 Flow Chart Conventions

The legend seen in Fig 6.1 lists the components and connectors that comprise a flowchart. This section describes each entity in Fig 6.1 in more detail to facilitate easy reading of this chapter.

- **Interrupt Branch** - The first item presented in Fig 6.1 is a connector labeled "Interrupt Branch". This connector is used to express a change in the program counter from anywhere in the source program segment to the beginning of an interrupt service request handler. It can also represent the return from an

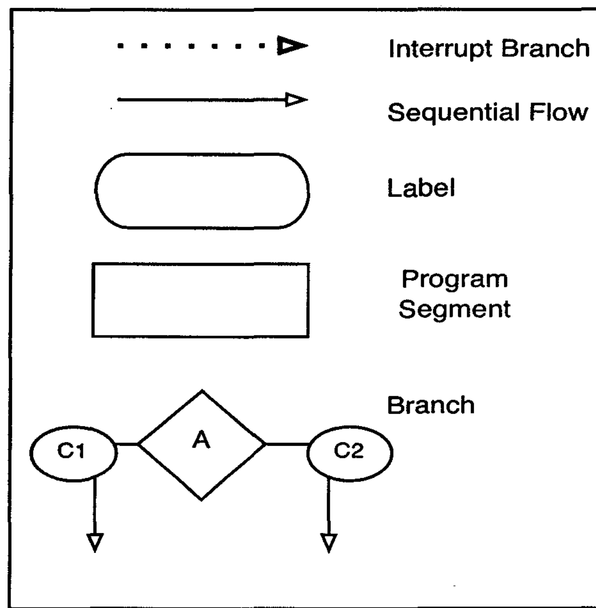


Figure 6.1: Flow Chart Legend

interrupt service request handler to the point in a program segment where an interrupt caused a jump in execution. As a convention, the code segment used to service interrupts will be labeled with ISR, for "Interrupt Service Routine". Any interrupt branch connectors ending at that segment represent a branch to the interrupt service routine, and any interrupt branch connectors starting at that segment represent a branch to the code at the value of the program counter before the interrupt was serviced.

- **Sequential Flow** - The sequential flow arrow is a connector used to express composition of program segments. When two items are joined by this connector it means that the item at the tail of the arrow is executed or evaluated immediately before the item at the head of the arrow (barring a jump to the interrupt service routine). When one of these connectors ends at another like connector, the expressed composition is between the source segment and the segment arrived at by following connectors in the direction of their arrow heads.

- **Label** - The label component is used at the beginning of all flow charts and at the end of any that describe a code segment with a returning execution path. They are used to indicate the starting and ending points of execution in the charts. The text inside these components are to be used as labels in the implementation assembly code.

- **Program Segment** - This component represents a collection of program instructions.

- **Branch** - This component is used to represent conditional branching in the program code. A is an expression that evaluates to an element of some type T. Each connector leaving the branch is labeled with a constant, or a collection of constants of the same type. For each possible value of A, $C_{0\cdots i}$ of type T, there is exactly one branch labeled with either the value itself or a collection containing that value. This construct dictates that if at some point during runtime, the program reaches the branch, execution will continue along the connector whose label is consistent with the value of the expression A.

6.3 General Program Structure

This section describes the overall flow of the software. The flowchart in Fig 6.2, begins with the start label "System Boot". The software will start at this point when the system is first powered up or when it is recovering from a *brown out*, or *software reset*. The missing end label indicates that the software will loop indefinitely. Other important conclusions that can be drawn from the structure of the flowchart are listed below.

- The program must completely execute the Initialization program segment before starting execution of the Main Loop, or Programming Loop program segments.

- On any given execution of the program, only one of the Main Loop or Programming Loop will be executed.

- Interrupts are disabled while the Programming Loop is being executed.

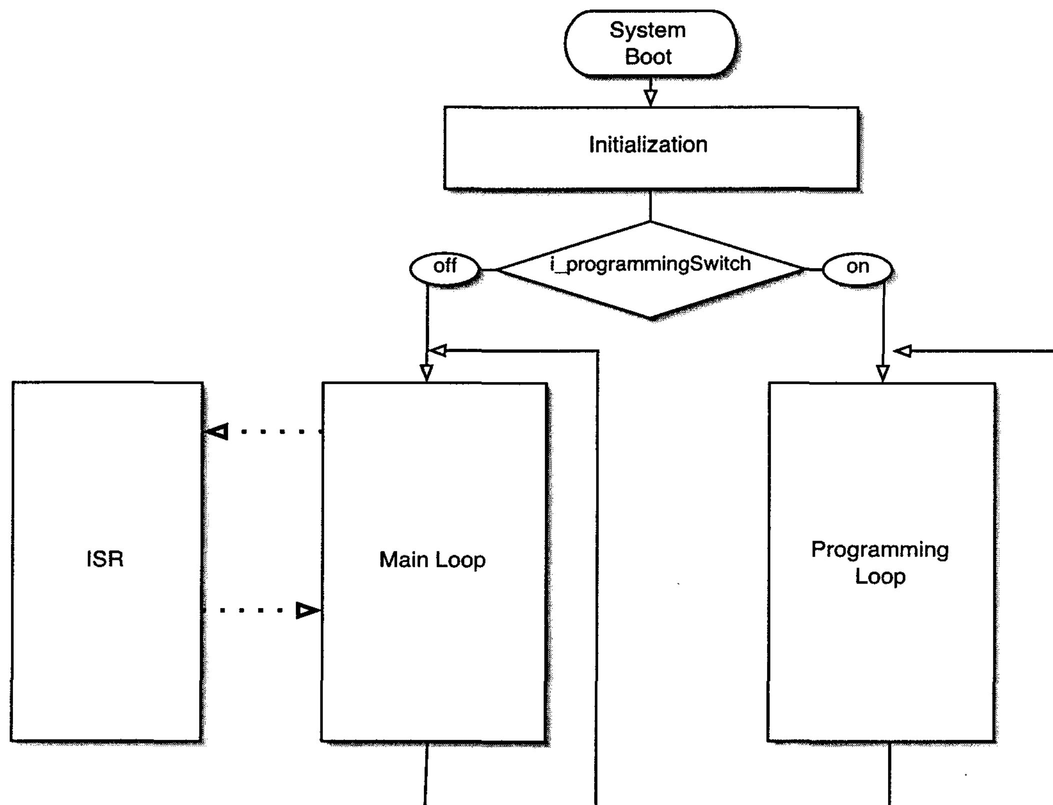


Figure 6.2: Program Structure Flowchart

- The value of i_programmingSwitch is checked only once and immediately after initialization. This is in keeping with our hold-on-boot, hidden switch design.

6.4 Initialization Program Segment

This section describes the controller's initialization routine as it is presented in Fig 6.3. Initialization entails writing to a subset of the micro controller's special function registers. This is done in order to initialize the hardware to perform the functionality that the control task requires. A specific example of this, is setting the A/D control register so that the micro controller uses 5 volts as a reference voltage. Another is

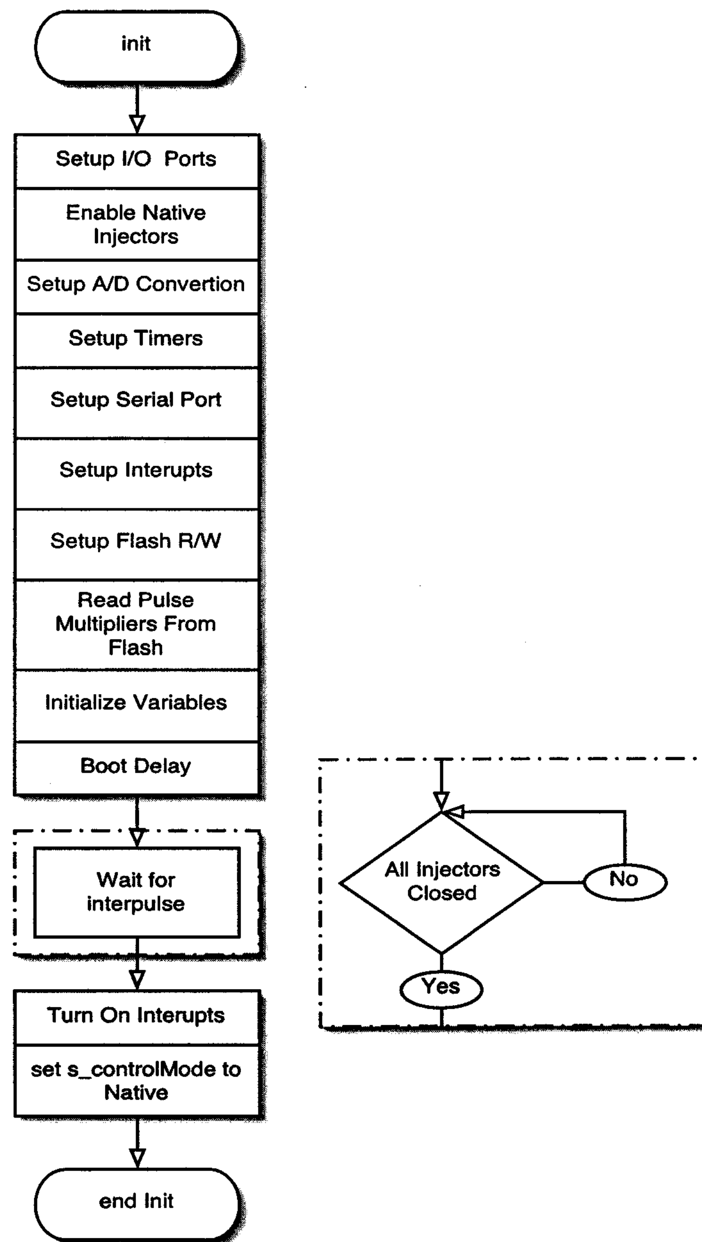


Figure 6.3: Initialization Flowchart

writing to the baud rate generator special function register to set the serial port baud rate to be 9600 bps. All of these tasks will be described fully in the chapter relating to implementation. Conclusions that can be drawn from the structure of the flowchart are listed below.

- The native injectors are enabled, and remain enabled during initialization. As a result the vehicle will be injecting the native fuel when it enters both the Main Loop and Programming Loop program segments.
- Interrupts are only enabled after all other initialization tasks are completed. This guarantees that the system will never enter the ISR before the system is prepared to time and transmit injection pulses.
- Since we have only one segment of the initialization code that branches, and we have a hard time bound for how long the program can remain in that segment (the maximum native injection length), we have a hard time bound for how long initialization will take.

6.5 Mode Update and Data Acquisition Main-Loop

This section describes the structure of the main control loop as it is presented in Fig 6.4. The main loop is the program segment that deals with setting the control mode, as well as data acquisition and serial transmission. The loop begins by checking i_programmingSwitch that is used to notify the system of which fuel is desired by the user for injection. If the native fuel is desired, the controller has nothing else to verify and proceeds to take the necessary action of setting s_controlMode to native, and enabling the native injectors. Next the user interface LED's are updated and o_heaterRelay and o_lockOffRelay are set to *inactive*. If the switch indicates that the alternate fuel is desired for injection, the controller branches depending on which fuel is being injected at that time. This is necessary as the requirements for switching to alternate injection are more stringent than those for continuing alternate injection.

If the value of s_controlMode is *native*, the controller reassigns it with the value stbyNat, sets both o_heaterRelay and o_lockOffRelay to *active*, updates the user in-

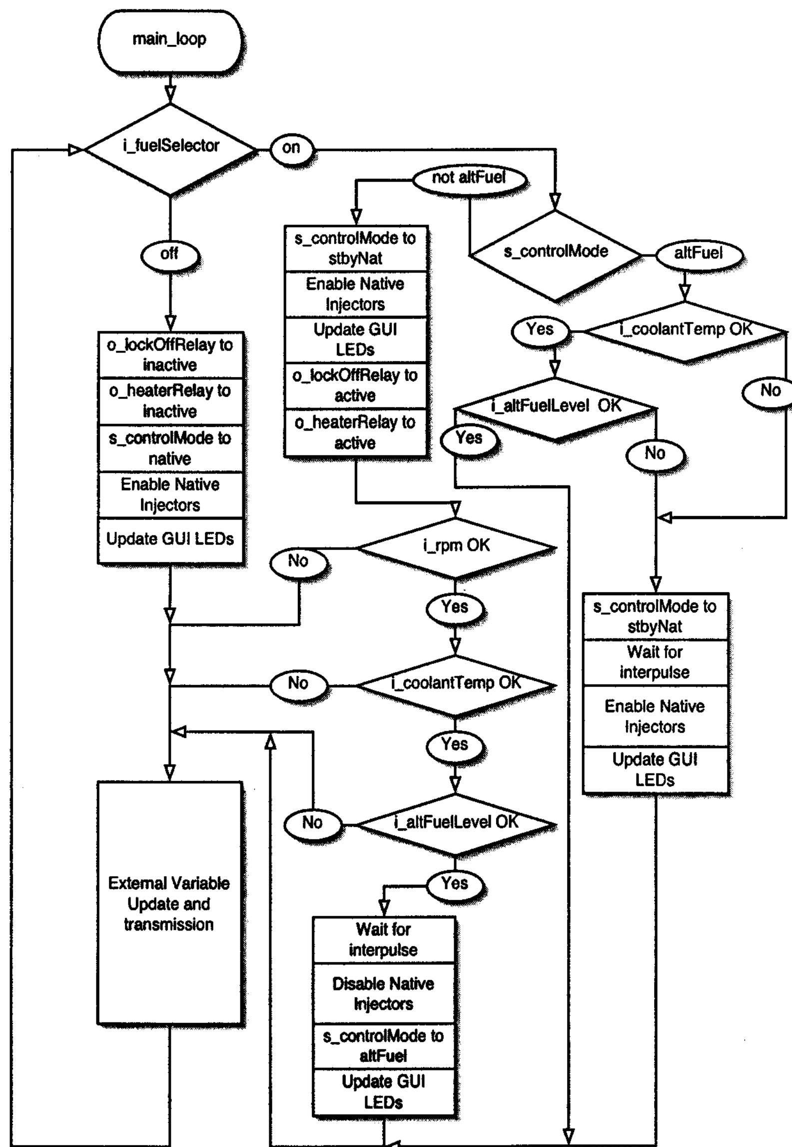


Figure 6.4: Main-Loop Flowchart

terface, and then determines if the values of the monitored variables allow for the transition to alternate fuel. If they do, s_controlMode mode is set to altFuel and the native injectors are disabled. If the system was already injecting the alternate fuel, it

checks that the values of the monitored variables allow for that to continue, in which case no further action is taken. If, however, the values of the monitored variables are such that a return to native injection is required, s_controlMode is set to stbyNat, the native injectors are enabled and the GUI LED's are updated. In each of the above mentioned situations, the next step is the same, update and transmit the values of the state variables that represent the analogue inputs. The process then repeats. A list of conclusion that can be drawn from the structure of the main loop are listed below.

- Every time the i_fuelSelector branch forces execution along the path labeled off, the values of s_controlMode, o_lockOffRelay, and o_heaterRelay are updated.

- i_rpm is not a factor in determining if alternate injection should continue, but is a factor in determining if alternate injection should begin.

- There are no sub-loops in this flowchart, which would seem to imply that there is a hard time bound for the execution of the main loop. However, Figure 6.2 dictates that interrupts can cause the program counter to jump from this program segment, and figure 6.6 shows that the injector enable bit is only updated between incoming pulses. As such, timing results cannot be given without first analyzing the interrupt service routine and the behavior of the system at times of mode transitions.

6.5.1 External Variable Update and Transmission Sub-Chart

Figure 6.5 is used to give a detailed description of the "Variable Update and Transmission" code segment that appears near the bottom of Fig 6.4. The "7X" found at the left of the flowchart describes how the entire block is performed seven times before execution leaves the code segment. The code does not loop but rather is repeated once for each analogue channel. The segment starts by performing A/D conversion. The digital result is then collected and stored in the appropriate register. Finally, the result is transmitted over the serial port.

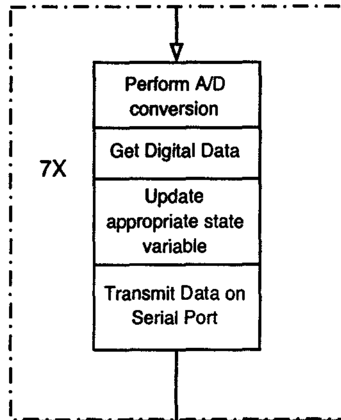


Figure 6.5: External Variable Update and Transmit

6.5.2 Enable/Disable Native Injectors Sub Chart

Figure 6.6 is used to give a detailed description of the "Enable Native Injectors" and "Disable Native Injectors" code segments. They illustrate how the state of the native injector enable bit is only updated when it is not consistent with the current mode, and how it is only updated when there are no incoming injection pulses.

- All analogue values are collected, recorded and transmitted over the serial port each time through the MainLoop code segment.

6.6 Interrupt Service Request Handler

This section describes how interrupts are handled by the system and will refer to Fig 6.7. In this design, there are three conditions that when detected, cause the program counter to jump to the interrupt service request handler. These conditions are $\neg(\text{i_injector}_{k,k=1\cdots8} = \textit{Previous_Value}(\text{i_injector}_{k,k=1\cdots8}))$, TIMER 1 overflow flag bit $= 1$, and TIMER 2 period match flag bit $= 1$. The first task in the ISR is determining which of the three conditions was detected. This is reflected in the flowchart by the first branch element. Each execution path leading from this branch will be discussed in its own sub-section.

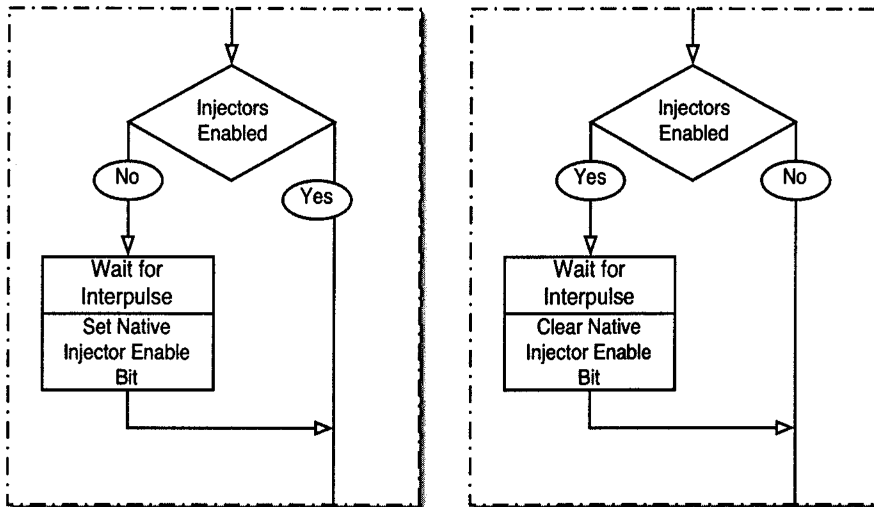


Figure 6.6: Enable/Disable Native Injectors

6.6.1 TIMER 1 Overflow

TIMER 1 is used to measure the duration of incoming pulses. Each time a new pulse is detected by the controller, TIMER 1 is reset and started. When the end of an incoming pulse is detected, the timer is stopped. TIMER 1 has been configured in such a way that the amount of time before a rollover, is greater than the longest possible incoming pulse length. An overflow of TIMER 1 indicates that a pulse has arrived at the controller that was longer than any intended to be handled by the controller. The detection of this condition implies that the event *non_term_pulse* has occurred and, therefore, the system must set s_trouble to *on*. In this situation, all alternate injectors are closed, native injectors are enabled, both o_heaterRelay and o_lockOffRelay are assigned the value *inactive*, s_controlMode is assigned the value native and the user interface LEDs are updated. Once again, conclusions that can be drawn from this flowchart are as listed here.

- After a TIMER 1 overflow, fuel injection is performed exclusively by the native injectors.
- After a TIMER 1 overflow, o_lockOffRelay and o_heaterRelay have the value

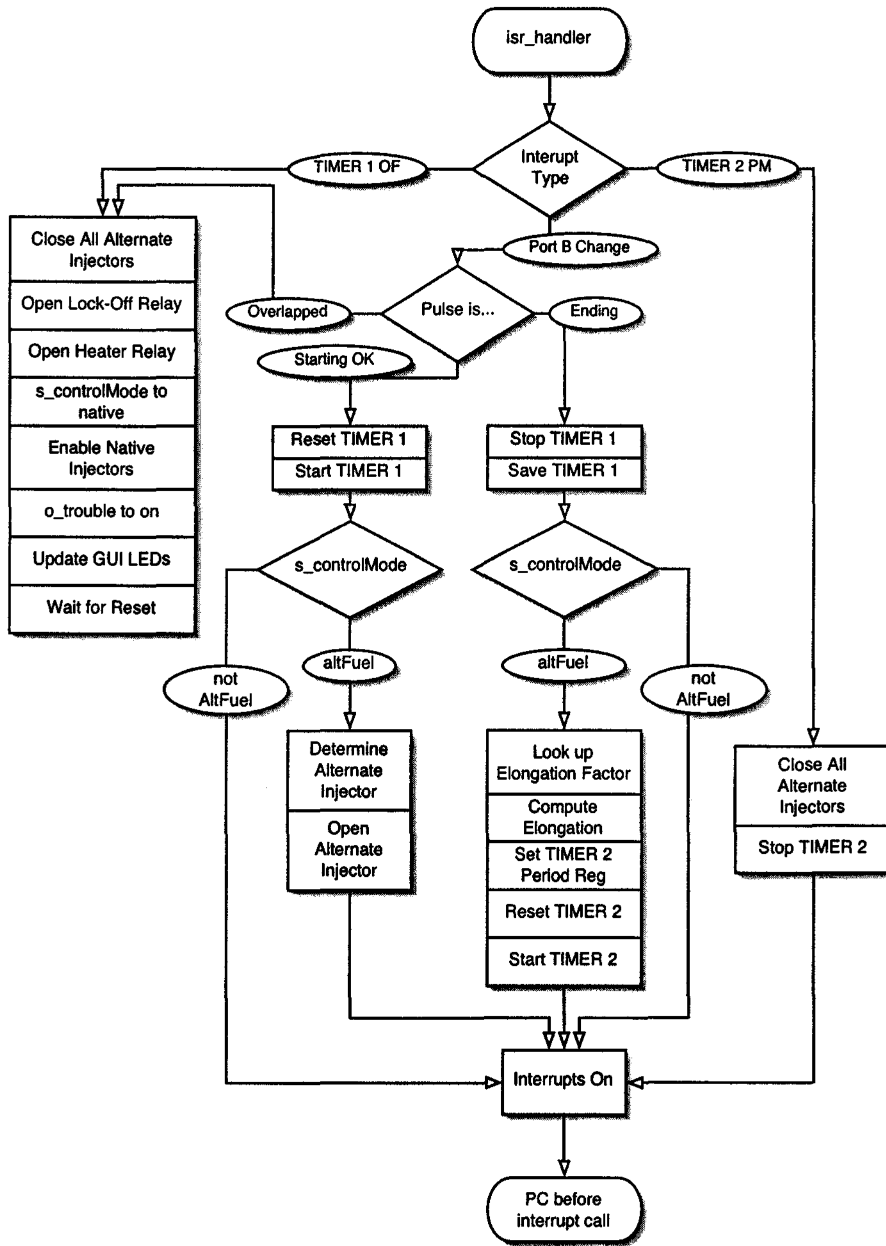


Figure 6.7: Interrupt Flowchart

inactive.

- After a TIMER 1 overflow, s_trouble has the value *on*.

6.6.2 Change on Port B

Port B is connected to the native injection signal lines. The change on port B interrupt occurs whenever the controller detects the start or end of a native injection pulse. The detection of an incoming pulse before the completion of a previous pulse implies that the event *elong_over_pulse* has occurred. In this situation, execution continues in the code segment described in the TIMER 1 overflow sub-section. If a native injection pulse is detected, and there is no overlap, TIMER 1 is started in order to measure the duration of the pulse. Then, if s_controlMode has the value altFuel, the appropriate alternate injector is determined and opened. If the pulse is ending, the value in the TIMER 1 register is saved and TIMER 1 is stopped. If s_controlMode has the value altFuel, the length of time by which to elongate the native pulse is determined and the corresponding value is moved to the period register of TIMER 2. TIMER 2 is then reset and started. Change on Port B flowchart implications are listed here.

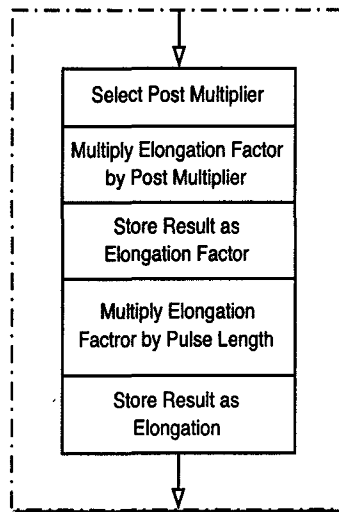


Figure 6.8: Compute Elongation - Sub-Chart

- After detecting overlapping pulses, fuel injection is performed exclusively by the native injectors.
- After detecting overlapping pulses, o_lockOffRelay and o_heaterRelay have the value *inactive*.
- After detecting overlapping pulses, s_trouble has the value *on*.
- If TIMER 1 is running, the value in the TIMER 1 register is equal to the amount of time that a native pulse has been present on Port B minus a delay proportional to the speed of the micro controller.
- If TIMER 1 is stopped, its value is the length of the most recent injection pulse minus a delay proportional to the speed of the micro controller.
- When s_controlMode has the value altFuel, the start of alternate injection pulses coincide with the detection of native injection pulses.
- Regardless of the value of s_controlMode, the lengths of native injection pulses are measured.

6.6.3 Timer 2 Match

TIMER 2 is used to size pulse elongations. A TIMER 2 match interrupt occurs when the value of TIMER 2 is equal to the value stored in the TIMER 2 period register. This interrupt indicates that the elongation has reached its specified duration at which point all alternate injectors are closed, and TIMER 2 is stopped.

6.6.4 Interrupt Race Conditions

Provided that the event *elong_over_pulse* does not occur, this design will not lead to a situation where two interrupts can occur at once, or in the wrong order. A TIMER 1 overflow means that there has been an uninterrupted pulse on Port B for an extended length of time. This obviously can not occur at the same time as a change on Port B. Also, if TIMER 2 is started, then TIMER 1 is stopped and cannot overflow.

6.7 Programming Program Segment

This section describes Fig 6.9, which is a more detailed view of the code segment labeled Programming Loop on Fig 6.2. The programming code segment has no exit path and cannot be interrupted. Execution will remain in the code segment until a reset occurs. The programming strategy is quite simple. Set the start address, and then continue to check the serial port for incoming data. Data is collected 2 bytes at a time. The first byte represents the value to be written to the flash ram, the second byte represents whether the data is being transmitted for the first time, or if it is being retransmitted as a result of an error on a previous attempt. If the value is new, the address into the flash ram is incremented by one before the data is written, otherwise, the data is written in place. In either case, the data is echoed before the serial port is checked again.

6.8 PC Programming Segment

This section describes a design for code that will reside on a personal computer and not on the controller. The flowchart being described is Fig 6.10. The PC application transmits data to the controller in two byte blocks. The first of these is a value byte, and the second indicates if the value is being transmitted for the first time, or if it is being retransmitted. Once these 2 bytes have been transmitted, the program checks the data echoed by the controller to determine the result of the most recent communication. There are three possible communication results:

- Success - This result indicates that the value byte was correctly echoed by the controller, and the echoed retransmit byte was dominated by 1's, or 0's, appropriately. In this case, the next value from the fuel map file will be the value byte of the next transmission, and the retransmit byte will have the value 255 (decimal).
- Recoverable - This result indicates that the controller's echo of the value byte was incorrect. However, the retransmit bye was dominated by the correct bit value. In this situation, the next transmission is composed of the same value byte and a retransmit byte of '0'.

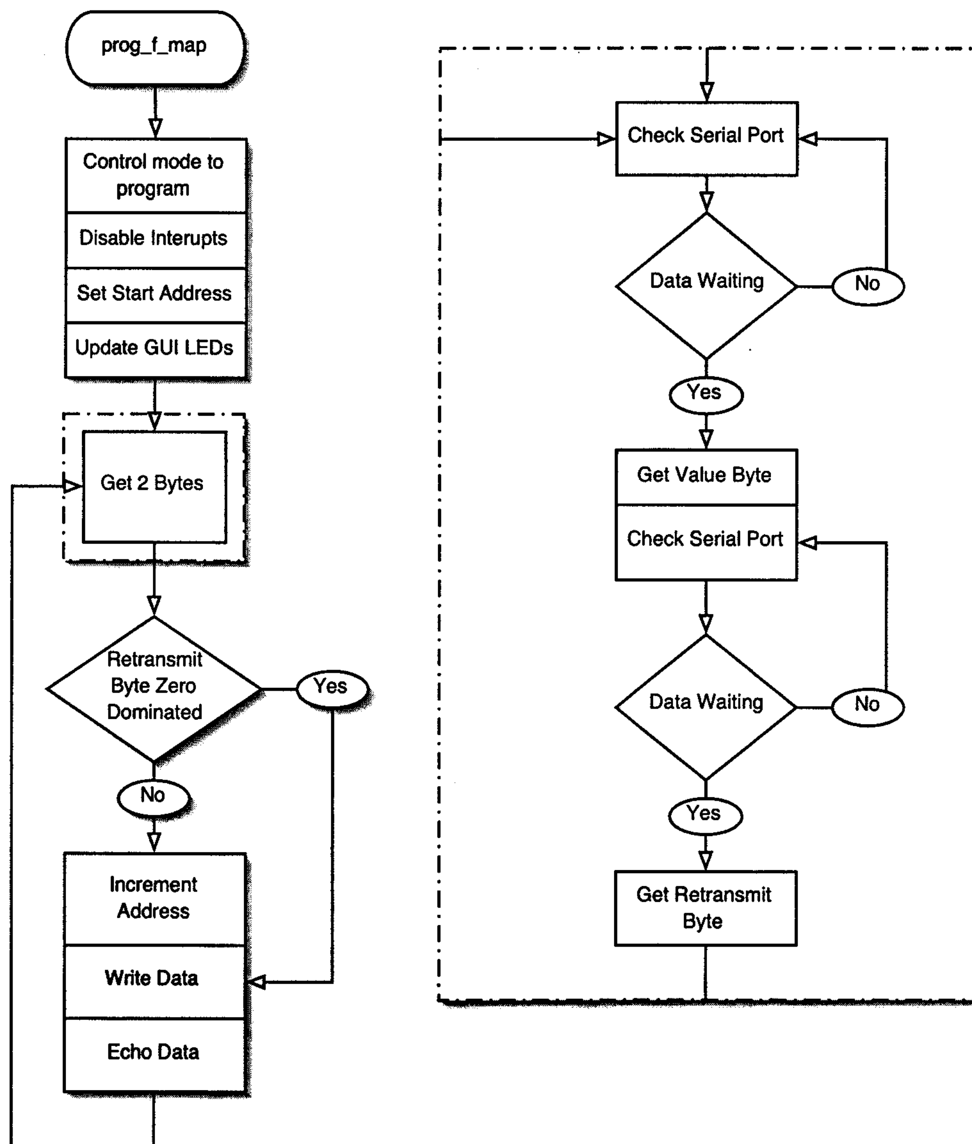


Figure 6.9: Programming Flowchart - PIC Side

- Unrecoverable - This result indicates a failure to correctly echo the retransmit byte with even 50% accuracy. Transmission ends, and the program informs the user that programming must be restarted.

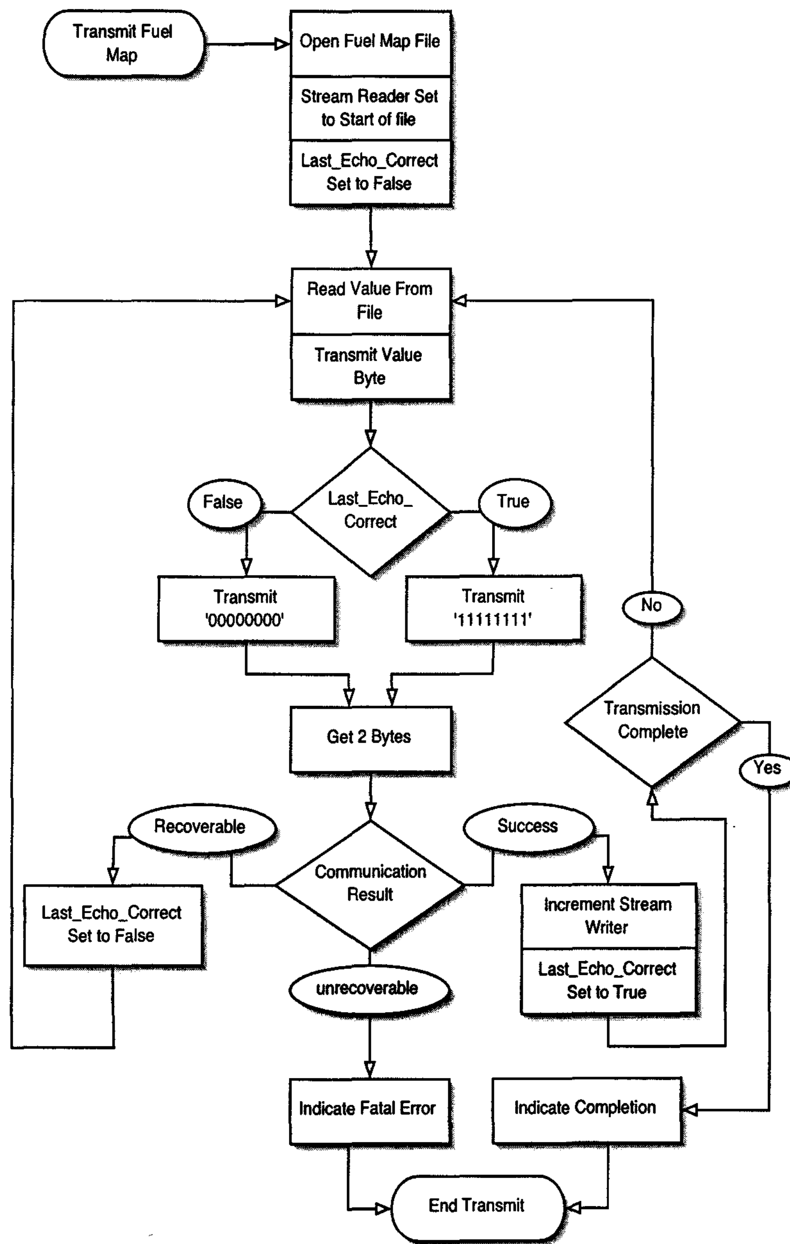


Figure 6.10: Programming Flowchart - PC Side

If no unrecoverable error is encountered, programming ends when the last value in the fuel map is successfully transmitted.

Chapter 7

Implementation Description

7.1 Overview

This chapter provides an implementation of the Fuel Injection Controller software component. The platform is the PIC 18F452 micro-controller. The chapter begins with a section on PIC configuration, which deals with the setting of PIC configuration bits as well as timing requirements and the implications they have on PIC setup. Next is a description of the coding conventions adhered to during code development. This is followed by a discussion of a mapping used to relate state variables in the design to approximations of the state variables in the runtime environment. The remaining sections provide unit specific information, including a list of assumptions, lists of variables, and the PIC micro assembly implementation code. Variables are presented in as many as three separate lists, one for variables that are global and are read but not changed, another for those that are global and are changed, and a third for those that are local to the software unit. These lists are omitted only when empty. Each variable name is accompanied by a short description and the letters "SF" or "GP". All variables accompanied by "SF", are special function registers of the PIC, while those accompanied by "GP" are general purpose registers with user defined identifiers. Code development was supported by each of [7, 8, 2, 10, 9].

7.2 PIC Configuration

This section provides the settings of the PIC configuration and timer setup bits.

7.2.1 Configuration Bits

The configuration bits are used to configure the system oscillator, the watchdog timer, brown out detection and other PIC facilities. These bits exist in program memory and are only accessible to PIC software via flash reads. Listed here are the configuration bit settings that are used in conjunction with the implementation code presented later in this chapter. Correct controller operation depends on proper PIC configuration.

- **Oscillator** - HS - PLL- Enabled
- **Osc Switch Enable** - Disabled
- **Power Up Timer** - Enabled
- **Brown Out Detect** - Enabled
- **Brown Out Voltage** - 4.5 Volts
- **Watchdog Timer** - Disabled, controlled by SWDTEN Bit
- **Watchdog Post Scaler** - 1:128
- **CCP2 Mux** - Rc1
- **Stack Overflow Reset** - Enabled
- **Low Voltage Program** - Disabled
- **Code Protect** - Disabled
- **Table Write Protect** - Disabled
- **Data EE Write Protect** - Disabled
- **Table Write Protect Boot** - Disabled
- **Config Write Protect** - Disabled
- **Table Read Protect** - Disabled
- **Table Read Protect Boot** - Disabled

7.2.2 Timers

This section provides timing information needed to successfully implement the fuel injection controller using a PIC18F452 micro-controller. Timer configurations are chosen based on the upper and lower limits of engine RPM, maximum fuel injector duty cycles, and the frequency of oscillation (FOSC) of the micro-controller. Discussion will be limited to 8 cylinder engines as this is the engine format of the test vehicle.

TIMER 1

Timer 1 is a 16 bit timer with a frequency of (FOSC / 4) that can be pre-scaled by 1, 2, 4 or 8. Timer one is used to measure the length of incoming injection pulses. More detail on TIMER 1 can be found in the PIC18Fxx2 data sheet [10].

TIMER 2

Timer 2 is an 8 bit timer also with a frequency of (FOSC / 4) that can be both pre and post-scaled by 2, 4, 8 or 16. Timer 2 is used to determine when to terminate alternate pulse elongations. More detail on TIMER 2 can also be found in the PIC18Fxx2 data sheet [10].

Timing Results of RPM Requirements

The following is a list of excerpts from the controller requirements specification that deal with values of RPM and have implications for timing and timer settings.

- The controller shall be capable of detecting and generating pulses to keep an 8 cylinder engine running at up to 6500 RPM.
- The controller shall be capable of detecting and generating pulses to keep an 8 cylinder engine running at as few as 500 RPM.

Based on these numbers and a basic understanding of an engine revolution, the maximum possible length of a fuel injection pulse can be determined. In the engines we are concerned with, every cylinder has a single fuel injector, and each injector

is opened once for every two revolutions of the engine. Therefore, in an 8 cylinder engine, there will be 4 injections/revolution.

- 6500 is the maximum required RPM value.
- At 6500 RPM, there are approximately 109 revolutions per second.
- These correspond to approximately 434 injections per second,
- Yielding a maximum injection window of 2.3 milliseconds at 6500 RPM

At the lower RPM Limit, we have the following results.

- 500 is the minimum required RPM value.
- At 500 RPM there are approximately 8 revolutions per second.
- These correspond to approximately 32 injections per second,
- Yielding a maximum injection window of 31.2 milliseconds at 500 RPM.

These values represent the maximum time fuel injectors would remain open based on a full duty cycle (the situation where one fuel injector is always open). However, the duty cycle may not be 100% or else there would not be sufficient time for pulse elongation. The maximum recommended duty cycle for most fuel injectors does not exceed 80%. This provides a more realistic timing of

- 1.84 milliseconds per injection at 6500 RPM
- 25 milliseconds per injection at 500 RPM

TIMER 1 Configuration

Based on these numbers and the 80% duty cycle assumption, TIMER 1 must resolve a range from zero to 25 milliseconds. The ideal configuration for TIMER 1 when the PIC is running at 40MHz is described below.

- @40 Mhz (FOSC)

- TIMER 1 maximum frequency is 10 Mhz or (FOSC / 4)
- 1 tick per 100 ns
- 65 536 ticks before rollover
- 6.5536 milliseconds to rollover

When prescaled by 4, at 40 MHz, this gives a time to rollover of 26.2132 milliseconds which is safely above our maximum pulse length of 25 milliseconds. If, however, the duty cycle of the injectors exceeds 80%, the operating range between 500 and 578 RPM may be problematic.

TIMER 2 Configuration

The maximum length of a pulse elongation was not specified. For this reason, calculations are provided which relate to longest possible pulse elongation that can be produced using timer 2 with a FOSC of 40 MHZ.

- @40 Mhz (FOSC)
- TIMER 2 maximum frequency is 10 Mhz or (FOSC / 4)
- 1 tick per 10-7 seconds
- 256 ticks before rollover
- 2.56e-05 seconds to rollover

Pre-scaling and post-scaling TIMER 2 by 16 at 40 Mhz gives a time to rollover of 6.5536 milliseconds. At low RPM values, this elongation represents at least 25% of the incoming pulse length. This percentage increases along with RPM.

7.3 Coding Conventions

This section provides a list of conventions that have been followed during production of the implementation code. These include conventions to increase readability, and those that are necessary to ensure safety and correctness.

- PIC Micro Assembly instructions are written in upper case letters.
- Labels are written in lower case letters.
- Variables are assigned addresses starting from 0x01 and occupy a contiguous block of memory.
- Variable identifiers consist only of letters, numbers, and underscore characters.
- Identifiers of variables local to a software unit are written in lower case letters and start with that unit's 3 character prefix.
- Global variables are written in only upper case letters.
- Global variables identified as being input or output variables exclusive to a unit, are prefixed with that unit's 3 character prefix.
- GOTO instructions, external to the interrupt service routine, must not cause execution to jump to a location inside the interrupt service routine.
- The size of the call stack must not exceed 31 at any time during execution.
- When the value of a register is set by directly moving in an 8-bit value, the result of the move is described with an in-line comment.
- Explicit banking is not permitted.

7.4 Requirement Variable PIC Representations

This section describes how state space variables identified in the specification and design chapters are represented in the runtime environment. This is done using an function referred to as the $\mathcal{L}$ mapping, which relates values of the state space variables to values in the PIC micro controller.

Using the $\mathcal{L}$ mapping, variables are associated with registers and bits in the PIC micro controller. Registers are referenced by their identifiers and not by their addresses. Square brackets enclosing a number between 0 and 7, or an expression evaluating to the same range are used to specify specific bits of a register. The following

Table 7.1: $\mathcal{L}$ Mapping Definition

Type	Abstraction Function $\mathcal{L}$
$\{closed, open\}$	$\mathcal{L}(closed) = 0$
	$\mathcal{L}(open) = 1$
$\{off, on\}$	$\mathcal{L}(off) = 0$
	$\mathcal{L}(on) = 1$
$\{down, up\}$	$\mathcal{L}(down) = 0$
	$\mathcal{L}(up) = 1$
$\{inactive, active\}$	$\mathcal{L}(inactive) = 0$
	$\mathcal{L}(active) = 1$
$\{\mathsf{native}, \mathsf{stbyNat}, \mathsf{altFuel}, \mathsf{init}, \mathsf{program}\}$	$\mathcal{L}(\mathsf{native}) = 0$
	$\mathcal{L}(\mathsf{stbyNat}) = 1$
	$\mathcal{L}(\mathsf{altFuel}) = 2$
	$\mathcal{L}(\mathsf{init}) = 4$
	$\mathcal{L}(\mathsf{program}) = 8$
$\{\mathsf{interpulse}, \mathsf{pulse}, \mathsf{elongation}\}$	$\mathcal{L}(\mathsf{interpulse}) = 0$
	$\mathcal{L}(\mathsf{pulse}) = 1$
	$\mathcal{L}(\mathsf{elongation}) = 2$
volts	$\mathcal{L}(x) = Round(51x)$
real	$\mathcal{L}(x) = fixedpoint(x)$

table lists the mapping from state space variables in the requirements document to registers names in the PIC micro assembly code. The result of applying the $\mathcal{L}$ mapping to the state space variable in the first column, is stored in the register or bits listed in the second column.

Variable Name	Concrete Register Identifier or Value
$\mathsf{i_injector_k}, \mathsf{k} = 1 \cdots 7$	PORTB[k − 1]
i_fuelSelector	PORTA[4]
i_programingSwitch	PORTC[3]
i_coolantTemp	COOLANT_TEMP
i_o2Sensor	OXYGEN
i_baroPres	BARO

i_airInTemp	AIR_IN_TEMP
i_altFuelLev	ALT_FUEL_LEV
i_rpm	REVS
i_throttlePos	THROTLE_POS
i_o2Thresh$_1$	IR_02_THRESH_L
i_o2Thresh$_2$	IR_02_THRESH_LM
i_o2Thresh$_3$	IR_02_THRESH_HM
i_o2Thresh$_4$	IR_02_THRESH_H
i_tpThresh$_1$	IR_TP_THRESH_L
i_tpThresh$_2$	IR_TP_THRESH_LM
i_tpThresh$_3$	IR_TP_THRESH_HM
i_tpThresh$_4$	IR_TP_THRESH_H
i_o2Mult$_1$	IR_02_MULT_LL
i_o2Mult$_2$	IR_02_MULT_L
i_o2Mult$_3$	IR_02_MULT_LH
i_o2Mult$_4$	IR_02_MULT_HH
i_tpMult$_1$	IR_TP_MULT_LL
i_tpMult$_2$	IR_TP_MULT_L
i_tpMult$_3$	IR_TP_MULT_H
i_tpMult$_4$	IR_TP_MULT_HH
o_altinjector$_k$, $k = 1 \cdots 8$	PORTD[k − 1]
o_heaterRelay	PORTC[4]
o_lockOffRelay	PORTC[5]
s_trouble	TROUBLE
o_nativeLED	PORTC[0]
o_stbyLED	PORTC[1]
o_altLED	PORTC[2]
s_controlMode	MODE

In these tables, column headers include conditions which determine how state space variables are mapped to registers or values in the micro assembly code. If the condition atop a column is true, the value of the $\mathcal{L}$ mapping applied to the state space variable can be found in that column.

	Concrete Register Identifier or Value	
Variable Name	$\neg$(PORTB = 128)	PORTB = 128
$\mathsf{i_injector_8}$	0	1

	Concrete Register Identifier or Value	
Variable Name	PORTE[2] = 0	PORTE[2] = 1
$\mathsf{o_natInjector_k}, \mathsf{k} = 1 \cdots 7$	0	PORTB[k − 1]

	Concrete Register Identifier or Value	
Variable Name	$\neg$(PORTE[2] = 1 $\wedge$ PORTB = 128)	PORTE[2] = 1 $\wedge$ PORTB = 128
$\mathsf{o_natInjector_8}$	0	1

	Concrete Register Identifier or Value		
Abstract Variable	(T1CON[0] = 0) $\wedge$ (T2CON[0] = 0)	(T1CON[0] = 1) $\wedge$ (T2CON[0] = 0)	T2CON[0] = 1
s_injectionStatus	0	1	2

7.5 Initialization

7.5.1 Assumptions

- Jumps to instructions inside the Initialization routine that do not originate from within the initialization routine will not occur.

7.5.2 Variables

This section lists the global and local variables that are changed during the execution of the initialization sub-routine. The 3 character prefix for the initialization sub-routine is IN_.

Global Variables Changed - Initialization

Variable Name	Type & Description
WREG	SF - Operations Register

STATUS	SF - Operational Status Register
ADCON0	SF - A/D Control Register 0
ADCON1	SF - A/D Control Register 1
TRISA	SF - Port A Data Direction Register
TRISB	SF - Port B Data Direction Register
TRISC	SF - Port C Data Direction Register
TRISD	SF - Port D Data Direction Register
TRISE	SF - Port E Data Direction Register
PORTB	SF - Port B
PORTC	SF - Port C
PORTD	SF - Port D
PORTE	SF - Port E
T1CON	SF - Timer 1 Control Register
T2CON	SF - Timer 2 Control Register
TMR2	SF - Timer 2 Value
PR2	SF - Timer 2 Period
SPBRG	SF - Serial Baud Rate Generator
TXSTA	SF - Serial Transit Setup Register
RCSTA	SF - Serial Receive Setup Register
PIR1	SF - Peripheral Interrupt Register
RCON	SF - Reset Control Register
INTCON	SF - Interrupt Control Register
INTCON2	SF - Interrupt Control Register
PIE1	SF - Peripheral Interrupt Enable Register
AIR_IN_TEMP	GP - Stores Converted Value
BARO	GP - Stores Converted Value
OXYGEN	GP - Stores Converted Value
REVS	GP - Stores Converted Value
COOLANT_TEMP	GP - Stores Converted Value
ALT_FUEL_LEV	GP - Stores Converted Value
THROTTLE_POS	GP - Stores Converted Value

DELAY	GP - Counter Used to Wait for A/D Acquisition
MODE	GP - Stores Control Mode of System
FL_HIGH_ADR	GP - High Bits of Flash Address for Write
FL_READ_ADDR_H	GP - High bits of Flash Address for Read
FL_READ_ADDR_L	GP - Low bits of Flash Address for Read
FL_READ_VAL	GP - Register for Storing Results of Flash Reads
IR_02_THRESH_L	GP - Low O2 Threshold Value
IR_02_THRESH_LM	GP - Low Mid O2 Threshold Value
IR_02_THRESH_HM	GP - High Mid O2 Threshold Value
IR_02_THRESH_H	GP - High O2 Threshold Value
IR_TP_THRESH_L	GP - Low TPS Threshold Value
IR_TP_THRESH_LM	GP - Low mid TPS Threshold Value
IR_TP_THRESH_HM	GP - High mid TPS Threshold Value
IR_TP_THRESH_H	GP - High TPS Threshold Value
IR_02_MULT_LL	GP - Low Low Range O2 Multiplier
IR_02_MULT_L	GP - Low Range O2 Multiplier
IR_02_MULT_H	GP - High Range O2 Multiplier
IR_02_MULT_HH	GP - High High Range O2 Multiplier
IR_TP_MULT_LL	GP - Low Low Range TPS Multiplier
IR_TP_MULT_L	GP - Low Range TPS Multiplier
IR_TP_MULT_H	GP - High Range TPS Multiplier
IR_TP_MULT_HH	GP - High High Range TPS Multiplier
TROUBLE	GP - Stores Trouble State

Local Variables - Initialization

Variable Name	Type & Description
in_boot_delay	GP - Counter for timing boot delay

7.5.3 Initialization Code

```
1   ;******************* SETUP AND INITIALIZATION *********************;
    main
    init
4       CLRF    MODE            ;
        BSF     MODE, 2         ; Mode to init
        CLRF    ADCON0          ; All analogue input code
        BSF     ADCON0, ADCS1   ; AD clock set to fosc/32
8       BSF     ADCON0, ADON    ; AD unit on

        CLRF    ADCON1          ; left justified-all analogue in-fosc/32

12      MOVLW   0x7f            ; low 5 pins input code for portA
        MOVWF   TRISA           ; into the portA control register
                                ; set portA pin 6 to be an output pin,

16      MOVLW   0x03            ; Port E-pin 0 in, pin 1 in, pin 2 out
        MOVWF   TRISE           ; into the PORTE control register
                                ; This also sets port D to not act as a
                                ; Parallel port
20
        CLRF    TRISD           ; All output code for port D
        CLRF    PORTD           ; Close all alternate injectors

24      SETF    TRISB           ; all input code into portB control reg

        MOVLW   0x88            ; All outputs but pin 7 and 3
        MOVWF   TRISC           ; into port C control register
28                              ; pin 7-serial in, pin 3-prog switch

        BCF PORTC, 4            ; Lock off the heater
        BCF PORTC, 5            ; Lock off the lockoff
```

```
32
        ; *** SUMMARY ***
        ; ALL A/D pins enabled
        ; Port A all analogue input but pin 6, digital out
36      ; Port B all digital input
        ; Port C pins 7 & 3 in rest for output
        ; Port D all digital output
        ; Port E out, in, in
40
    timer_prep
        CLRF    T1CON           ; Setting up Timer 1
        BSF     T1CON, T1CKPS1  ; prescale timer 1 by 4
44

        CLRF    T2CON           ; Set up timer 2,
        BSF     T2CON, TOUTPS3  ; set the timer 2 postscaler to 16
48      BSF     T2CON, TOUTPS2
        BSF     T2CON, TOUTPS1
        BSF     T2CON, TOUTPS0
        BSF     T2CON, T2CKPS1  ; set the timer 2 prescaler to 16
52      CLRF    TMR2            ; set the timer to 0

        MOVLW   0x5f            ; initialize the period to something non-0
        MOVWF   PR2             ;
56

    Serial_prep
        MOVLW   0x40            ; This is the baud rate generator value
60      MOVWF   SPBRG           ; corresponding to 9600 baud @ 40 MHz

        MOVLW   0x20            ; Config serial transmit
        MOVWF   TXSTA           ;
64
        CLRF    PIR1            ; Config serial
        MOVLW   0x90            ;
```

```
        MOVWF   RCSTA           ; Config Serial Receive
68
    Inter_prep
        BCF     RCON, IPEN      ; Disable Priority Levels
        BCF     PIR1, TMR1IF    ; Clear Timer 1 overflow flag
72      BCF     PIR1, TMR2IF    ; Clear Timer 2 match flag
        MOVLW   0x48            ; Hex for, disable global interrupts,
                                ; enable peripheral interrupts and unmask
                                ; portb change interrupt
76      MOVWF   INTCON          ; Move this into the interrupt control reg
        MOVLW   0x81            ; Port B set by TRISB, change on b hi-pri
        MOVF    INTCON2         ; into incon2

80      CLRF    PIE1            ;
        BSF     PIE1, TMR1IE    ; Enable the timer 1 interrupt
        BSF     PIE1, TMR2IE    ; Enable the timer 2 interrupt

84  mode_and_interface_prep
        BSF     PORTC, 0        ; all lights on during boot
        BSF     PORTC, 1        ;
        BSF     PORTC, 2        ;
88
        BSF     PORTE, 2        ; enable native injectors

    flash_adr_prep
92      MOVLW   0x40            ; address high bits start at 16384
        MOVWF   FL_HIGH_ADR     ; into flash address high register


96  variable_prep
        CLRF    AIR_IN_TEMP     ; set analogue variables to 0
        CLRF    BARO            ;
        CLRF    OXYGEN          ;
100     CLRF    REVS            ;
        CLRF    COOLANT_TEMP    ;
```

```
        CLRF    ALT_FUEL_LEV    ;
        CLRF    THROTTLE_POS    ;
104



    get_multipliers
108     MOVLW   0x3f            ; Address for low 02 threshold
        MOVWF   FL_READ_ADDR_H  ;
        MOVLW   0xF0            ;
        MOVWF   FL_READ_ADDR_L  ;
112
        CALL    read_flash      ; lookup value
        NOP                     ;
        MOVF    FL_READ_VAL, W  ; Put the value in the low 02
116     MOVWF   IR_02_THRESH_L  ; threshold variable

        INCF    FL_READ_ADDR_L  ; This pattern continues
        CALL    read_flash      ; for all threshold and
120     NOP                     ; multiplier values
        MOVF    FL_READ_VAL, W  ;
        MOVWF   IR_02_THRESH_LM ;

124     INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
        NOP                     ;
        MOVF    FL_READ_VAL, W  ;
128     MOVWF   IR_02_THRESH_HM ;

        INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
132     NOP                     ;
        MOVF    FL_READ_VAL, W  ;
        MOVWF   IR_02_THRESH_H  ;

136     INCF    FL_READ_ADDR_L  ;
```

```
            CALL    read_flash       ;
            NOP                      ;
            MOVF    FL_READ_VAL, W   ;
140         MOVWF   IR_TP_THRESH_L   ;

            INCF    FL_READ_ADDR_L   ;
            CALL    read_flash       ;
144         NOP                      ;
            MOVF    FL_READ_VAL, W   ;
            MOVWF   IR_TP_THRESH_LM  ;

148         INCF    FL_READ_ADDR_L   ;
            CALL    read_flash       ;
            NOP                      ;
            MOVF    FL_READ_VAL, W   ;
152         MOVWF   IR_TP_THRESH_HM  ;

            INCF    FL_READ_ADDR_L   ;
            CALL    read_flash       ;
156         NOP                      ;
            MOVF    FL_READ_VAL, W   ;
            MOVWF   IR_TP_THRESH_H   ;

160         INCF    FL_READ_ADDR_L   ;
            CALL    read_flash       ;
            NOP                      ;
            MOVF    FL_READ_VAL, W   ;
164         MOVWF   IR_02_MULT_LL    ;

            INCF    FL_READ_ADDR_L   ;
            CALL    read_flash       ;
168         NOP                      ;
            MOVF    FL_READ_VAL, W   ;
            MOVWF   IR_02_MULT_L     ;
```

```
172     INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
        NOP                     ;
        MOVF    FL_READ_VAL, W  ;
176     MOVWF   IR_02_MULT_H    ;

        INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
180     NOP                     ;
        MOVF    FL_READ_VAL, W  ;
        MOVWF   IR_02_MULT_HH   ;

184     INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
        NOP                     ;
        MOVF    FL_READ_VAL, W  ;
188     MOVWF   IR_TP_MULT_LL   ;

        INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
192     NOP                     ;
        MOVF    FL_READ_VAL, W  ;
        MOVWF   IR_TP_MULT_L    ;

196     INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
        NOP                     ;
        MOVF    FL_READ_VAL, W  ;
200     MOVWF   IR_TP_MULT_H    ;

        INCF    FL_READ_ADDR_L  ;
        CALL    read_flash      ;
204     NOP                     ;
        MOVF    FL_READ_VAL, W  ;
        MOVWF   IR_TP_MULT_HH   ;
```

```
208
        CLRF    DELAY           ;
        CLRF    in_boot_delay   ;

212 boot_delay_out

        DECFSZ  DELAY,F         ; nested loop to delay during boot
        GOTO    boot_delay_in   ;
216     NOP                     ; Keeps the LEDs lit letting us
        GOTO    boot_delay_done ; know that a reset took place
        NOP

220 boot_delay_in
        DECFSZ  in_boot_delay,F ;
        GOTO    boot_delay_in   ;
        NOP                     ;
224     GOTO    boot_delay_out  ;
        NOP                     ;

    boot_delay_done
228
        BCF     PORTC, 0        ; all lights off after the boot delay
        BCF     PORTC, 1        ;
        BCF     PORTC, 2        ;
232
    inter_start
        MOVF    PORTB, 1        ; Clear mismatch condition
        CLRF    WREG            ; make W all zero's to mask for portb
236     IORWF   PORTB, 0        ; inclusive or with port b
        BTFSS   STATUS, Z       ; if everything is 0 then we are ok
        GOTO    inter_start     ; if not lets go around again
        NOP                     ;
240     BSF     INTCON,GIE          ; Turn on global interrupts
```

```
    check_secret_switch
244
                                ; put in native mode first to be safe
        CLRF    MODE            ; Mode to Native
        CLRF    TROUBLE         ; Trouble is off
248     BSF     PORTE, 2        ; Send pulses to the native injectors


        BTFSS   PORTC, 3        ; check programming switch
252     GOTO    check_ser_port  ; code that is used to get new fuel map
        NOP                     ;

    ;**************** END SETUP AND INITIALIZATION ********************;
```

7.6 Interrupt Service Routine (ISR)

7.6.1 Assumptions

- Execution of the Initialization routine is always completed before the execution of the ISR.
- Values of OXYGEN, BARO, AIR_IN_TEMP, COOLANT_TEMP, ALT_FUEL_LEV, and THROTTLE_POS are updated by another software unit at frequencies that satisfy the requirements document.
- Jumps to instructions inside the ISR that do not originate from within the ISR will not occur.

7.6.2 Variables

This section lists, in separate tables, the global variables that are read, the global variables that are changed, and local variables that are used during the execution of the interrupt service routine. The 3 character prefix for the ISR is IR_.

Variables Read - ISR

Variable Name	Type & Description
PORTB	SF - Port B
TMR1H	SF - High Bits of Timer 1
TMR1L	SF - Low Bits of Timer 2
PRODH	SF - High Bits of Multiplication Product
PRODL	SF - Low Bits of Multiplication Product
MODE	GP - Stores Control Mode of System
AIR_IN_TEMP	GP - Stores Converted Value
BARO	GP - Stores Converted Value
OXYGEN	GP - Stores Converted Value
THROTTLE_POS	GP - Stores Converted Value
FL_READ_VAL	GP - Value Most Recently Read From Flash
IR_O2_THRESH_L	GP - Low O2 Threshold Value
IR_O2_THRESH_LM	GP - Low Mid O2 Threshold Value
IR_O2_THRESH_HM	GP - High Mid O2 Threshold Value
IR_O2_THRESH_H	GP - High O2 Threshold Value
IR_TP_THRESH_L	GP - Low TPS Threshold Value
IR_TP_THRESH_LM	GP - Low Mid TPS Threshold Value
IR_TP_THRESH_HM	GP - High Mid TPS Threshold Value
IR_TP_THRESH_H	GP - High TPS Threshold Value
IR_O2_MULT_LL	GP - Low Low Range O2 Multiplier
IR_O2_MULT_L	GP - Low Range O2 Multiplier
IR_O2_MULT_H	GP - High Range O2 Multiplier
IR_O2_MULT_HH	GP - High High Range O2 Multiplier
IR_TP_MULT_LL	GP - Low Low Range TPS Multiplier
IR_TP_MULT_L	GP - Low Range TPS Multiplier
IR_TP_MULT_H	GP - High Range TPS Multiplier
IR_TP_MULT_HH	GP - High High Range TPS Multiplier

Variables Changed - ISR

Variable Name	Type & Description
WREG	SF - Operations Register
STATUS	SF - Operational Status Register
INTCON	SF - Interrupt Control Register
T1CON	SF - Timer 1 Control Register
T2CON	SF - Timer 2 Control Register
TMR2	SF - Timer 2 Value
PR2	SF - Timer 2 Period
PORTC	SF - Port C
PORTD	SF - Port D
PORTE	SF - Port E
PIR1	SF - Peripheral Interrupt Register
IR_TIMHI	GP - High Bits of Timer 1
TROUBLE	GP - Stores Trouble State
IR_TIMLO	GP - Low Bits of Timer 2
FL_READ_ADDR_H	GP - High Bits of Address for Flash Read
FL_READ_ADDR_L	GP - Low Bits of Address for Flash Read

Local Variables - ISR

Variable Name	Type & Description
ir_working_reg	GP - Used for Building Look-up Addresses, and Performing Fixed Point Multiplication
ir_02_mult	GP - Multiplier for Post Lookup 02 Calculation
ir_tps_mult	GP - Multiplier for Post Lookup TPS Calculation
ir_trbl_delay	GP - Counter for Flashing LEDs Indicating Trouble
ir_trbl_delay2	GP - Counter for Flashing LEDs Indicating Trouble

7.6.3 ISR Code

```
256  ;************************ ISR HANDLER *****************************;

         ORG     0x008           ; high-pri interrupt vector location
```

```
260  isr_handler
         BTFSC   INTCON,RBIF     ; Interrupt was change on B?
         CALL    change_onb      ; Call Change on B interrupt handler
         NOP                     ;
264
         BTFSC   PIR1, TMR2IF    ; Interrupt was timer 2 period match?
         CALL    timer_2         ; Call T2 period match interrupt handler
         NOP                     ;
268
         BTFSC   PIR1, TMR1IF    ; Interrupt was timer 1 overflow?
         GOTO    handle_trouble  ; Go to the trouble mode
         NOP                     ;
272
         RETFIE  1               ; Return with interrupts back on
                                 ; fast context switching enabled

276  ;*********************** END ISR HANDLER ***************************;

     ;*********************** ISR SUB ROUTINES **************************;

280  change_onb
         MOVF    PORTB, 1    ; Clear mismatch condition
         BCF     INTCON,RBIF ; clear the port B interrupt flag
         BTFSC   PORTB,7     ; Check if bit 7 went hi,
284      GOTO    new_pulse   ; Code to handle detection of new pulse
         NOP

     pulse_end
288      BCF     T1CON, TMR1ON   ; stop timer 1
         MOVF    TMR1H, W        ; Get high part of pulse Length
         MOVWF   IR_TIMHI        ; Save it to IR_TIMHI
         MOVF    TMR1L, W        ; Get low part of pulse Length
292      MOVWF   IR_TIMLO        ; Save it to IR_TIMLO
```

```
        BTFSS   MODE, 1           ; Check mode before computing elongation
        RETURN                    ; Mode[1] clear -> not alt so return
296     NOP

    find_o2_range
        MOVF    OXYGEN, W             ; o2 value into W
300     SUBWF   IR_02_THRESH_L, W     ; subtract it from the low threshold
        BTFSC   STATUS, C             ; carry bit is set -> correct range
        GOTO    o2_lowlow             ; go to the o2 lowlow code
        NOP
304     MOVF    OXYGEN, W             ; Same as above, however with
        SUBWF   IR_02_THRESH_LM, W    ; a different thresh value
        BTFSC   STATUS, C
        GOTO    o2_low
308     NOP
        MOVF    OXYGEN, W             ; Same as above, however with
        SUBWF   IR_02_THRESH_HM, W    ; a different thresh value
        BTFSC   STATUS, C
312     GOTO    o2_med
        NOP
        MOVF    OXYGEN, W             ; Same as above, however with
        SUBWF   IR_02_THRESH_H, W     ; a different thresh value
316     BTFSC   STATUS, C
        GOTO    o2_high
        NOP
        GOTO    o2_highhigh           ; only remaining possibility
320
    load_o2_mult
    o2_lowlow
        MOVF    IR_02_MULT_LL, W      ; move in the low low multiplier
324     MOVWF   ir_02_mult            ;
        GOTO    find_tps_range  ;
        NOP                           ;
    o2_low
328     MOVF    IR_02_MULT_L, W       ; move in the low multiplier
```

```
        MOVWF   ir_02_mult          ;
        GOTO    find_tps_range  ;
        NOP                         ;
332  o2_med
        MOVLW   0x80                ; '1' - fixed point
        MOVWF   ir_02_mult          ; move it in
        GOTO    find_tps_range  ;
336     NOP                         ;
     o2_high
        MOVF    IR_02_MULT_H, W     ; move in the high multiplier
        MOVWF   ir_02_mult          ;
340     GOTO    find_tps_range  ;
        NOP                         ;
     o2_highhigh
        MOVF    IR_02_MULT_HH, W    ; move in the high high multiplier
344     MOVWF   ir_02_mult          ;

     find_tps_range
        MOVF    THROTTLE_POS, W     ; tps value into W
348     SUBWF   IR_TP_THRESH_L, W   ; subtract it from the low threshold
        BTFSC   STATUS, C           ; carry bit is set -> correct range
        GOTO    tps_lowlow          ; go to the lowlow code
        NOP
352     MOVF    THROTTLE_POS, W     ; Same as above, however with
        SUBWF   IR_TP_THRESH_LM, W  ; a different thresh value
        BTFSC   STATUS, C
        GOTO    tps_low
356     NOP
        MOVF    THROTTLE_POS, W     ; Same as above, however with
        SUBWF   IR_TP_THRESH_HM, W  ; a different thresh value
        BTFSC   STATUS, C
360     GOTO    tps_med
        NOP
        MOVF    THROTTLE_POS, W     ; Same as above, however with
        SUBWF   IR_TP_THRESH_H, W   ; a different thresh value
```

```
364       BTFSC   STATUS, C
          GOTO    tps_high
          NOP
          GOTO    tps_highhigh
368
      load_tps_mult

      tps_lowlow
372       MOVLW   0x80                ; '1' - fixed point
          MOVWF   ir_tps_mult         ; Move it in
          GOTO    tps_mult_done       ;
          NOP                         ;
376   tps_low
          MOVF    IR_TP_MULT_LL, W    ; move in the low low multiplier
          MOVWF   ir_tps_mult         ;
          GOTO    tps_mult_done       ;
380       NOP                         ;
      tps_med
          MOVF    IR_TP_MULT_L, W     ; move in the low multiplier
          MOVWF   ir_tps_mult         ;
384       GOTO    tps_mult_done       ;
          NOP                         ;
      tps_high
          MOVF    IR_TP_MULT_H, W     ; move in the high multiplier
388       MOVWF   ir_tps_mult         ;
          GOTO    tps_mult_done       ;
          NOP                         ;
      tps_highhigh
392       MOVF    IR_TP_MULT_HH, W    ; move in the high high multiplier
          MOVWF   ir_tps_mult         ;


396   tps_mult_done                   ; now both multipliers are loaded

          MOVLW   0xFC                ; mask to keep the 6 MSBs
```

```
        ANDWF   AIR_IN_TEMP, 0      ; 6 MSBs of air in temp in w
400     MOVWF   ir_working_reg      ; and now in the temp register
        RRNCF   ir_working_reg, 1   ; shift until they take up the 6 lsb's
        RRNCF   ir_working_reg, 1   ;
        BSF     ir_working_reg, 6   ; now set the 2nd MSB to address fmap
404     MOVF    ir_working_reg, W   ; put this computed value in W
        MOVWF   FL_READ_ADDR_H      ; and into high reg for address lookup

                                    ; we still need 2nd LSB
408                                 ; this is more easily done later

        MOVLW   0xFE                ; Mask for 7 MSBs
        ANDWF   BARO, 0             ; 7 MSBs of BARO now in W
412     MOVWF   ir_working_reg,     ; now in the temp reg
        RRNCF   ir_working_reg, 1   ; Shift until they take 7 LSBs

        BTFSC   AIR_IN_TEMP, 1      ; if the 2nd LSB is set, set bit 7
416     BSF     ir_working_reg, 7   ;

        MOVF    ir_working_reg, W   ; into wreg
        MOVWF   FL_READ_ADDR_L      ; and into low reg for address lookup
420
        CALL    read_flash          ; LOOKUP
        NOP                         ;

424   post_multiply
        MOVF    FL_READ_VAL, W      ; get the lookup result
        MULWF   IR_TIMHI            ; high bits timer1 * factor

428     MOVF    PRODH, W            ; result in ProdH and Prodl
        MOVWF   ir_working_reg      ; high bits into temp
        RLNCF   ir_working_reg, 1   ; shift once left, to account for
                                    ; fixed point, and timer1,2 configs
432
        BTFSC   PRODL, 7            ; we need MSB of low bits
```

```
        BSF     ir_working_reg, 0   ; the rest are fractional

436     MOVF    ir_working_reg, W   ; formatted product into w
        MULWF   ir_02_mult          ; multiply by o2 factor

        MOVF    PRODH, W            ; result in ProdH and Prodl
440     MOVWF   ir_working_reg      ; get high bits of product
        RLNCF   ir_working_reg, 1   ; shift it once to the left.

        BTFSC   PRODL, 7            ; we need MSB of low bits
444     BSF     ir_working_reg, 0   ; rest are fractional

        MOVF    ir_working_reg, W   ; formatted product into w
        MULWF   ir_tps_mult         ; multiply by tps factor
448
        MOVF    PRODH, W            ; result in ProdH and Prodl
        MOVWF   ir_working_reg      ; get high bits of product
        RLNCF   ir_working_reg, 1   ; shift it once to the left
452
        BTFSC   PRODL, 7            ; we need MSB of low bits
        BSF     ir_working_reg, 0   ; the rest are fractional
        MOVF    ir_working_reg, W   ; formatted product into w
456

        CLRF    TMR2            ; Clear timer 2
        MOVWF   PR2             ; product is timer 2 period
460     BSF     T2CON,TMR2ON    ; Turn on timer 2
        RETURN
        NOP

464   new_pulse

        BTFSS   MODE, 1         ; Check mode before sending alt pulse
        GOTO    time_pulse      ; MODE[1] clear -> not alt
468     NOP                     ;
```

```
        MOVF    PORTB, W        ; Native bank to W
        XORLW   0x80            ; Xor with b10000000 to lose pin 7
472     BTFSC   STATUS, Z       ; Check if the xor killed all bits
        BSF     PORTD,7         ; if so Set 7 pin high,
        IORWF   PORTD, 1        ; Copy this to port D

476 time_pulse
        BTFSC   T2CON, TMR2ON   ; check for overlapped pulse/elongation
        GOTO    handle_trouble  ; if detected go to trouble
        NOP
480     CLRF    WREG            ;
        MOVWF   TMR1H           ; Clear the timer one high bits
        MOVWF   TMR1L           ; Clear the timer two low bits
        BSF     T1CON, TMR1ON       ; start timer 1
484     RETURN
        NOP


488 ;********************* TIMER 2 PERIOD MATCH  ***********************;
    timer_2
        CLRF    PORTD           ; end alt pulse
        BCF     PIR1, TMR2IF    ; clear timer 2 flag
492     BCF     T2CON,TMR2ON        ; Turn off timer 2
        CLRF    TMR2            ; Clear timer 2
        RETURN
        NOP
496
    ;***************************** TROUBLE ******************************;

    handle_trouble
500     CLRF    PORTD           ; end any alternate pulses
        BCF     PORTC, 4        ; Lock off the heater
        BCF     PORTC, 5        ; Lock off the lockoff
        CLRF    MODE            ; mode to native
```

```
504     BSF     PORTE, 2        ; Send pulses to the native injectors
        BSF     TROUBLE, 0      ; Trouble on
        BSF     PORTC, 0        ; Red light on
        BCF     PIR1, TMR1IF    ; clear for reset
508     CLRF    ir_trbl_delay   ; initialize loop counters
        CLRF    ir_trbl_delay2  ;

    toggle_leds
512     MOVLW   0x18            ; set outside loop counter
        MOVWF   DELAY           ;
        BTFSC   PORTC, 1        ; Check if the leds are on
        GOTO    leds_off        ; if so turn them off
516     NOP                     ; if not turn them on
        BSF     PORTC, 1        ; yellow on
        BSF     PORTC, 2        ; Green on
        GOTO    trbl_2out_loop  ; loop to take up time
520     NOP

    leds_off
        BCF     PORTC, 1        ; yellow off
524     BCF     PORTC, 2        ; Green off

    trbl_2out_loop
        DECFSZ  DELAY,F         ; Outer Outer loop
528     GOTO    trbl_out_loop   ;
        NOP                     ;
        GOTO    toggle_leds     ;
        NOP                     ;
532
    trbl_out_loop
        DECFSZ  ir_trbl_delay   ; Outer Loop
        GOTO    trbl_in_loop    ;
536     NOP                     ;
        GOTO    trbl_2out_loop  ;
        NOP                     ;
```

```
540  trbl_in_loop
         DECFSZ  ir_trbl_delay2  ; Inner Loop
         GOTO    trbl_in_loop    ;
         NOP                     ;
544      GOTO    trbl_out_loop   ;
         NOP

     ;********************** END ISR SUBROUTINES *************************;
```

7.7 Main Loop

7.7.1 Assumptions

- Execution of the Initialization routine is always completed before the execution of the Main Loop.

- The ISR will have control of the processor for less than 80% of time as to allow the Main Loop to meet its time bounds.

- The alternate fuel tank has a thermostat that is used in conjunction with o_heaterRelay for controlling the alternate fuel reserve heater.

7.7.2 Variables

This section lists, in separate tables, the global variables that are read, the global variables that are changed, and local variables that are used during the execution of the Main Loop. The 3 character prefix for the Main Loop is ML_.

Variables Read - Main Loop

Variable Name	Type & Description
PORTA	SF - Port A
PORTB	SF - Port B
PR2	SF - Timer 2 Period
AD_HI	GP - Register for Storing A/D High Bits
IR_TIMHI	GP - Register for Storing High Bits of Timer 1
IR_TIMLO	GP - Register for Storing Low Bits of Timer 1
FL_READ_VAL	GP - Register for Storing Results of Flash reads

Variables Changed - Main Loop

Variable Name	Type & Description
WREG	SF - Operations Register
STATUS	SF - Operational Status Register
PORTC	SF - Port C
PORTE	SF - Port E
MODE	GP - Stores Control Mode of System
AIR_IN_TEMP	GP - Stores Converted Value
BARO	GP - Stores Converted Value
OXYGEN	GP - Stores Converted Value
REVS	GP -Stores Converted Value
COOLANT_TEMP	GP - Stores Converted Value
ALT_FUEL_LEV	GP - Stores Converted Value
THROTTLE_POS	GP - Stores Converted Value
AD_CHAN	GP - Channel for A/D Conversion
SP_VAL_OUT	GP - Serial Value to Transmit
SP_CODE_OUT	GP - Describes Nature of Serial Transmission

7.7.3 Main Loop Code

```
548   ;************************* MAIN LOOP ********************************;
```

```
    main_loop

552 mode_update
        BTFSC   PORTA, 4        ; check the fuel selector switch
        GOTO    switch_on       ;
        NOP
556     BCF     PORTC, 5        ; lock off the lock off
        BCF     PORTC, 4        ; turn off heater relay
        CLRF    MODE            ; Mode to native

560     BTFSC   PORTE, 2        ; if already burning native, do nothing
        GOTO    native_leds     ;
        NOP                     ;

564 enable_nat
        CLRF    WREG            ; make W all zero's to mask for portb
        IORWF   PORTB, 0        ; inclusive or with port b
        BTFSS   STATUS, Z       ; if everything is 0 then we are ok
568     GOTO    enable_nat      ; if not lets go around again
        NOP                     ;
        BSF     PORTE, 2        ; Send pulses to the native injectors

572 native_leds
        BSF     PORTC, 0        ; turn on the red light
        BCF     PORTC, 1        ; turn off the yellow
        BCF     PORTC, 2        ; turn off the green
576
        GOTO    ad_block        ; mode update done,
        NOP                     ;

580 switch_on
        BTFSC   MODE, 1         ; mode is stby or alt fuel?
        GOTO    alt_mode        ; Goto alt, or run through to stby
        NOP
584     BSF     MODE, 0         ; Mode to stbynat
```

```
        BTFSC   PORTE, 2         ; if already burning native, skip enable
        GOTO    stby_leds        ;
588     NOP                      ;

    enable_stby
        CLRF    WREG             ; make W all zero's to mask for portb
592     IORWF   PORTB, 0         ; inclusive or with port b
        BTFSS   STATUS, Z        ; if everything is 0 then we are ok
        GOTO    enable_stby      ; if not lets go around again
        NOP                      ;
596     BSF     PORTE, 2         ; enable native injectors

    stby_leds
        BSF     PORTC, 5         ; open the lock off
600     BSF     PORTC, 4         ; turn on heater relay
        BSF     PORTC, 1         ; turn on the yellow light
        BCF     PORTC, 2         ; turn off the green one one
        BCF     PORTC, 0         ; turn off the red one
604
        MOVLW   0xC4             ; engine revving sufficiently hight?
        ADDWF   IR_TIMHI, 0      ; high rpm implies short pulse length
        BTFSC   STATUS, C        ; carry bit was set -> timer 1 too high
608     GOTO    ad_block         ; if not, continue
        NOP

        MOVLW   0xCE             ; check for sufficient coolant temp
612     ADDWF   COOLANT_TEMP, 0  ; if the add creates a carry we are ok
        BTFSS   STATUS, C        ; if not, got to ad section
        GOTO    ad_block         ;
        NOP
616
        MOVLW   0xCE             ; check for sufficient fuel level
        ADDWF   ALT_FUEL_LEV, 0  ; if the add creates a carry we are ok
        BTFSS   STATUS, C        ; if not, goto ad section
```

```
620     GOTO    ad_block          ;
        NOP

    disable_nat
624     CLRF    WREG              ; make W all zero's to mask for portb
        IORWF   PORTB, 0          ; inclusive or with port b
        BTFSS   STATUS, Z         ; if everything is 0 then we are ok
        GOTO    disable_nat       ; if not lets go around again
628     NOP                       ;
        BCF     PORTE, 2          ; Disable native injectors

        BSF     MODE, 1           ;
632     BCF     MODE, 0           ; Mode is now altfuel

    alt_leds
        BSF     PORTC, 2          ; turn on the green LED
636     BCF     PORTC, 1          ; turn off yellow
        BCF     PORTC, 0          ; turn off red

        GOTO    ad_block          ;
640     NOP                       ;

    alt_mode
        MOVLW   0xD8              ; check for sufficient coolant temp
644     ADDWF   COOLANT_TEMP, 0 ; if the add creates a carry we are ok
        BTFSS   STATUS, C         ; if not, back to stby
        GOTO    leave_alt         ;
        NOP
648
        MOVLW   0xD8              ; check for sufficient fuel level
        ADDWF   ALT_FUEL_LEV, 0 ; if the add creates a carry we are ok
        BTFSS   STATUS, C         ; if not, back to stby
652     GOTO    leave_alt         ;
        NOP
```

```
        GOTO    ad_block        ;
656     NOP                     ;

    leave_alt
        BCF     MODE, 1         ;
660     BSF     MODE, 0         ; Mode is now standby


    enable_stby2
664     CLRF    WREG            ; make W all zero's to mask for portb
        IORWF   PORTB, 0        ; inclusive or with port b
        BTFSS   STATUS, Z       ; if everything is 0 then we are ok
        GOTO    enable_stby2    ; if not lets go around again
668     NOP                     ;
        BSF     PORTE, 2        ; Enable native injectors

        BCF     PORTC, 2        ; turn off the green LED
672     BSF     PORTC, 1        ; turn on yellow
        BCF     PORTC, 0        ; turn off red

    ad_block
676
    get_intake_temp
        CLRF    AD_CHAN         ; Select channel 0
        CALL    adcget          ;
680     NOP                     ;

        MOVF    AD_HI, W        ; move the converted value into w
        MOVWF   AIR_IN_TEMP     ; store it in the correct place
684     MOVWF   SP_VAL_OUT      ; prepare for serial transmission
        MOVLW   0x81            ; move 129 into w, intake temp code
        MOVWF   SP_CODE_OUT     ;
        CALL    serial_tx       ;
688     NOP                     ;
```

```
        get_baro_pres
            INCF    AD_CHAN, 1      ; select channel 1
692         CALL    adcget          ;
            NOP                     ;

            MOVF    AD_HI, W        ; move the converted value into w
696         MOVWF   BARO            ; store it in the correct place
            MOVWF   SP_VAL_OUT      ; prepare for serial transmission
            MOVLW   0x82            ; move 130 into w, Baro code
            MOVWF   SP_CODE_OUT     ;
700         CALL    serial_tx       ;
            NOP

        get_oxygen
704         INCF    AD_CHAN, 1      ; select channel 2
            CALL    adcget          ;
            NOP                     ;

708         MOVF    AD_HI, W        ; move the converted value into w
            MOVWF   OXYGEN          ; store it in the correct place
            MOVWF   SP_VAL_OUT      ; prepare for serial transmission
            MOVLW   0x83            ; move 131 into w, O2 code
712         MOVWF   SP_CODE_OUT     ;
            CALL    serial_tx       ;
            NOP                     ;

716     get_rpm
            INCF    AD_CHAN, 1      ; select channel 3
            CALL    adcget          ;
            NOP                     ;
720
            MOVF    AD_HI, W        ; move the converted value into w
            MOVWF   REVS            ; store it in the correct place
            MOVWF   SP_VAL_OUT      ; prepare for serial transmission
724         MOVLW   0x84            ; move 132 into w, RPM code
```

```
        MOVWF   SP_CODE_OUT     ;
        CALL    serial_tx       ;
        NOP
728
    get_coolant_temp
        INCF    AD_CHAN, 1      ; select channel 4
        CALL    adcget          ;
732     NOP                     ;

        MOVF    AD_HI, W        ; move the converted value into w
        MOVWF   COOLANT_TEMP    ; store it in the correct place
736     MOVWF   SP_VAL_OUT      ; prepare for serial transmission
        MOVLW   0x85            ; move 133 into w, coolant temp code
        MOVWF   SP_CODE_OUT     ;
        CALL    serial_tx       ;
740     NOP

    get_fuel_lev
        INCF    AD_CHAN, 1      ; select channel 5
744     CALL    adcget          ;
        NOP                     ;

        MOVF    AD_HI, W        ; move the converted value into w
748     MOVWF   ALT_FUEL_LEV    ; store it in the correct place
        MOVWF   SP_VAL_OUT      ; prepare for serial transmission
        MOVLW   0x86            ; move 134 into w, fuel level code
        MOVWF   SP_CODE_OUT     ;
752     CALL    serial_tx       ;
        NOP

    get_throttle_pos
756     INCF    AD_CHAN, 1      ; select channel 6
        CALL    adcget          ;
        NOP                     ;
```

```
760        MOVF     AD_HI, W          ; move the converted value into w
           MOVWF    THROTTLE_POS      ; store it in the correct place
           MOVWF    SP_VAL_OUT        ; prepare for serial transmission
           MOVLW    0x87              ; move 135 into w, TPS code
764        MOVWF    SP_CODE_OUT       ;
           CALL     serial_tx         ;
           NOP                        ;

768    send_plength
           MOVF     IR_TIMHI, W       ; move the most sig 8 bits of the
           MOVWF    SP_VAL_OUT        ; 16 bit timer into w
           MOVLW    0x88              ;
772        MOVWF    SP_CODE_OUT       ;
           CALL     serial_tx         ;
           NOP                        ;

776        MOVF     IR_TIMLO, W       ; move the least sig 8 bits of the
           MOVWF    SP_VAL_OUT        ; 16 bit timer into w
           MOVLW    0x89              ;
           MOVWF    SP_CODE_OUT       ;
780        CALL     serial_tx         ;
           NOP                        ;

       send_elongation
784        MOVF     PR2, W            ;
           MOVWF    SP_VAL_OUT        ;
           MOVLW    0x8A              ;
           MOVWF    SP_CODE_OUT       ;
788        CALL     serial_tx         ;
           NOP                        ;

           GOTO     main_loop         ;
792        NOP                        ;

       ;*********************** END MAIN LOOP ******************************;
```

7.8 Programming Loop

7.8.1 Assumptions

- Execution of the Initialization routine is always completed before the execution of the Programming Loop.
- The serial port will only be connected to properly configured machines running the Car Communicator PC application.
- Validity of fuel maps is determined by the Car Communicator PC application.
- Communication is synchronized by resetting the controller before initiating fuel map transfer.

7.8.2 Variables

This section lists, in separate tables, the global variables that are read, and the global variables that are changed during the execution of the Programming Loop. The 3 character prefix for the programmingLoop is PL_.

Variables Read - Programming Loop

Variable Name	Type & Description
PIR1	SF - Peripheral Interrupt Register
SP_VAL_IN	GP - Incoming Serial Value
SP_SUCC_IN	GP - Incoming Serial Code

Variables Changed - Programming Loop

Variable Name	Type & Description
WREG	SF - Operations Register
INTCON	SF - Interrupt Control Register
PORTE	SF - Port E
TBLPTRU	SF - Table Pointer Upper Bits
TBLPTRH	SF - Table Pointer High Bits
TBLPTRL	SF - Table Pointer Low Bits
MODE	GP - Stores Control Mode of System
FL_WRITE_VAL	GP - Value to be Written to Flash
FL_LAST_RES	GP - Result of Last Communication with PC

7.8.3 Programming Loop Code

```
    ;*********************** PROGRAM FUEL MAP ***************************;
796 prog_f_map
        ; the controller is set to burn native fuel and
        ; only transmit serial info relating to the serial
        ; programing
800     CLRF    MODE        ;
        BSF     MODE, 3     ; Program Mode On
        BCF     INTCON,GIE  ; Turn off global interrupt switch
        BSF     PORTE, 2    ; Ensure car is burning native
804
        CLRF    TBLPTRU     ; clear table pointer upper bits
        MOVLW   0x3f        ; set table pointer high bits
        MOVWF   TBLPTRH     ;
808     MOVLW   0xF0        ; set table pointer low bits
        MOVWF   TBLPTRL     ;

        CALL    erase_flash ;
812     NOP


    fuel_map_loop
```

```
816       BTFSS   PIR1, RCIF      ; check for new data
          GOTO    fuel_map_loop   ; loop until there is data waiting
          NOP                     ;

820   data_waiting
          CALL    serial_rc       ; get the data from the serial Port
          NOP                     ;

824       MOVF    SP_SUCC_IN, W   ; data into the correct for write
          MOVWF   FL_LAST_RES     ;
          MOVF    SP_VAL_IN, W    ; data into the correct for write
          MOVWF   FL_WRITE_VAL    ;
828       GOTO    write_flash     ; write the flash
          NOP                     ;

          GOTO    fuel_map_loop   ;
832
      ;******************* END PROGRAM FUEL MAP *************************;
```

7.9 Serial Port

7.9.1 Assumptions

- Execution of the Initialization routine is always completed before the execution of any routine in the Serial Port software unit.
- Before a call to `serial_tx` the value to be transmitted is stored in the SP_VAL_OUT register, and the code to be transmitted is stored in the SP_CODE_OUT register.
- After a call to `serial_rc` the 2 bytes most recently received are stored in the SP_VAL_IN and SP_SUCC_IN registers.
- All serial port parameters have been configured by the Initialization routine to be consistent with those of the Car Communicator PC application.

7.9.2 Variables

This section lists, in separate tables, the global variables that are read, and the global variables that are changed during the execution of routines in the Serial Port software unit. The 3 character prefix for the Serial Port unit is SP_.

Variables Read - Serial Port

Variable Name	Type & Description
PIR1	SF - Peripheral Interrupt Register
RCREG	SF - Register for Outbound Serial Data
SP_CODE_OUT	GP - Describes Nature of Serial Transmission
SP_VAL_OUT	GP - Serial Value to Transmit

Variables Changed - Serial Port

Variable Name	Type & Description
WREG	SF - Operations Register
PORTC	SF - Port C
TXREG	SF - Register for Inbound Serial Data
SP_SUCC_IN	GP - Incoming Serial Code
SP_VAL_IN	GP - Incoming Serial Value

7.9.3 Serial Port Code

```
836  ;************************ SERIAL RECEIVE CODE *********************;
     serial_rc

     check_for_ser_data
840      BTFSC   PIR1, RCIF      ; check for new data
         GOTO    get_ser_data    ;
         RETURN                  ;
```

```
844  get_ser_data
         MOVF    RCREG, W         ; get the data and put it into
         MOVWF   SP_VAL_IN        ; the serial val register

848  second_half
         BTFSS   PIR1, RCIF       ; check for new data
         GOTO    no_sync          ;
         NOP
852
         BCF     PORTC, 1         ;
         MOVF    RCREG, W         ; data into the register used to verify
         MOVWF   SP_SUCC_IN       ;
856      RETURN                   ;
         NOP                      ;

     no_sync
860      BSF     PORTC, 1         ; flashes the yellow led between bytes
         GOTO    second_half      ;
         NOP

864  ;*********************** END SERIAL RECEIVE CODE ********************;

     ;************************ SERIAL TRANSMIT CODE *********************;

868  serial_tx
     send_code
         BTFSS   PIR1,TXIF        ; wait for transmit buffer to empty
         GOTO    send_code        ;
872      NOP                      ;
         MOVF    SP_CODE_OUT, W   ;
         MOVWF   TXREG            ; send to serial Pin
         NOP                      ;
876      NOP                      ;
```

```
    send_val
        BTFSS   PIR1,TXIF       ;
880     GOTO    send_val        ;
        NOP                     ;
        MOVF    SP_VAL_OUT, w   ;
        MOVWF   TXREG           ;
884     NOP                     ;
        NOP                     ;
        RETURN                  ;
        NOP                     ;
888
    ;********************** END SERIAL TRANSMIT CODE ********************;
```

7.10 A/D Conversion

7.10.1 Assumptions

- Execution of the Initialization routine is always completed before the execution of the adcget routine.
- Before a call to adcget the number of the channel which is to have its value converted is stored in the AD_CHAN register.
- After a call to adcget the high bits of the conversion result are stored in AD_HI and the low bits are stored in AD_LO.

7.10.2 Variables

This section lists, in separate tables, the global variables that are read, the global variables that are changed and the local variables that are used during the execution of the A/D Conversion routine. The 3 character prefix for the A/D sub-routine is AD_.

Variables Read - A/D Conversion

Variable Name	Type & Description
ADRESH	SF - Register Where High Bits of Converted A/D Values are Stored
ADRESL	SF - Register storing Low Bits of Converted A/D Value
AD_CHAN	GP - Channel for A/D Conversion

Variables Changed - A/D Conversion

Variable Name	Type & Description
WREG	SF - Operations Register
STATUS	SF - Operational Status Register
ADCON0	SF - A/D Control Register 0
PIR1	SF - Peripheral Interrupt Register
AD_HI	GP - Register for Storing A/D High Bits
AD_LO	GP - Register for Storing A/D Low Bits
DELAY	GP - Counter Used to Wait for A/D Acquisition

Local Variables - A/D Conversion

Variable Name	Type & Description
ad_store	GP - Used to Prepare Byte for ADCON0

7.10.3 A/D Conversion Code

```
    ;************************* A/D COMVERSION CODE **********************;
892
    adcget
        BCF      STATUS, C        ; clear carry flag
        MOVF     AD_CHAN, W       ; get channel number
896     MOVWF    ad_store         ; temporarily store
        RLNCF    ad_store, F      ; shift left thrice
        RLNCF    ad_store, F      ; so it is in the correct bits
        RLNCF    ad_store, F      ; suited to ADCON0
```

```
900     MOVLW   0x81            ; set clock FOSC/32 with A/D on
        IORWF   ad_store, 0     ; or this with the modified chan number
        MOVWF   ADCON0          ; move new val into into adcon0
        CLRF    DELAY           ; set delay counter to 0
904
    acq_wait
        DECFSZ  DELAY,F         ; wait for A/D capacitor to charge
        GOTO    acq_wait        ; to channel voltage
908     NOP                     ;

    sample
        BCF     PIR1, ADIF      ;
912     BSF     ADCON0, GO          ; start A/D conversion

    conv_wait
        DECFSZ  DELAY,F         ; wait for A/D conversion
916     GOTO    conv_wait       ;
                                ;
    val_wait
        BTFSC   ADCON0, GO      ; Wait for value to be in registers
920     GOTO    val_wait        ;


    getval
924     MOVF    ADRESH, W       ; get high word
        MOVWF   AD_HI           ; put it in ad_hi
        MOVF    ADRESL, W       ; get low word
        MOVWF   AD_LO           ; put it in ad_lo
928     RETURN                  ;
        NOP

    ;*********************** END A/D CONVERSION CODE *******************;
```

7.11 Flash

7.11.1 Assumptions

- Execution of the Initialization routine is always completed before the execution of any routine in the Flash software unit.
- Before a call to `write_flash` the value to be written is stored in the FL_WRITE_VAL register, and the result of the last transmission is stored in the FL_LAST_RES register.
- Before a call to `read_flash` the high bits of the address of the byte to be read are stored in the FL_READ_ADDR_H register, the low bits of the address are stored in the FL_READ_ADDR_L register.
- The car communicator PC application strictly adheres to the communication protocol defined in the high level design document.
- Mandatory flash erases and long writes are handled automatically by the `write_flash` sub-routine.

7.11.2 Variables

This section lists, in separate tables, the global variables that are read and changed, and the local variables that are used during the execution of sub-routines in the Flash software unit. The 3 character prefix for the Flash software unit is FL_.

Variables Read - Flash

Variable Name	Type & Description
FL_READ_ADDR_H	GP - High Bits of Address for Flash Read
FL_READ_ADDR_L	GP - Low Bits of Address for Flash Read
FL_WRITE_VAL	GP - Value to be Written to Flash
FL_LAST_RES	GP - Result of Last Communication with PC
SP_VAL_IN	GP - Incoming Serial Value

Variables Changed - Flash

Variable Name	Type & Description
WREG	SF - Operations Register
STATUS	SF - Operational Status Register
TBLPTRU	SF - Table Pointer Upper bits
TBLPTRH	SF - Table Point High Bits
TBLPTRL	SF - Table Pointer Low bits
TABLAT	SF - Table value
EECON1	SF - EEPROM Control Register 1
EECON2	SF - EEPROM Control Register 2
INTCON	SF - Interrupt Control Register
FL_READ_VAL	GP - Register Storing Results of Flash Reads
SP_CODE_OUT	GP - Describes Nature of Serial Transmission
SP_VAL_OUT	GP - Serial value to transmit

Local Variables - Flash

Variable Name	Type & Description
fl_count_bits	GP - Used to Count the Number of High Bits in fl_last_res

7.11.3 Flash Code

```
932  ;************************** READ FLASH ******************************;

     read_flash
         CLRF    TBLPTRU
936      MOVF    FL_READ_ADDR_H, W    ; high flash address into,
         MOVWF   TBLPTRH              ; table address reg
         MOVF    FL_READ_ADDR_L, W    ; low flash address into
         MOVWF   TBLPTRL              ; table address reg
940
         TBLRD   *                    ; initiate a table read
```

```
        MOVF    TABLAT, W           ;
        MOVWF   FL_READ_VAL         ; result into val register
944
        RETURN                      ;
        NOP                         ;

948  ;*********************** END READ FLASH ****************************;

     ;************************ ERASE FLASH *****************************;

952  erase_flash

     erase_row

956     BSF     EEC0N1, EEPGD   ; point to flash not eep
        BCF     EEC0N1, CFGS    ; access flash prog memory
        BSF     EEC0N1, WREN    ; enable writes to memory
        BSF     EEC0N1, FREE    ; enable row erase operation
960
        MOVLW   0x55            ;
        MOVWF   EECON2          ; write 55 hex as part of safety
        MOVLW   0xAA            ;
964     MOVWF   EECON2          ; write AA as part of safety

        BSF     EEC0N1, WR      ; start erase
        BCF     EEC0N1, FREE    ; disable row erase
968
        RETURN
        NOP

972  ;*********************** END ERASE FLASH **************************;

     ;************************** WRITE FLASH ***************************;
     write_flash
976
```

```
        CLRF    fl_count_bits        ; this block of code counts the high
        BTFSC   FL_LAST_RES, 0       ; bits to determine if the byte should
        INCF    fl_count_bits        ; be written in place or incremented
980     BTFSC   FL_LAST_RES, 1       ;
        INCF    fl_count_bits        ;
        BTFSC   FL_LAST_RES, 2       ;
        INCF    fl_count_bits        ;
984     BTFSC   FL_LAST_RES, 3       ;
        INCF    fl_count_bits        ;
        BTFSC   FL_LAST_RES, 4       ;
        INCF    fl_count_bits        ;
988     BTFSC   FL_LAST_RES, 5       ;
        INCF    fl_count_bits        ;
        BTFSC   FL_LAST_RES, 5       ;
        INCF    fl_count_bits        ;
992     BTFSC   FL_LAST_RES, 6       ;
        INCF    fl_count_bits        ;
        BTFSC   FL_LAST_RES, 7       ;
        INCF    fl_count_bits        ;
996
        MOVLW   0xFB                 ; number of bits set greater than 4?
        ADDWF   fl_count_bits, 0     ; if yes carry bit is set
        BTFSS   STATUS, C            ; check carry bit
1000    GOTO    last_result_wrong    ; if not then the last com failed
        NOP

    last_result_right
1004    MOVF    TBLPTRL, W           ; last com ok,
        ANDLW   0x07                 ; check if a long write is needed
        XORLW   0x07                 ;
        BTFSC   STATUS, Z            ;
1008    CALL    long_write_flash     ; if so, do a long write
        NOP             ;

    check_for_erase
```

```
1012    MOVF    TBLPTRL, W          ; check if an erase is needed
        ANDLW   0x3f                ;
        BTFSS   STATUS, Z           ;
        GOTO    short_write_inc     ; if not, do a short write
1016    NOP                         ;
        CALL    erase_flash         ; if yes, perform an erase
        NOP                         ;

1020 short_write_inc
        MOVF    FL_WRITE_VAL, w     ; load value
        MOVWF   TABLAT              ; perform short write
        TBLWT   +*                  ; increment after write
1024

        GOTO    verify              ; verify the last com

1028 last_result_wrong

     short_write
        MOVF    FL_WRITE_VAL, w     ; load value
1032    MOVWF   TABLAT              ; perform short write
        TBLWT   *                   ; do not increment

        GOTO    verify              ; verify the last com
1036


     ;************************* END WRITE FLASH **************************;
1040
     ;************************* LONG WRITE FLASH *************************;
     ;************** DO NOT ALTER, MANUFACTURER SPECIFIED ****************;

1044 long_write_flash
        BSF     EECON1, EEPGD       ; point to flash not eep
        BCF     EECON1, CFGS        ; access flash progie memory
```

```
        BSF     EEСON1, WREN        ; enable writes to memory
1048    BCF     INTCON, GIE         ; disable interrupts

        MOVLW   0x55            ;
        MOVWF   EECON2          ; write 55 hex as part of safety
1052    MOVLW   0xAA            ;
        MOVWF   EECON2          ; write AA as part of safety

        BSF     EECON1, WR          ; start long write
1056
        RETURN
        NOP

1060 ;************************ END LONG WRITE FLASH **********************;

     ;******************************* VERIFY *****************************;

1064 verify
        MOVF    SP_VAL_IN, W        ; load the transmit registers
        MOVWF   SP_CODE_OUT         ;
        MOVF    FL_LAST_RES, W      ;
1068    MOVWF   SP_VAL_OUT          ;
        CALL    serial_tx           ; perform a serial transmit
        NOP                         ;
        GOTO    fuel_map_loop       ; go around again
1072    NOP                         ;

     ;****************************** END VERIFY *************************;
```

Chapter 8

Testing, Inspection, and Verification

8.1 Overview

This chapter describes the tools and techniques used to verify the hardware and software components of the alternate fuel injection controller. The process was divided into laboratory testing, field testing, inspection, and verification. This chapter is divided along these lines as well.

8.2 Laboratory Testing

Laboratory testing, as it pertains to this system, refers to all activities performed without the use of an automobile, where the observable properties and behavior of the fuel injection controller are evaluated for conformance to the requirements specification and design chapters. This section describes the tools that were used to simulate inputs to the controller and to measure its responses. It also describes the tests that have been run and their results.

8.2.1 Laboratory Testing Tools

Provided here is a description of the tools employed during laboratory testing.

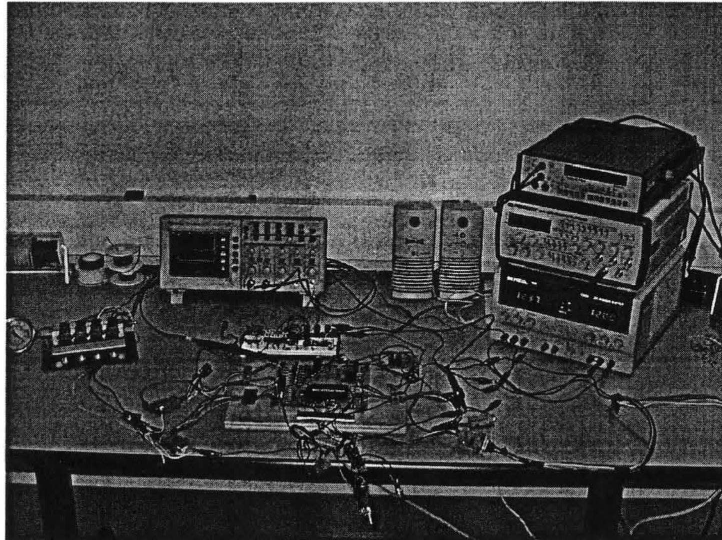


Figure 8.1: Picture of Lab Testing Environment

OnBoard Computer (OBC) hardware simulator

In the absence of an automotive OBC, it was necessary to devise a means of producing and delivering fuel injection type pulses to the controller. This was accomplished with the development of an OBC simulator that relies on a function generator for input and supplies 8 simulated injector signals. A square wave signal from the function generator is used as an input to a cascading CMOS chip. The chip has 8 output pins exactly one of which is always high. The high pin ordering is predetermined and the output is updated upon the detection of a rising edge on the input. Each output pin is anded with the signal from the function generator and then inverted. The resulting circuit delivers 8 signals closely resembling those sent from an automotive OBC to gasoline fuel injectors. The signals consist of pulses of equal length to those of the input square wave, but with $\frac{1}{8}$th the frequency.

Potentiometers

To fully test the controller required analogue signals that approximated those produced by sensors in the car. These signals were produced using potentiometers which were connected to the analogue inputs of the controller.

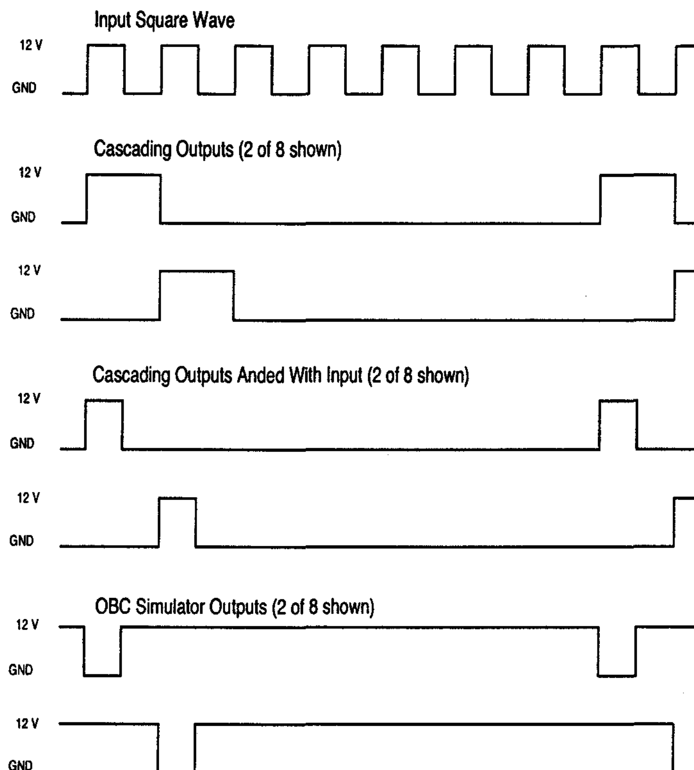


Figure 8.2: OBC Simulator Waveforms

Oscilloscope

An oscilloscope was used during testing to measure several properties of injector waveforms, analogue inputs, and serial communication signals. It was the most valuable tool for verifying the correctness of the controller's hardware components and provided measurements necessary to reason about conformance to timing and safety requirements. A multi-meter was used to a lesser extent for measuring voltage and current.

"Car Communicator" PC Application

The "Car Communicator" computer application is used to monitor and configure the alternate fuel injection controller. It is a valuable tool for both laboratory and field

testing. In the lab, the application was used to test the controller's serial communication functionality, fuel map programming, flash transactions, and A/D conversion. In the field it is used for data logging and monitoring the operating conditions of the automobile.

MPLAB® IDE and the PICSTART Plus

The MPLAB® Integrated development environment and the PICSTART Plus are used primarily for writing and delivering PIC micro assembly programs. They are also useful for testing flash read and write operations as together they provide a means of reliably reading the complete program memory of PIC micro-controllers.

8.2.2 Lab Tests and Results

Provided here are a list of lab test that were performed along with their results. Test are described in terms of tools employed, testing procedure and expected results.

Power Up Test

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope and the "Car Communicator" PC application.

Procedure

- With the OBC simulator delivering no pulses, the "Car Communicator" application monitoring the serial port and the i_fuelSelector switch set to *off*, supply power to the controller.
- Once two or more seconds have passed, activate the OBC simulator.
- Verify that pulses are being delivered on the correct outputs and that they are of appropriate timing and length.
- Repeat the test, this time with the OBC simulator active before the controller receives power.

- Repeat both of these tests again this time with the i_fuelSelector switch set to *on*.

Expected Results

In each case, the controller should first light o_nativeLED, o_stbyLED, and o_altLED. After a short period of time, (less than half of one second) only one of these LED's should remain lit. In the case that the i_fuelSelector switch was set to *off*, the LED that remains lit should be the o_nativeLED. In the case that the i_fuelSelector switch was set to *on*, the LED that remains lit should be one of o_stbyLED or o_altLED. The "Car Communicator" program should be updating correctly and injection pulses should be consistent with the input and mode of the controller.

Power Cycling Test

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope and the "Car Communicator" PC application.

Procedure

- With the OBC simulator delivering no pulses, the "Car Communicator" application monitoring the serial port and the i_fuelSelector switch set to *off*, supply power to the controller.
- Once two or more seconds have passed, activate the OBC simulator.
- Quickly cycle the power.
- Perform the test again with the controller in stbyNat mode, and a third time with the controller in altFuel mode.
- Repeat the tests cycling the power several times in rapid succession.

Expected Results

The results to expect are exactly those listed under the Expected Results heading of the Power Up Test.

Brown Out Test

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope and the "Car Communicator" PC application.

Procedure

- With the OBC simulator delivering no pulses, the "Car Communicator" application monitoring the serial port, and the i_fuelSelector switch set to *off*, supply power to the controller.
- Once two or more seconds have passed, activate the OBC simulator.
- Lower the voltage being delivered to the power supply of the controller to below 4.5 volts.
- Observe the behavior of the controller while "Browned Out".
- Bring the voltage back up to 12 volts.

Expected Results

When the system is being underpowered, output pulses to the fuel injectors should stop along with serial communication. The LED's should not be lit. Once the correct voltage is restored, the controller should behave as described under the expected results heading of the Power Up Test.

Mode Switching Test - init to program

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope and the "Car Communicator" PC application.

Procedure

- With the i_programmingSwitch in the *down* position, supply power to the controller.
- Hold the i_programmingSwitch in the *down* position for at least half of one second.
- Activate the OBC simulator and begin monitoring the serial port with the PC application.
- Toggle the i_fuelSelector switch.

Expected Results

Upon supplying power to the controller, the o_nativeLED, o_altLED, and the o_stbyLED should light for a period of time less than one half second. Once this time has passed, only the o_nativeLED and o_altLED should remain lit. The PC application should not be receiving data from the controller. Outbound injection pulses should be consistent with those expected in the native mode, regardless of the state of the i_fuelSelector and all other measured quantities. If power is supplied to the controller while the i_programmingSwitch is in the *up* position, and remains that way for at least one half second, the i_programmingSwitch should have no effect if later depressed.

Mode Switching Test - native to stbyNat

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope, and potentiometers.

Procedure

- Begin with the controller operating in the native mode and with the OBC Simulator delivering pulses.
- By adjusting the output frequency of the function generator, and the potentiometers, produce each general set of conditions that does not permit the controller to switch from the stbyNat mode to the altFuel mode. (not including the value of s_injectionStatus) These conditions can be determined by examining the stbyNat row of the Mode transition table of s_controlMode, found in the requirements specification chapter.
- Switch the i_fuelSelector to the *on* position.

Expected Results

At the beginning of the test, only the o_nativeLED should be lit and both o_heaterRelay and o_lockOffRelay should be *inactive*. Once the i_fuelSelector is in the *on* position, the o_nativeLED should turn off and o_stbyLED should light. Both the relays should become *active*. Outgoing pulses should not be effected by this mode transition.

Mode Switching Test - stbyNat to native

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope, and potentiometers.

Procedure

- Begin with the controller in the stbyNat mode and with the OBC Simulator delivering pulses.
- Switch the i_fuelSelector to the *off* position.

Expected Results

At the beginning of the test only the o_stbyLED should be lit, and both o_heaterRelay and o_lockOffRelay should be *active*. Once the value of i_fuelSelector is changed, the o_stbyLED should turn off and the o_nativeLED should light. Both relays should become *inactive*. Outgoing pulses should not be effected by this mode transition.

Mode Switching Test - stbyNat to altFuel

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope, and potentiometers.

Procedure

- Begin with the controller in the stbyNat mode and with the OBC Simulator delivering pulses.
- By adjusting the output frequency of the function generator, and the potentiometers, produce a set of conditions that causes the system to transition from the stbyNat mode to the atFuel mode. These conditions can be determined by examining the stbyNat row of the Mode transition table of s_controlMode, found in the requirements specification chapter.

Expected Results

At the beginning of the test, outgoing pulses should be consistent with the stbyNat mode, and only the o_stbyLED should be lit. Once operating parameters have been appropriately adjusted, the o_stbyLED should turn off and the o_altLED should light. Also, at this time pulses should be consistent with the altFuel mode.

Mode Switching Test - altFuel to native

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope, and potentiometers.

Procedure

- Begin with the controller in the altFuel mode and with the OBC Simulator delivering pulses.
- Switch the i_fuelSelector to the *off* position.

Expected Results

At the beginning of the test, only the o_altLED should be lit, both o_heaterRelay and o_lockOffRelay should be *active*, and pulses should be consistent with the altFuel mode. After the i_fuelSelector is moved to the *off* position, the o_altLED should turn off, and the o_nativeLED should light. Both relays should become *inactive*, and pulses should be consistent with the native mode.

Mode Switching Test - altFuel to stbyNat

Tools

Tools needed for this test include the OBC Simulator, the Oscilloscope, and potentiometers.

Procedure

- Begin with the controller in the altFuel mode and with the OBC Simulator delivering pulses.
- Adjust the voltage of the signal connected to the i_altFuelLevel input line so that the signal approximates one sent from a sensor on an empty tank.

Expected Results

At the beginning of the test, only the o_altLED should be lit, and pulses should be consistent with the altFuel mode. After the voltage adjustment, the o_altLED should turn off, and the o_stbyLED should light. Pulses should be consistent with the stbyNat mode.

Program Mode Test

Tools

Tools need for this test include the "Car Communicator" PC application, the MPLAB® IDE, and the PICSTART Plus.

Procedure

- Supply power to the controller with the i_programmingSwitch in the *down* position.
- Hold the i_programmingSwitch in the *down* position for another half second.
- Using the PC application, transfer a fuel map to the controller.
- Wait for the completion of the transfer which is indicated by the PC application.
- Remove the PIC from the controller and place it into the PICSTART Plus.
- Read the flash memory of the PIC using the MPLAB® IDE.
- Compare the flash memory starting at location 0x3E70, with the fuel map.

Expected Results

The values displayed in the MPLAB® IDE should exactly match those in the fuel map.

A/D and Serial Data Test

Tools

Tools needed for this test include the "Car Communicator" PC application, potentiometers, the oscilloscope, and a multi-meter.

Procedure

- Begin the test with the controller in the native mode.

- Using the multi-meter and the potentiometers, deliver specific known voltages to the analogue inputs of the controller (i_coolantTemp, i_o2Sensor, i_throttlePos, i_baroPres, i_airInTemp, i_altFuelLev).
- Start monitoring the controller with the PC application.
- Check the values being displayed for each analogue input.
- Adjust the voltages being supplied to the analogue inputs.
- Monitor the changes of the values displayed by the PC application.
- Repeat the test in the stbyNat mode.
- Repeat the test in the altFuel mode, this time measuring pulse elongations with the oscilloscope, and monitoring the displayed pulse elongation value.

Expected Results

Using the abstraction function $\mathcal{L}$, defined in the Implementation Description Document, verify that the values being displayed by the controller match the voltages being delivered on the analogue inputs. Also, ensure that changes in the voltage to any input are tracked by the PC application. In the final part of the test, concerned with the altFuel mode, verify that the displayed pulse elongation values are consistent with measured elongation lengths.

native and stbyNat Mode Injection Pulse Test

Tools

Tools needed for this test include the OBC Simulator and the oscilloscope.

Procedure

- Begin this test with the controller in the native mode.
- Activate the OBC simulator.
- With the oscilloscope, observe each o_altInjector signal.

- With the oscilloscope, observe each i_injector signal and the corresponding o_natInjector signal, where i_injector_k corresponds to o_natInjector_k $k = 1 \cdots 8$..
- Repeat the test in the stbyNat mode.

Expected Results

The signal on each o_altInjector should hold at 12 volts, corresponding to a value of *closed*. While noise on these signals is acceptable, the voltage should never drop below 7 volts. The o_natInjector signals should be the same as their corresponding i_injector signals but may be delayed by as much as 5 μs. It is not necessary for the o_natInjector signals to track noise present in the i_injector signals.

altFuel Mode Injection Pulse Test

Tools

Tools needed for this test include the OBC Simulator, the oscilloscope, and the "Car Communicator" PC application.

Procedure

- Begin this test with the controller in the altFuel mode.
- Activate the OBC simulator.
- Begin monitoring the controller with the PC application.
- With the oscilloscope, observe each o_natInjector signal.
- With the oscilloscope, observe each i_injector signal and the corresponding o_altInjector signal, where i_injector_k corresponds to o_altInjector_k $k = 1 \cdots 8$.
- Compare the observed elongation length with the elongation value displayed in the PC application.

Expected Results

The signal on each o_natInjector should hold at 12 volts, corresponding to a value of *closed*. While noise on these signals is acceptable, the voltage should never drop below 7 volts. The o_altInjector signals should ground within 5 μs of their corresponding i_injector signals. o_altInjection signals should return to 12 volts once an amount of time, equal to within 5 μs of the displayed pulse elongation, has passed since the corresponding i_injector signal returned to 12 volts. It is not necessary for the o_altInjector signals to track noise present in the i_injector signals.

altFuel Elongation value Test

Tools

Tools needed to perform this test include the "Car Communicator" PC application and potentiometers.

Procedure

- Using the values of i_o2Thresh and the i_tpThresh, choose specific regions of the controllers operating range. Since every combination of regions should be tested, at least 25 points in the operating range must be chosen.
- Compute the expected pulse elongation length.
- Begin monitoring the controller with the PC application.
- With the controller operating in the altFuel mode, use the potentiometers to simulate each chosen point in the operating range.
- For each point, compare the displayed and computed values of pulse elongation length.

Expected Results

The displayed and computed values should match exactly, provided that the points in the operating range can be simulated exactly. If this poses a problem, select a nearby point in the operating range that can be simulated exactly and repeat the test.

Fuel Level Test

Tools

Tools needed to perform this test include the "Car Communicator" PC application and potentiometers.

Procedure

- Using a potentiometer, vary the input voltage to i_altFuelLev.
- Compare the value displayed on the PC application with the value of o_altFuelLevelDisp.

Expected Results

The value of o_altFuelLevelDisp should be consistent with the value displayed for i_altFuelLev in the PC application.

Non-terminating Pulse Test

Tools

The oscilloscope is needed to perform this test.

Procedure

- Begin with the controller in the altFuel mode.
- Ground one of the i_injection inputs for a period of at least one half second.

Expected Results

The controller should switch to the native mode, which is demonstrated by output injector signals being sent on the o_natInjector output lines only and by o_nativeLED having the value *on*. The controller should not switch modes again until a reset occurs. o_stbyLED and o_altLED should flash until the controller has been reset.

Overlapping Pulse Test

Tools

Tools needed to perform this test are the oscilloscope, potentiometers, and the OBC simulator.

Procedure

- Begin with the controller in the altFuel mode.
- Set the duty cycle and frequency of the function generator, and the potentiometers in such a way that there is not sufficient time for pulse elongations to complete before the arrival of a subsequent pulse.

Expected Results

The results to expect are exactly those listed in in the Non Terminating Pulse Test.

8.3 Field Testing

Field testing, as it pertains to this project, refers to all activities performed using an automobile, fully equipped with the alternate fuel system, where the observable properties and behavior of the automobile and the controller are evaluated for conformance to the requirements specification and design chapters. It is important to note that in many cases the results of field tests will rely not only on the software design and implementation but also on the fuel map, and the values of the i_o2Thresh, i_o2Mult, i_tpThresh, and i_tpMult vectors. Seeing as the development of fuel maps, and the choosing of thresholds and multipliers falls outside the scope of this project, a field test failure is not necessarily indicative of a software design or implementation flaw. It is for this reason, that so much time and effort was spent on laboratory testing. This section lists the laboratory tests that can be slightly modified and performed as field tests. In addition to this, new tests specifically designed for the field are presented. In each case, tests are evaluated for their ability to be used as acceptance criteria for the design and implementation of the controller.

8.3.1 Field Tests and Results - Modified Lab Tests

The tests listed below have all been described in detail in the laboratory testing section of this chapter. This section describes changes to lab test procedures that will make them applicable field tests, along with their expected results. These tests were chosen based on their ability to be run in the field and their likelihood of rendering results inconsistent with lab tests.

Power Up Test

Changes to Procedure

When performing this test, supplying power to the controller is equivalent to turning the ignition key to the *auxiliary*, *on*, or *start* positions. A field test similar to the Power Up Test run in the lab is described below.

- Turn the key to the *on* position.
- Wait for at least half of one second.
- Attempt to start the automobile.
- Perform the test again, this time turning the key immediately to the *Start* position.

Expected Results

The expected results are exactly those listed for the in-lab version of this test.

Mode Switching Test - init to program

Changes to Procedure

This test can be performed as it was described for laboratory testing, where supplying power to the controller is done by turning the key to the *on* position, and turning the key to the *start* position initiates the transmission of injection pulses from the OBC.

Expected Results

The expected results are exactly those listed for the in-lab version of this test.

Mode Switching Test - native to stbyNat

Changes to Procedure

An equivalent mode switching field test procedure is presented here.

- Start the engine of the automobile.
- Switch the i_fuelSelector switch to the *on* position while the engine idles.

Expected Results

The expected results are exactly those listed for the in-lab version of this test.

Mode Switching Test - stbyNat to native

Changes to Procedure

Performing this test in the field involves idling the car in the stbyNat mode, and then switching the i_fuelSelector to the *off* position.

Changes in Results

The expected results are exactly those listed for the in-lab version of this test.

Mode Switching Test - stbyNat to altFuel

Changes to Procedure

An equivalent mode switching field test procedure is presented here.

- Start the test with the car idling and the controller in the stbyNat mode. The alternate fuel reserves should be near full.
- Allow the car to idle until such a time that the coolant temperature is greater than *THRESH_COOL_TEMP* degrees Celsius.
- Rev the engine to at least 1500 RPM.

Expected Results

The expected results include those listed for the in-lab version of this test. In addition to these, it is expected that the engine will continue to run smoothly on the alternate fuel. This expectation, however, can not be defined as an acceptance criterion as the performance of the engine depends highly on fuel maps, thresholds, and multipliers.

Mode Switching Test - altFuel to native

Changes to Procedure

Performing this test in the field involves idling the car in the altFuel mode, and then switching the i_fuelSelector to the *off* position.

Expected Results

The expected results are exactly those listed for the in-lab version of this test. To successfully perform this test, the automobile must be capable of smoothly idling while burning the alternate fuel. If the test can be performed, the results can be used as acceptance criteria since the operation of the car after the last step of the testing procedure has been performed does not depend on fuel maps, thresholds or multipliers.

Mode Switching Test - altFuel to stbyNat

Changes to Procedure

Performing this test in the field involves idling the car in the altFuel mode, and then waiting for the alternate fuel reserves to be mostly depleted.

Expected Results

The expected results are exactly those listed for the in-lab version of this test. To successfully perform this test, the automobile must be capable of smoothly idling while burning the alternate fuel. If the test can be performed, the results can be used as acceptance criteria since the operation of the car after the last step of the testing procedure has been performed does not depend on fuel maps, thresholds or multipliers.

A/D and Serial Data Test

Changes to Procedure

This test requires a tool that has not yet been mentioned. Automotive scan tools interface directly with an automobile's on-board computer and are capable of displaying values for nearly every engine operating parameter. When performing the A/D and Serial Data Test in the field, the only difference is that an automotive scan tool is used to provide a basis for comparison rather than a multi-meter.

Expected Results

The values displayed by the PC application should match those displayed by the automotive scan tool.

8.3.2 Field Tests and Results - New Tests

The tests described in this section have been designed to evaluate the performance of an automobile equipped with the alternate fuel system. These tests should only be performed once all lab tests, and their field counterparts have been successfully completed. The results of these test rely heavily on the quality of fuel maps and post multipliers, therefore, they will not be considered as acceptance tests.

Idle Test

Procedure

Idle the car in the altFuel mode.

Expected Results

The engine should idle with a steady RPM value approximately equal to the gasoline RPM idle value. The i_o2Sensor reading should oscillate around the value that indicates trim operation. There should be no bias toward rich or lean combustion.

Acceleration Test

Procedure

With the car idling in the altFuel mode, apply throttle steadily until the car reaches a speed of 20 km/h. Repeat the test for several different cruising speeds.

Expected Results

In each case, the car should accelerate without hesitation and smoothly maintain its cruising speed.

Full Throttle Test

Procedure

With the car idling in the altFuel mode, quickly apply full throttle. Keep the throttle in the full open position until the car reaches a cruising speed of 100 km/h. Back off the throttle and maintain speed.

Expected Results

The car should accelerate without hesitation and smoothly maintain its cruising speed.

Extreme Temperature Test

Procedure

Perform the idle, acceleration, and full throttle field tests with ambient temperatures as close to −30°C as possible, and again with ambient temperatures as close to 40°C as possible.

Expected Results

Test results should not change with temperature.

8.4 Inspection

The inspection process involved a thorough manual examination of the controller implementation code, performed in an effort to increase confidence in the software, and to unearth any remaining defects that had not been detected through testing. The code was also examined to ensure conformance to the coding conventions outlined in the implementation description chapter. This section describes the criteria that drove the inspection process.

8.4.1 Explicit Banking

No implementation instruction may have the explicit bank switch set.

The register space of the PIC18FXX2 series of micro-controllers, consists of 16 banks of 256 registers. Since registers are addressed in the code with only 8 bits, it is left to the developer to ensure that the bank select bits are appropriately set before any operation is performed that reads from or writes to a register. However, explicit banking can be avoided through use of the "access bank". This is a virtual bank that consists of the lower 128 bytes of bank 0, and the upper 128 bytes of bank 15. This is useful since the lower bytes of bank 0 are the most commonly used registers for storing user defined variables, and the upper bytes of bank 15 store the special function registers. As the access bank is used for the controller implementation, it is necessary to inspect the code to ensure that no instructions are written with the explicit bank switch set. Exclusive use of the access bank has the disadvantage of limiting the number of usable general purpose registers to 128. Fortunately, the limit did not pose a problem for this implementation.

8.4.2 Variable Addressing

All user defined variables must be declared with addresses in the first 128 bytes of register bank 0.

For a user defined variable to be addressable in the access bank, it must have an address in the first 128 bytes of bank 0. As the access bank is being used to address registers, it is necessary to inspect variable declarations to ensure that all variable address are within this range.

8.4.3 GOTO ISR Instructions

No GOTO instruction outside the interrupt service routine may cause execution to jump to any instruction inside the interrupt service routine. The interrupt service routine is the code that is responsible for the most important control action performed by the system, the opening and closing of fuel injectors. This code has been designed with the assumption that it would only be executed in response to an interrupt. It is, therefore, necessary to inspect the code to ensure that no situations exists where a GOTO from outside the ISR causes execution to continue inside the ISR.

8.4.4 Call Stack Size

The size of the call stack must never exceed 31.

The PIC18FXX2 series of micro-controller has a call stack with a maximum size of 31. In the event that the call stack has a size of 31 and a call is made or an interrupt occurs, the call stack overflows and the device is reset. Inspection is necessary to ensure that no execution path exists that makes this situation a possibility.

8.4.5 Flash Write Instruction Sequence

The flash write "required sequence" provided in the PIC18FXX2 data sheet[10] must appear unaltered in the implementation code.

To perform a flash program memory write requires the execution of 4 consecutive specific instructions. As flash program memory writing is necessary, the code was inspected to ensure that the sequence appeared unaltered in the flash program segment.

8.4.6 Variable Naming and Code Casing

All variable naming and code casing conventions outlined in the implementation description chapter of this document must be adhered to in the implementation code.

In the interest of code readability and maintainability it is important to follow

these simple code conventions. For this reason, the implementation code was inspected for proper variable naming and code casing.

8.5 Verification

The software development process which produced the injection controller software did not involve mathematically rigorous verification. However, it was deemed necessary to verify that the controller implementation and design do satisfy certain important requirements. This section will reason about the correctness of the software with respect to the requirements chosen for informal verification. Design verification is based on the flowcharts presented in the high level design chapter, while implementation verification is based on the code found in the implementation chapter. Since the design did not deal with timing constrains, verification of timing requirements will only be discussed with respect to implementation.

8.5.1 Verification of Safety Requirements

Once the system is initialized, exactly one fuel injector shall fire for every incoming injection pulse.

Design Verification

The diagrams that are relevant to the design verification of this requirement are figures 6.3, 6.4, 6.7, and 6.9. In figure 6.3 the design dictates that when initialization is completed, the mode of the system is native, both interrupts and native injectors are enabled, and each member of the o_altInjector array has the value *closed*. From here, execution will continue in either the Programming Loop or in the Main Loop. The programming flow chart, figure 6.9, dictates that immediately following a mode transition to program, interrupts are disabled and remain so until the system is reset. Therefore, once the system is in the program mode, there is no execution path to either the initialization routine or the interrupt service routine. Since each alternate injector has the value *closed* when entering this mode, and these values are only changed in the interrupt service routine and during initialization, it can be stated that no incoming injection pulses received while the system is in the program mode, will arrive at an

alternate injector. Since native injectors are enabled when entering the program mode, and are never disable therein, it can be safely stated that any injection pulse received while the system is in the program mode will arrive at a native injector. Therefore, the software design satisfies this requirement while the system resides in the program mode. In figures 6.4 and 6.7 the design dictates that as soon as the mode of the system becomes native or stbyNat, and all o_natInjector and o_altInjector signals have the value *closed*, the native injectors are enabled. Therefore, any injection pulse that arrives, while the controller is in either of these modes, will reach a native injector. The design also dictates that native injectors are disabled immediately before the mode of the system transitions to altFuel. Since native injectors are not enabled by any other subsequent program segment, it can be stated that a pulse received while the controller is in the altFuel mode will not arrive at a native injector. It now remains to be shown that any incoming injection pulse received while the mode of the system is not altFuel will not arrive at an alternate injector, and that any incoming injection pulse received while the mode of the system is altFuel will arrive at an alternate injector. These properties can both be verified by examining figure 6.7. In this figure, the code that opens the alternate injectors is guarded by the condition that the mode of the system is altFuel. It can therefore be stated that the design does satisfy this requirement.[1]

Implementation Verification

Implementation verification of this requirement involved checking for correspondence between the design and the implementation with respect to the following criteria.

- Immediately after the initialization routine is complete, the system mode is native, both native injectors and interrupts are enabled, and all alternate injectors are closed.(Lines 22, 69-82, 240, 246, 248)
- Alternate injector values are not set outside of the interrupts service and initialization routines.(Lines 548-1072)

[1]It should be noted that a race condition exists that can result in a single injection being missed, or an injection pulse arriving at both native and alternate injectors, immediately following a mode transition. It has been determined that this is an acceptable deviation from the requirements.

- Interrupts become and remain disabled immediately after the system enters the program mode. (Line 802)
- Native injectors remain enabled while the system resides in the program mode.(Lines 803-888, 932-1072)
- Native injectors are enabled when the system enters the native or stbyNat modes.(Lines 248, 504, 564-570, 591-596, 664-669)
- Native injectors are disabled when the system enters the altFuel mode.(Lines 624-629)
- In the interrupt service routine, alternate injectors are opened when, and only when, the system resides in the altFuel mode. (Lines 466-467)

Fuel injectors fired must be associated with the same cylinders as incoming injection pulses.

Design Verification

The association between input injection signals and native fuel injectors is managed in hardware. The correct association between incoming injector signals and outgoing alternate injector signals is assumed by the software design and is verified below.

Implementation Verification

Verification of this requirement, relies on the assumption that the controller is wired in such a way that for all k, the alternate fuel injector associated with PORTD[k] operates on the same cylinder as the native fuel injector associated with PORTB[k]. Also, since not all pins of PORTB generate interrupts on change, it was necessary to send the conjunction of all incoming injection signals to one of the pins that does. The pin chosen for this task was PORTB[7]. This means that while the arrival of an incoming injection pulse to PORTB[k], $k = 0 \cdots 6$, is indicative of a pulse bound for a fuel injector associated with PORTB[k] , the same cannot be said for PORTB[7]. The arrival of an injection pulse to PORTB[7] is only indicative of a pulse bound for an injector associated with PORTB[7] when all other bits of PORTB have the value

0. Therefore, verification of this requirement involved checking for the following properties in the code.

- When a change on PORTB interrupt occurs, and PORTB has the value 10000000_2, PORTD is assigned the value 1000000_2.[2]
- When a change on PORTB interrupt occurs, and PORTB has 2 high pins, PORTD is assigned the value (PORTB & 01111111_2).[3]

In all situations where s_trouble = *on*, fuel injection control shall default to the car computer.

Design Verification

The only flow chart necessary for design verification of this requirement is figure 6.7. This is the only segment of the design where s_trouble becomes *on*. It can be seen from this flow chart that when s_trouble is set, the native injectors are enabled, and all alternate injectors are closed. Furthermore, since the design dictates that the controller waits for a reset, there is no possibility of these conditions changing.

Implementation Verification

Implementation verification of this requirement involves a very small section of the code. The value of s_trouble is only set to *on* in one place, inside the interrupt service routine (Line 505). Immediately before this happens, the native injectors are enabled, and all alternate injectors are closed (Lines 500, 504). Shortly thereafter, the code loops indefinitely, never disabling the native injectors, and never opening an alternate injector (Lines 508, 544). It can therefore be stated that the implementation satisfies this requirement.

[2]The 7th bit of a port is the MSB of the binary representation.
[3]'&' is the bitwise and operator.

The lock off relay must remain open in the native and program modes

Design Verification

For design verification of this requirement, it is assumed that the initialized value of o_lockOffRelay is *inactive*. It can then be determined, that when the control mode of the system is program, o_lockOffRelay is never written, and thus remains *inactive* until the controller is reset(see figure 6.9). There are three situations in the design where the mode of the system becomes native. These situations are depicted in figures 6.3, 6.7 and 6.4. In each case, the value of o_lockOffRelay is *inactive* before the mode transition, and remains that way until the mode becomes stbyNat or altFuel. It can therefore be stated that the design satisfies this requirement.

Implementation Verification

Implementation verification of this requirement involved checking for correspondence between the design and the implementation with respect to the following criteria.

- During variable initialization, the value of o_lockOffRelay is set to *inactive*.(Line 31)
- The value of o_lockOffRelay is *inactive* immediately before any mode transition to native.(Line 556)
- o_lockOffRelay is never written when the control mode of the system is program or native.(Lines 503-544, 560-561, 572-577, 675-791, 803-888, 932-1072)

Pulse elongations must not overlap with subsequent native injection pulses.

Design Verification

The detection of an overlapping pulse causes the system to close all alternate injectors and enable native injectors as is seen in figure 6.7. Unfortunately, one pulse elongation must violate this requirement before action can be taken. This has been identified as an acceptable deviation from the requirements.

Implementation Verification

Implementation verification of this requirement involved checking for correspondence between the design and the implementation with respect to the following criterion.

- If a change on B interrupt occurs as a result of an arriving pulse, and TIMER2 is running, then the system must transition to the native mode. (Lines 477-478, 503)

8.5.2 Verification of Timing Requirements

The controller shall be capable of detecting and generating pulses to keep an 8 cylinder engine at up to 6500 RPM.

Implementation verification of this requirement was performed by ensuring that the controller timers were initialized as discussed in the implementation chapter. (Lines 41-55)

The controller shall be capable of detecting and generating pulses to keep an 8 cylinder engine at as few as 500 RPM.

Implementation verification of this requirement was performed by ensuring that the controller timers were initialized as discussed in the implementation chapter. (Lines 41-55)

Every injection pulse generated by the car computer must arrive at the destination cylinder within 5μs.

Assuming that signal propagation time is negligible, verification of this requirement is only necessary for alternate injection pulses. This is a direct result of native injection pulses being managed in hardware. Verification for alternate pulses depends on each of the following issues:

- Frequency of oscillation
- Interrupt generation speed
- The maximum number of instructions needed to deliver an injection pulse

The PIC micro-controller is being run at a frequency of 40 MHz, and therefore completes single cycle instructions at a rate of 10MHz. Jumps to the Interrupt service routine occur within two cycles of the event which causes them. This means that as much as $.2\mu s$ may pass before the program jumps to the ISR. This leaves $4.8\mu s$ or the time it takes to execute 48 single cycle instructions before the appropriate alternate injector must be opened. The maximum number of instruction cycles, counting from the start of the ISR, needed to open an alternate injector is 17 and well within the 48 cycle limit.

Chapter 9

Conclusion

This chapter presents conclusions pertaining to the success of the controller and the documentation strategy. Also provided are discussions of possible future work and proposed changes to the development process.

9.1 Controller Evaluation

When evaluating the controller, it was important to consider not only the results of the testing, inspection, and verification, but also the sentiments of the industry contact from Cosimo's Garage Ltd. Fortunately, these measures of success agreed in their endorsement of the controller as an acceptable final product. While some field testing, relying on final versions of fuel maps remains to be completed, lab tests and verification suggest that the controller satisfies the requirements specification presented in chapter 3.

9.2 Documentation Strategy Evaluation

Evaluation of the documentation strategy is more difficult than that of the controller itself. This section provides a description of how documents in the strategy provided, or failed to provide benefits to those involved in the development process.

9.2.1 The Requirements Specification

When deciding on the organization and representation of the requirements that would eventually constitute the specification, there were two main considerations. These were client readability and comprehension, and the ability of the document to support the remainder of the development process.

In an effort to support understanding by the industrial partner, the requirements were organized in a style similar to the one in which they were delivered. The organization grouped the requirements specifying the necessary behavior of the system separately from those specifying the negation of unacceptable behavior. It is my opinion that this organization facilitated client comprehension and did so without hindering the design or verification of the controller software.

In regards to development support, it is my opinion that the specification which aimed to be output focused, tabular, complete, and consistent proved very successful. As the specification was output focused, it directly supported testing and verification which was performed in part by comparing observable system outputs to specified outputs. The tabular nature of the specification not only increased readability but also supported development by helping to identify incompleteness and inconsistency during requirements elicitation. Finally, the benefit of completeness and consistency in the specification was most clearly demonstrated during the first field test which was highly successful, and also by a minimal need for communication with the industrial partner following the completion of requirements elicitation.

9.2.2 High Level Design Document

Flow charts were selected for use in the software design for their ability to express timing and sequencing information, their support of abstraction, and the fact that their semantics can be easily expressed. Upon completion of the development process it has been determined that flowcharts did provide the expected benefits. Timing requirements were among the most important to the success of the project. Since all cycles in the design are visualized in flowcharts, time bound analysis was quickly directed to the most timing sensitive areas. The ability of flowcharts to express program segment sequencing, reduced the amount of effort required to bring the project from design to implementation. It also eased to task of design/implementation verification

since these two work products exhibited similar structure. Flowchart abstraction, or the representation of a single program segment component as a separate flowchart, increased readability and facilitated fast design navigation.

9.2.3 Implementation Description

The implementation description, illustrates the connections between the state space variables and their implementation counterparts, and provides interfaces in terms of global variables for each software unit. In addition to this, the chapter provides the implementation code. Evaluation of the implementation, which was performed through testing, inspection, and verification of the controller software, has been completed and is well documented. Evaluation of the implementation chapter is a separate issue. For the content and structure of this chapter to be considered a success, it must be able to support others whose role it is to maintain the software. Since no one has of yet begun this task, the evaluation of this chapter remains incomplete.

9.3 The Evolution of the Development Process

Upon completion of a software development project, such as the one described in this document, it is important to consider how certain problems could have been avoided by changes to the development process or documentation style. While the development of the fuel injection controller did proceed very smoothly, there were certain aspects that could have been improved. The most notable of these were the need for late changes to the requirements specification, and evaluation criteria which depended on fuel maps that have not yet been completed. Both of these problems could have derailed the project had development been driven by monetary compensation rather than completion of degree requirements. For this reason, any future development would include the specification of a date, after which, no changes to the requirements could be made without the agreement of the developers and the client. Also, all acceptance criteria would be based on factors which do not depend on untested systems or documents produces by others.

9.4 Reuse

In developing the fuel injection controller, it was necessary to specify, design and develop sub routines to provide services relating to serial communication, flash reading and writing, analogue to digital conversion, and interrupt handling. Seeing as the interfaces to these routines are well defined and their implementation code is self contained, there is great potential for reuse in future PIC applications. Also, since the development platform chosen, the PIC18F series, is the latest generation of PIC micro controllers, it is very likely that the platform will be available for some time to come.

9.5 Future Work

Through discussions with the industry contact, a list of future work has been compiled, that if completed would ease the task of installers and fuel map developers. The list is presented here.

- Real time visualization of the operational data transmitted by the controller.
- Offline fuel map optimization based on logged data.
- Improved fuel map development tools.
- USB communication support.

THE END

Index

$\mathcal{L}$ mapping, 65

A/D Conversion, 103
  assumptions, 103
  code, 104
  variables, 103
access bank, 133
alternate injector transistor bank, 39
analogue converter level adjustment, 37

branch, 45

Car Communicator, 114
circuit diagrams, 40
coding conventions, 60, 64, 133
components, 43
connectors, 43
controlled quantities, 17

documentation strategy, 12, 142
  decomposition, 13
  high level design, 13
  implementation, 14
  interface specification, 14
  requirements specification, 12
    organization, 13
  verification, 14
driver interface, 39
duty cycle, 62

elongation factor, 24
events, 19
explicit banking, 133

Flash, 106
  assumptions, 106
  code, 107
  variables, 106
flowchart, 13
Flowcharts, 43
  enable/disable native injectors, 51
  general program structure, 45
  initialization program segment, 46
  interrupt service request handler, 51
  mode update and data acquisition main-loop, 48
  PC programming segment, 56
  programming program segment, 56
  TIMER 1 Overflow, 52
flowcharts, 143
frequency of oscillation, 62
fuel injection, 2
fuel map, 24
future work, 144

general purpose registers, 60
GOTO, 134

hardware design, 34

Hoffman and Strooper Method, 8
- Module Guide, 9
- Module Interface Specification, 10
- Module Internal Design, 10
- Requirements Specification, 9

IEE Guideline, 6
- Feasibility Study, 7
- Functional Specification, 7
- Software System Specification, 7
- Test Documents, 8
- User Requirements Specification, 7

implementation, 60
information hiding, 14
Initialization, 68
- assumptions, 68
- code, 71
- variables, 68

injection window, 63
inspection, 133
interrupt branch, 43
invariants, 18
ISR, 78
- assumptions, 78
- code, 80
- variables, 78

label, 45

Main Loop, 89
- assumptions, 89
- code, 90
- variables, 89

mapping, 60
monitored quantities, 16
MPLAB IDE, 115
multipliers, 25

native diode bank, 39
native injector resistor bank, 38
native injector transistor bank, 39
natural gas, 4

OBC simulator, 113
oscillator, 61
oscilloscope, 114

petroleum, 4
PIC 18F452 micro-controller, 60
PIC configuration, 61
- configuration bits, 61
- timers, 62
  - TIMER 1, 62, 63
  - TIMER 2, 62, 64

PIC support, 39
PICSTART Plus, 115
pin assignments, 35
pin diagram, 36
post-scaling, 64
pre-scaling, 64
program segment, 45
Programming Loop, 98
- assumptions, 98
- code, 99
- variables, 98

propane, 4
prototype, 39

race condition, 136

recoverable, 56
Requirements
  communication, 32
  display, 30
  driver interface, 27
  Fuel Map, 31
  injector grounding, 22
  logging, 30
  mode switching, 21
  numerical representation, 26
  OBC diagnostics, 28
  operating environment, 20
  platform, 32
  Safety, 31
  safety, 19
  serial connection, 27
  timing, 20, 32
requirements, 15
reuse, 144

sequential flow, 44
serial level converter, 39
Serial Port, 100
  assumptions, 100
  code, 101
  variables, 101
Software Cost Reduction, 11
  Four Variable Model, 12
  IN, 12
  NAT, 12
  OUT, 12
  REQ, 12
  Requirements, 11
  tabular notation, 11
special function registers, 60
state charts, 13
state space variables, 65
state variables, 17
success, 56

Testing
  field testing, 127
  laboratory testing, 112
  laboratory testing tools, 112
thresholds, 25
trouble, 22

unrecoverable, 57

Verification, 135
  safety requirements, 135
  timing requirements, 140

watchdog timer, 61

Bibliography

[1] Alan C. Shaw, *Real-Time Systems and Software.* John Wiley and Sons, Inc., 2001.

[2] Brett Duane, *Migrating Designs from PIC16C74A/74B to PIC18C442.* Microchip Technology Inc., 1999. Preliminary.

[3] Constance L. Heitmeyer, "Software cost reduction," *Encyclopedia of Software Engineering*, Jan. 2002.

[4] Daniel Hoffman and Paul Strooper, *Software Design, Automated Testing, and Maintenance.* International Thomson Computer Press, 1995.

[5] Frank Vahid and Tony Givargis, *Embedded Systems Design, A Unified Hardware/Software Introduction.* John Wiley and Sons, Inc., 2002.

[6] James E. Duffy, *Modern Automotive Technology.* Goodheart-Willcox, 1994.

[7] John Becker, "Pic16f87x mini tutorial," *Everyday Practical Electronics/ETI*, pp. 742–748, Oct. 1999.

[8] Microchip Technology Inc., *PICmicro Mid-Range MCU Family Reference Manual*, ds33023a ed., 1997.

[9] Microchip Technology Inc., *PIC16F87X Data Sheet, 28/40-Pin 8-Bit CMOS FLASH Microcontrollers*, ds30292c ed., 2001.

[10] Microchip Technology Inc., *PIC18FXX2 Data Sheet, High Performance Enhanced FLASH Microcontrollers with 10-Bit A/D*, ds39564b ed., 2002.

[11] The Institution of Electrical Engineers, "Guidelines for the documentation of computer software for real time and interactive systems," tech. rep., The Institution of Electrical Engineers, 1990.

1305 24