HARDWARE REQUIREMENTS FOR FUZZY LOGIC CONTROL SYSTEMS

by

MARK WORKMAN, B.S.E.T.

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Accepted

December, 1996

ACKNOWLEDGEMENTS

Special thanks go to:

Dr. Donald Gustafson for all your time, assistance and guidance throughout this project. You always gave me the freedom (and will) to pursue my own thoughts and ideas. Yet, you always remained my ever attentive tutor. And to my other committee members, my thanks go to Dr. Noe Lopez-Benitez for your direction, support and ceaseless encouragement and to Dr. Hua Li for your great suggestions and advice.

A special thanks to my lovely wife, Diana. You are my never ending confidante and advocate! I can never describe how much you mean to me. Together, I look forward to entering the next arena of our life and I know that the best is yet to come!

Thanks to my family: Irene, Don, Skeet, David, Iven, Ruby, Clayton, Nina and Melissa. All of you have been the greatest. To each and everyone of you, thanks!!! Especially to Charlie. How lost would I have been without your direction? You were tough, but you were always right.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii
ABSTRACT vii
LIST OF TABLES ix
LIST OF FIGURES x
LIST OF EQUATIONS xii
CHAPTER
I INTRODUCTION 1
Thesis Problem Description 1
Absence of FUDGE Memory Models 2
Absence of FUDGE Processing Models 2
Absence of FUDGE Support for the C++ Language 3
Objectives of Thesis 3
Development of FUDGE Hardware Models 5
Development of FUDGE C++ Code Support 6
Introduction to FUDGE 6
II TOPIC SURVEY 8
Fuzzy Logic Control Systems 8
History of Fuzzy Logic 8
A Description of a Fuzzy Logic System 9
Bivalent Systems versus Fuzzy Logic Systems 10
Humans and Fuzzy Logic 10
Fuzzy Logic: Perfect for Small Memory and Low Processing Power Applications 12
Fuzzy Logic: Degrees of Truthfulness 15
Current State of Fuzzy Logic 18
The Fuzzy Logic Engine 19

III RESEARCH METHODOLOGY 22
Implementation of Research Methodology 23
Fuzzy Engine Hardware Models 23
C++ High Level Language Support 24
Description of Examples 25
Washing Machine Example 26
Traffic Light Example 31
Truck Backing Example 37
IV RESEARCH RESULTS 44
Fuzzy Engine Structure 44
Knowledge Base 45
Fuzzy Inference Processor 46
Fuzzy Arrays 46
Hardware Requirement Models for the Fuzzy Engine 46
Fuzzy Engine Models 48
Memory Models 48
68HC05 Fuzzy Engine 48
Memory Requirements for Motorola 68HC05 Microcontrollers 50
Memory Requirements for C and C++ Applications 52
Processor Power Models 53
68HC05 Processing Power Requirements 53
Fuzzification Cycles 56
Rule Evaluation Cycles 58
Defuzzification Cycles 61
XFUDGE Translator 62
Benefits of XFUDGE 62
XFUDGE Programs 64
XFUDGE_D.EXE 64

XFUDGE_W.EXE 65
XFUDGE.EXE 65
XFUDGE C++ Implementation 65
Example Testing and Evaluation of Results 69
C and C++ Fuzzy Engine Comparisons 69
68HC05, C and C++ Hardware Model Tests 69
V CONCLUSION 73
Hardware Prediction Models 73
Microcontroller Support 73
Hardware Model Benefits 74
XFUDGE: C++ Object-Oriented Support 74
Object-Oriented Benefits 74
Further Fuzzy Logic Research Topics 75
Extended FUDGE Hardware Model Support 75
Incorporate FUDGE C++ Support 75
Improvements to FUDGE Fuzzy Inference Processor 76
Summation 76
REFERENCES 77
APPENDIX
A. XFUDGE TRANSLATOR MAIN SOURCE CODE 78
B. XFUDGE TRANSLATOR CLASS DEFINITIONS 82
C. XFUDGE TRANSLATOR CLASS FUNCTIONS 84
D. XFUDGE STRING CLASS DEFINITIONS 90
E. XFUDGE STRING CLASS FUNCTIONS 91
F. XFUDGE HEADER FILE 94
G. ANSI C IMPLEMENTATION OF WASHING MACHINE 95
H. ANSI C IMPLEMENTATION OF TRAFFIC LIGHT 98
I. ANSI C VERSION OF TRUCK BACKER-UPPER 102
J. ANSI C VERSION OF FUZZY.H HEADER FILE 106

K. C++ VERSION OF WASHING MACHINE 107
L. C++ VERSION OF TRAFFIC LIGHT 109
M. C++ VERSION OF TRUCK BACKER-UPPER 112
N. C++ FUZZY ENGINE CLASS DEFINITION 115
O. C++ FUZZY CLASS DEFINITIONS 117
P. 68HC05 CODE FOR WASHING MACHINE 122
Q. 68HC05 CODE FOR TRAFFIC LIGHT 123
R. 68HCO5 CODE FOR TRUCK BACKER-UPPER 126
S. 68HC05 CODE FOR FUZZY INFERENCE PROCESSOR 129
T. XFUDGE GUI INTERFACE GLOBAL PROCEDURES 137
U. XFUDGE GUI INTERFACE FORM 140
V. XFUDGE GRAPHICAL USER INTERFACE 151

ABSTRACT

The FUzzy Design GEnerator (FUDGE) is an inexpensive, but surprisingly powerful, fuzzy logic design tool. It can be used to develop, test and implement fuzzy logic controllers in a wide variety of applications. So, it is the purpose of this thesis to evaluate and improve this fuzzy logic design tool. This thesis also discusses several topics related to FUDGE that are either hard to find or have not been thoroughly documented by Motorola.

Chapter I gives a overall introduction to goals and ambitions of this thesis, which include the development of some hardware requirement models for fuzzy logic control systems developed with the FUDGE environment. Plus, the development of a C++ translation program. This translation program provides object-oriented, C++, support for the FUDGE tool.

Chapter II provides a basic overview of fuzzy logic. It begins by discussing the past historical developments of fuzzy logic systems. Then it covers some of the current attitudes and misconceptions about using fuzzy logic in control system applications. This is followed by a primer on the goals and benefits of implementing control systems with fuzzy logic. Included in this discussion is the implementation of fuzzy logic systems with Binary Input-Output Fuzzy Associative Memories (BIOFAMs) and rule inference with the Max-Min composition relation. For a more in depth study of theoretical fuzzy logic design, the reader is referred to such excellent text books as Bart Kosko's, Neural Networks and Fuzzy Systems.

Chapters III and IV describe the goals, expectations and results of this research and can be best described in two major topics. The first topic is the development of hardware models for fuzzy logic control systems implemented with the FUDGE software. These models can be used to predict the memory and processing power requirements needed to implement a proposed fuzzy logic design. The second portion relates to increasing the number of high level languages that are supported by the FUDGE tool. Since FUDGE is both a design and implementation tool, it can create the output code

necessary to implement a fuzzy logic design in several forms of microprocessor. The current version of FUDGE (Version 1.02) supports several of Motorola's assembly languages, as well as the ANSI C language. In this second topic, a fuzzy logic translation program is also described. This program translates the source code for a C based fuzzy engine (produced by FUDGE) into a functionally equivalent C++ based fuzzy engine object. This allows a designer to implement a fuzzy logic design in the high level languages of C or C++.

Chapter V contains a summary of the work done in this thesis. It reviews the hardware models for memory allocation and processor execution delays, followed by an overview of the XFUDGE translation software and its contribution to the Fuzzy Design Generator.

LIST OF TABLES

1. Assumed Number of Bytes for Data Types 48
2. Project Files for Thesis Examples 49
3. 68HC05 Specification Limitations 55
4. Execution Cycles for Fuzzification of Crisp Inputs 57
5. Execution Cycles for Antecedent Conditions 60
6. Execution Cycles for Consequence Conditions 60
7. Execution Cycles for Output Membership Functions 62
8. Execution Cycles and Time Delays for Fuzzy Engine Examples 72

LIST OF FIGURES

1. Fuzzy Engine Description 4
2. Microcontroller Description 5
3. FUDGE Description 7
4. Washing Machine Controller Example 11
5. Comparisons between Controller Implementation Styles 15
6. Bivalent Water Level Example 16
7. Water Level Membership Function 17
8. FUDGE Software Parameters 19
9. Fuzzy Engine Block Diagram 20
10. Fuzzy Engine Component Structure 21
11. Fuzzy Logic Design Parameters 24
12. XFUDGE Translator Block Diagram 25
13. Fuzzy Washing Machine Controller 27
14. Washing Machine Membership Functions 29
15. Washing Machine Rules 30
16. Traffic Light Membership Functions 33
17. Traffic Light Rules 34
18. Truck Backer-Upper Diagram 37
19. Truck Backer-Upper Membership Functions 39
20. Truck Backer-Upper Rules 40
21. Fuzzy Engine Structural Block Diagram 45
22. Fuzzy System Specifications 47
23. Fuzzy Input Membership Function 57
24. XFUDGE Translation Procedure 63
25. Available XFUDGE Program Descriptions 64
26. XFUDGE Command Line Options 65
27. Example Fuzzy Engine Knowledge Base File 67

28. Fuzzy Class Definition 68
29. 68HC05 RAM Bytes for Three Fuzzy Engine Examples 70
30. 68HC05 ROM Bytes for Fuzzy Engine Examples 71
31. XFUDGE Graphical User Interface 151

LIST OF EQUATIONS

Eq. (1) 68HC05 ROM Memory Requirements 51
Eq. (2) 68HC05 RAM Memory Requirements (Model 1) 52
Eq. (3) 68HC05 RAM Memory Requirements (Model 2) 52
Eq. (4) C Memory Requirements 53
Eq. (5) C++ Memory Requirements 53
Eq. (6) Total Fuzzy Engine Execution Cycles 55
Eq. (7) Fuzzy Engine Execution Delay Calculation 55
Eq. (8) 68HC05 Clock Cycles for Fuzzification 58
Eq. (9) 68HC05 Clock Cycles for Antecedent Evaluation 61
Eq. (10) 68HC05 Clock Cycles for Consequence Evaluation 61
Eq. (11) Clock Cycles for Defuzzification 62

CHAPTER I

INTRODUCTION

This thesis is dedicated to the analysis, enlightenment and progression of fuzzy logic design. Specifically, it focuses on the hardware requirements (both memory allocation and execution delays) for fuzzy logic control systems designed with Motorola's Fuzzy Design Generator. It also expands the programming languages supported by this design tool to include: assembly, C and now C++. In short, it provides the design engineer with several tools to simplify and automate the design of fuzzy logic control systems.

Thesis Problem Description

The FUzzy Design GEnerator (often referred to by its acronym FUDGE) is an inexpensive, but surprisingly powerful, fuzzy logic design tool. It can be used to develop fuzzy logic systems for control applications. These applications can be implemented into one of Motorola's 68HC05 series of microcontrollers with a low level assembly language or into more complex systems with the high level C programming language.

Currently, the FUDGE tool provides the control system designer with an integrated development environment in which to create, test and implement fuzzy logic control applications. However, there have been no hardware models developed for predicting memory size or processing throughput for fuzzy logic controllers developed in the FUDGE environment, which means that these important hardware specifications cannot be predetermined before a control application is developed. The FUDGE tool also does not provide any support for the popular C++ programming language, which prevents the design engineer form utilizing the benefits offered by object oriented software design.

Absence of FUDGE Memory Models

An important design characteristic is the amount of memory required to implement a fuzzy logic design into some processing element. Currently, designers have no modeling tools to help them determine the amount of memory needed to implement their design into a 68HC05 microcontroller. Without this model, a designer cannot predetermine the amount of memory that will be required by the fuzzy engine implementation, which can present a problem when implementing fuzzy logic controllers into the limited available memory of the 68HC05 series of microcontrollers.

The lack of a memory model can also cause problems with the C language implementation of fuzzy logic systems. Currently, a designer has no easy means of determining how much memory is required for high level language implementation. While memory is not normally an issue in the larger, more powerful systems used to implement C programs, the designer is still interested in how much memory will be used by the fuzzy logic system. In particular, they are specifically interested in how the memory requirements vary as the system design changes. Questions arise like: If they change the number of system inputs in the system, will there be exponential change in the amount of memory required for the C language implementation?

Absence of FUDGE Processing Models

Another important design characteristic is the processing performance (throughput) of a fuzzy logic controller design. This is especially important at the low processing speeds required by the 68HC05 series of microcontrollers. Note that the 68HC05 series of microcontrollers typically run at frequencies less than 4.20 MHz. So, the throughput of large, complex fuzzy logic design might not be able to keep up with the real-time control system application. Currently, there are no models to predict the processing throughput (processing delays) of fuzzy logic controllers implemented with the 68HC05 series of microcontroller.

Absence of FUDGE Support for the C++ Language

Currently, ANSI C is the only high level programming language supported by the FUDGE environment, which limits the choices available to the system designer when implementing a fuzzy logic control system. Also, the C language does not easily or uniformly relate to the symbolic fuzzy logic design process. This fact can actually increase the overall complexity of the fuzzy logic control application. While in contrast, the object-oriented C++programming language is naturally adapted to the fuzzy logic design methodology.

Object-oriented programming and fuzzy logic complement each other by simplifying the design process. The C++ object-oriented language relates software design to natural real-world objects. Similarly, fuzzy logic relates natural real-world descriptions of objects to control system designs. In effect, both design methodologies combine to relate discrete computer processes to the designer's symbolic and analog thought processes. This results in the designer being able to quickly and easily develop robust control system applications. But, most importantly, these two methodologies provide an easy connection between the system designer and the fuzzy logic implementation by enhancing the natural symbolic interface between the designer and the design.

Objectives of Thesis

The research in this thesis develops hardware models to predict the requirements for fuzzy logic control systems implemented with Motorola's FUzzy Design GEnerator (FUDGE). This includes memory and processing power models of controllers developed for the 68HC05 series of microcontrollers or for the more complex processors that utilize the C programming language. In addition, it also expands the high level programming languages supported by this tool. This expansion is accomplished by adding C++ (object-oriented) to the list of high level programming languages supported by FUDGE.

One of the largest gaps in the FUDGE development process lies in the hardware requirements needed when implementing a fuzzy logic control systems. Therefore, this

thesis will develop memory and execution models for fuzzy control systems developed with the FUDGE tool. Specifically, it includes several formulas, tables and diagrams that can be used to predict memory requirements and processing performance for fuzzy controller implementations.

To help illustrate typical hardware requirements, three fuzzy logic controller example applications have been developed. These fuzzy logic control system examples include: a washing machine controller, a traffic light controller and a truck backing controller. Each example control system was developed in the FUDGE environment and then evaluated for its specific hardware requirements. Afterwards, the results from the three example systems are used to test and confirm the memory and performance models.

All of these controller implementations are selected because of their ability to demonstrate the use and benefits of the hardware models developed for the FUDGE tool. The models developed from these examples can be applied to a wide variety of system types. For example, the models can be applied to high processing power, large memory devices (such as a personal computer) or applied to microcontrollers with limited amounts of memory and less processing power.

A fuzzy engine is the mechanism in which a fuzzy controller relates system input states to the appropriate system output states.

Figure 1. Fuzzy Engine Description

To implement a design, FUDGE creates a unique software element called a "fuzzy engine."[5] This fuzzy engine is the heart of the fuzzy logic control system, and it is currently created by the FUDGE environment as assembly or ANSI C code. If the system designer wants to implement a fuzzy engine using C++ object-oriented code, this document also contains a C to C++ code translator called XFUDGE. The benefits of C++ code include: object-oriented design, data encapsulation (data protection), and most

of all, it greatly simplifies the programmer's interface to the fuzzy engine. The C++ version of a fuzzy engine manages all these improvements with little or no performance degradations over the original C code implementation. The XFUDGE translation software is not part of the original FUDGE development tool. It has been added as an external program to provide C++ support for the FUDGE environment.

Development of FUDGE Hardware Models

The first portion of this research consist of developing hardware models (memory allocation and processing power, respectively) for fuzzy logic control systems implemented with the FUDGE software. These models can be used by any engineer to predict the memory and processing power requirements needed to implement a proposed fuzzy logic application. These models cover fuzzy controllers implemented in one of Motorola's 68HC05 series of microcontrollers or possibly a larger more complex microprocessor found in a personal computer (i.e., 8086, 80386, i486, Pentium, etc.).

A microcontroller is a small, self-contained processing element. It contains a low processing power microprocessor and a small amount (about 4k to 8k bytes, typical) of onboard memory. All of which is completely integrated and packaged as a single unit.

Figure 2. Microcontroller Description

The microcontroller is designed and marketed as an "all-in-one" controller. It contains a simple microprocessor with built in ROM and RAM memory. Plus, it has basic input/output capabilities and a peripheral interface all integrated into one package. A microcontroller is particularly useful in the design of small inexpensive control applications. The models in this thesis will be especially beneficial to any designer trying to implement fuzzy logic control systems into these inexpensive microcontrollers.

Development of FUDGE C++ Code Support

The second portion of this research relates to increasing the number of high level languages supported by the FUDGE tool. Since FUDGE is both a design and implementation tool, it can create the output source code to implement a fuzzy logic system into some form of microprocessor. The current version of FUDGE (Version 1.02) supports several of Motorola's assembly languages, as well as the ANSI C language. However, it does not produce object-oriented (C++) source code. Hence, a translation program has been created to allow designers to utilize the C++ language with their fuzzy logic implementation.

This thesis describes and demonstrates this translation program, called XFUDGE. The XFUDGE software converts the fuzzy engine (C source) code created by the FUDGE program into functionally equivalent (C++ source) fuzzy engine code. This C++ code (a "fuzzy" class) allows the design engineer to implement a fuzzy engine design with the popular C++ object-oriented language.

Introduction to FUDGE

It is not the intention of this thesis to address the design and implementation of very large and complex fuzzy control systems. So, it will focus on Motorola's simple and inexpensive (free) FUzzy Design GEnerator (FUDGE). The FUDGE software was developed as a basic development and educational tool for fuzzy logic systems. It provides an easy to use, Windows 3.1, graphical user interface for the design and evaluation of simple fuzzy control systems. In FUDGE, users can create a fuzzy logic application and then implement their design into a personal computer or a variety of Motorola's microcontrollers.

FUDGE is a development tool for designing, testing and implementing fuzzy logic control systems.

Figure 3. FUDGE Description

The FUDGE tool generates assembly code for several of Motorola's microcontrollers and/or standard ANSI C code. This code can then be used to implement a fuzzy engine into a real-time control system application. This fuzzy engine will respond to a controller's inputs and produce the appropriate outputs based on the system's design specifications. These specifications are implemented into FUDGE by the system designer. FUDGE is available as freeware from Motorola's microcontroller web page. FUDGE Internet address:

http://129.38.232.2/freeweb/pub/fuzzy/fuzdsk3.zip

CHAPTER II

TOPIC SURVEY

Fuzzy Logic Control Systems

Fuzzy logic is a powerful, but often overlooked, design philosophy for describing and developing control systems. It provides a simple and intuitive method for design engineers to implement complex systems in everyday terminology. As many engineers are discovering, it also can produce more efficient and economical solutions than other, more traditional control system implementations[3]. Often times, it allows a control system to be designed and built for systems that could not be easily modeled or designed using other, more traditional design methodologies.

History of Fuzzy Logic

Fuzzy logic has been around since the mid-1960s when the father of fuzzy logic, Professor Lotfi A. Zadeh of U. C. Berkeley, first described such systems and coined the word "Fuzzy"[3]. Yet, only recently, has it received more than scant attention outside the academic word. Most professionals in the engineering world seemed to have been turned-off by the imprecise and inaccurate sounding name, "fuzzy math" or "fuzzy control." They preferred to stick to their time tested and well proven traditional models and mathematical formulas to design control systems. It seems that the power, versatility and ease of designing fuzzy systems into control applications had gotten lost behind the funny, fuzzy sounding name.

Then in the late 1980s something began to change. Several Japanese electronics corporations began releasing consumer products boasting of "Fuzzy Logic" control. It was the surprising success of these products that sparked the renewed interest in the field of fuzzy logic design. Although fuzzy logic has quickly gained acclaim and acceptance in the East. Where the English term "Fuzzy" has quickly been hailed as meaning "smart," or having the ability to" think like a human" [3]. However, the West has approached fuzzy logic with a more cautious and skeptical eye. Some Western engineers mark fuzzy

logic off as simply being a passing "fad," while still other engineers are waiting for the "new kid on the block" to prove itself in applications where traditional forms of design have normally been used before.

It has only been recently that fuzzy logic has been recognized and supported by a few, far-sighted and progressive companies. As major corporations, like Motorola, come on-line with new products developed with, or for, fuzzy logic systems, fuzzy logic will certainly begin to gain recognition and acclaim in the design community. But of course, only time will tell if fuzzy logic is just a passing "fad" or if it will become a part of the accepted main stream design process.

A Description of a Fuzzy Logic System

Fuzzy logic is itself a tool. It allows a designer to implement a control system using more symbolic, "human-like," thought processes and terms. In fact, there is nothing "fuzzy" about fuzzy logic. It is actually a branch of mathematical set theory [8] and provides a sound numerical foundation for handling expert knowledge[1]. This ability to evaluate expert knowledge and to derive an exact answer from imprecise (fuzzy) input data is what makes fuzzy logic so powerful.

With fuzzy logic this can be done without the need to formulate complex mathematical models or look-up tables. These traditional forms of mapping system outputs from system input values can be difficult, time-consuming or even impossible for some control systems. Also, solving complex mapping equations (typically differential equations) requires a great deal of processing power and look-up tables require large amounts of memory, while fuzzy controllers can handle even highly non-linear control mapping without the need for great processing power or large amounts of memory. This provides a valuable commodity in engineering: compromise! A fuzzy engine requires only moderate processing power and a few simple look-up tables to operate. This allows the designer to use a less expensive, lower power microprocessor. Thus, the use of fuzzy logic can reduce both development time and hardware cost.

Bivalent Systems versus Fuzzy Logic Systems

In conventional bivalent logic systems, a particular piece of input data either is or is not a member of some output set. These systems usually evaluate input data into TRUE or FALSE values indicating the state (or set) that an input belongs too [3]. While these types of bivalent systems are perfect for computers, they do not conform well to describing natural systems. Since most natural systems are made up of many shades of grey, they often exist within several different descriptive states (sets) at any one time. So, they must be described by the degree (truthfulness) at which they are a member of some group of states. Thus, fuzzy logic systems allow for an input to exist (with varying degree) in more than one state at a time. This allows the engineer to describe the system in more natural terms.

Humans and Fuzzy Logic

Humans relate to the world around them in symbolic terms. In contrast, computers require exact numerical or mathematical information in order to relate to the world around them. In the past, the burden of converting these natural symbolic terms into such computer friendly terms has been placed on the system designer. Now fuzzy logic provides a method for the design engineer to describe a system's operation in human-like symbolic terms and then easily convert those symbols into a from that can be used by the computer.

As humans, we relate to physical objects in symbolic mental images. These images represent objects as we perceive them through our five senses. For example, we do not naturally relate the size of a child in terms of their exact physical dimensions or determine the child's growth by figuring the derivative of the rate of change in their height. Instead, humans process a child's height more like: the child's size is "larger than a babe but smaller than an adult" or characterize the kid's growth rate as "growing like a weed." Of course, we as humans use this symbolic type of processing. This allows our own bodies and minds to quickly process and react to input stimuli, which is the exact thing a computer must do in a real-time control system. Yet, when engineers design a

control system, they must convert all these symbolic terms into exact numerical or mathematical commands that can be understood by a computer.

Simple examples like determining the length of time to wash a load of clothes in a washing machine can present a designer with a complex mathematical relationship[7]. As a human, the engineer knows how long to wash the clothes by relating to their past wash load experiences of similar types and amounts of dirt.

So, assume that the clothes are *not very dirty* and the type of dirt is *non-greasy*. Then the clothes only need to be washed for a *short time*. However, if the same clothes were *not very dirty* but the type of dirt is *greasy*, then the wash time would need to be lengthened to some *longer time*. If the clothes were both *very dirty* and the type of dirt was *very greasy*, then the clothes would need to be washed for a *very long time*. This sets up a classical input/output control system problem, as shown in Figure 4.

Washing Machine

Problem:

Determine the length of time to wash the clothes based on the amount and type of dirt.

Solution:

- If the amount of dirt is *not very dirty* and the type of dirt is *non-greasy*, then wash the clothes for a *short time*.
- If the amount of dirt is not *very dirty* and the type of dirt is *greasy*, then wash the clothes for a *longer time*.
- If the amount of dirt is *very dirty* and the type of dirt is *very greasy*, then wash the clothes for a *very long time*.

Figure 4. Washing Machine Controller Example

But to implement such a control scenario into some processing unit, the designer must develop either an exact numerical lookup table or mathematical function to relate wash load inputs to washer time outputs, which means that they must deal with all of the human (symbolic) terms that they have used to develop the system. In any case, the system designer must determine the exact expected wash time outcome for every conceivable combination of dirtiness and type of dirt. Then the designer has to go through the lengthy experimental process of determining values for a lookup table or try to develop an adequate function to model the system's operation. In either case, the designer is now working outside of the familiar and natural symbolic environment in which the brain works best. Ironically, it is the fact that such harsh numerical and mathematical restrictions are placed on the designer that has inhibited the common acceptance of fuzzy logic in the design community.

Actually, fuzzy logic is a methodology for producing more accurate, realistic and efficient solutions to complex control systems, while relieving the design engineer from the tedious, time (and money) consuming details of developing such numerically intensive control methods. Fuzzy logic provides a convenient interface between humans symbolic thought processes and the numerical descriptions required by computers. As defined by Bart Kosko, "Fuzzy systems directly encode structured knowledge but in a numerical frame work" [2]. This allows for a designer to quickly implement the symbolic control operation into a system using a fuzzy design technique. The result is a control system that still conforms to the systems operation requirements but takes much less time to develop.

Fuzzy Logic: Perfect for Small Memory and Low Processing Power Applications

In conventional microprocessor control systems, inputs are mapped to outputs by one of two primary methods. First, either the controller has a precompiled lookup table (stored in RAM or ROM memory) that relates an input state to the appropriate output

state. Second, the controller performs some mathematical transform function that relates the input condition to the correct output condition.

In the lookup table implementation of a control system mapping, a microprocessor is provided with a table that contains all the possible input to output correlations. When the controller senses a change in its input, it looks up the correct output condition for that input state. The controller then responds with the preprogrammed response. Since most microprocessors take only a few clock cycles to retrieve a value from a lookup table, this implementation allows the controller to quickly respond to changes at its inputs. Thus, the controller only has to wait a few clock cycles before knowing how to respond to an input condition. This can be a great advantage in real-time control systems where the controller must almost instantly respond to changes in the system.

However, these types of systems require a large amount of memory to implement a table that contains every single input/output condition. So, systems incorporating lookup tables must have access to large amounts of RAM or ROM memory. This also requires a great deal of preplanning to insure that all possible input/output conditions are implemented into the table. The functional testing of all the possible input/output combinations can also be time or cost prohibitive. (NOTE: Remember the division error in the lookup table for the Intel Pentium processor in 1995.) So, lookup tables for large complex systems can have the advantage of increased system response time. Yet, they require large amounts of computer memory and can be difficult to implement, test and maintain.

Some control systems use mathematical mapping functions to calculate the appropriate output for a particular input state. These systems require that a microprocessor recalculate an output state for every change in the input state. This type of system implementation does not need the large amount of RAM or ROM memory required to store large lookup tables. They only need enough memory to store the mapping formula and any required temporary variables. While this type of system implementation does not require a large amount of RAM or ROM memory, it does

require a significant amount of processing power to calculate the output result. This means a processor that can handle large complex mathematical operations must be selected. Thus, the microprocessor must execute many instructions to solve the input/output mapping function. This can result in delays between input state changes and output responses. A control system must either compensate for these processing delays or operate at higher clock frequencies. Also, a design engineer must be able to mathematically model a control system in order to develop an input/output mapping function. This can require a great deal of effort since not all systems can be modeled by simple mapping functions.

So, in the two traditional control system implementations, an engineer must either allow for large amounts of memory to incorporate lookup tables or choose a processor that can handle the computation needs of the mapping function. In both cases, the control system cost is increased. Fortunately, fuzzy logic provides for a workable compromise between large memory and high processing power requirements.

Fuzzy logic control systems utilize both lookup tables, as well as mathematical functions to implement input/output mappings. Yet, both are used on a significantly smaller scale than in either of the traditional implementations. In fuzzy logic systems, small lookup tables are used to hold the fuzzy input and output functions and relational rules. Then these simple rules are executed in order to perform input/output conversions. These conversions do not require a lot of heavy duty processing. This means that a fuzzy system must have enough memory to hold these tables and enough processing power to do these simple conversion calculations.

So fuzzy logic provides a control system platform that allows for a workable compromise between large precompiled lookup tables and complex mathematical mapping functions, thus allowing a designer to design hardware systems that require small amounts of memory and can still operate in microprocessors with amounts low processing power, as symbolized in Figure 5. The return for the design engineer that implements fuzzy logic into their controller is quicker development, decreased testing requirements and lower system costs.

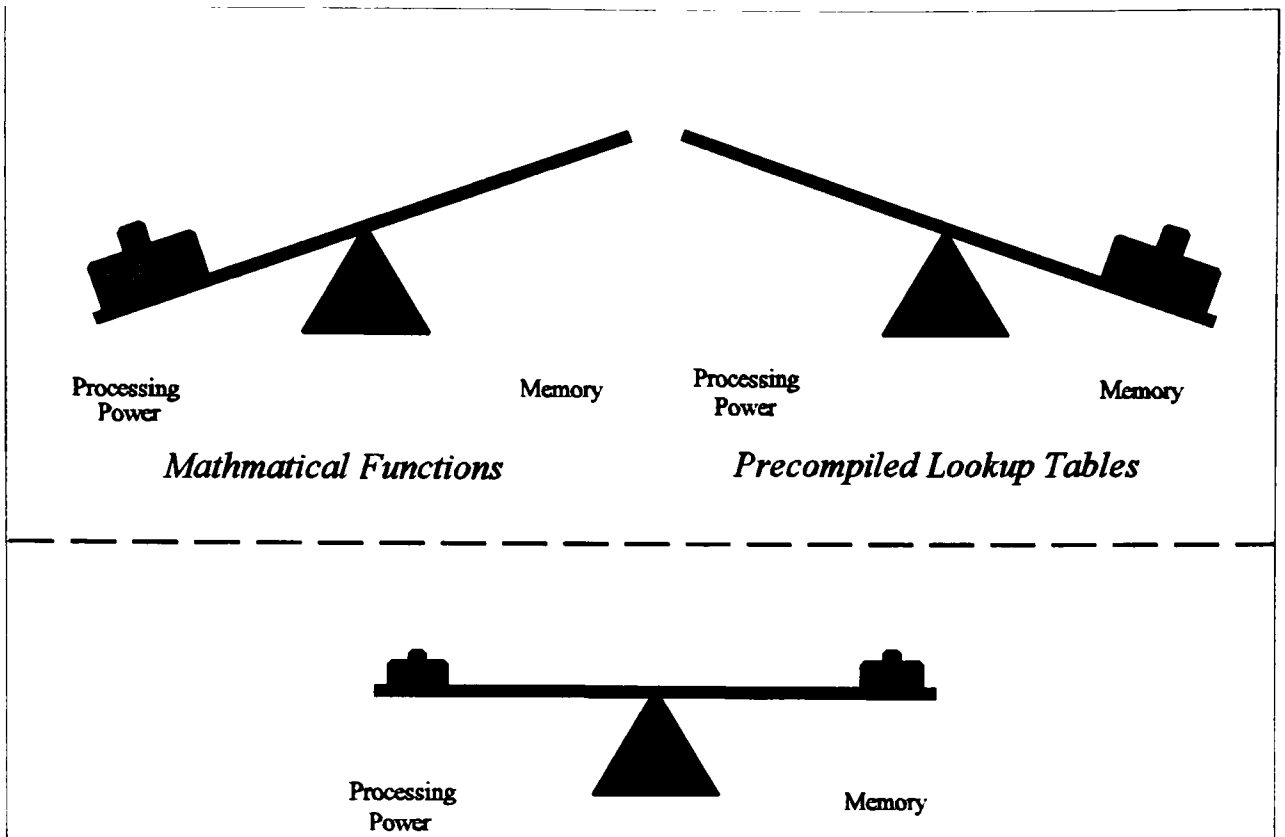


Figure 5. Comparisons between Controller Implementation Styles

Fuzzy Logic: Degrees of Truthfulness

Take for example, a glass of water. The glass can be described as being *Full*, *Half* or *Empty*. A typical bivalent system would measure the height of the water level in the glass. If the level were at the top of the glass, then the system would say that the glass was *Full*. Then over time the water would evaporate out of the glass. Soon, enough water will have evaporated out of the glass that the system would quickly declare the level in the glass as *Half*. Then, one magical day, more water molecules would evaporate out of the glass, and the system would instantly declare the glass *Empty*. As shown in Figure 6, this type of system signal looks very much like something an engineer's digital computer would understand. Yet, I do not think the engineer's mother would understand it nearly as easily.

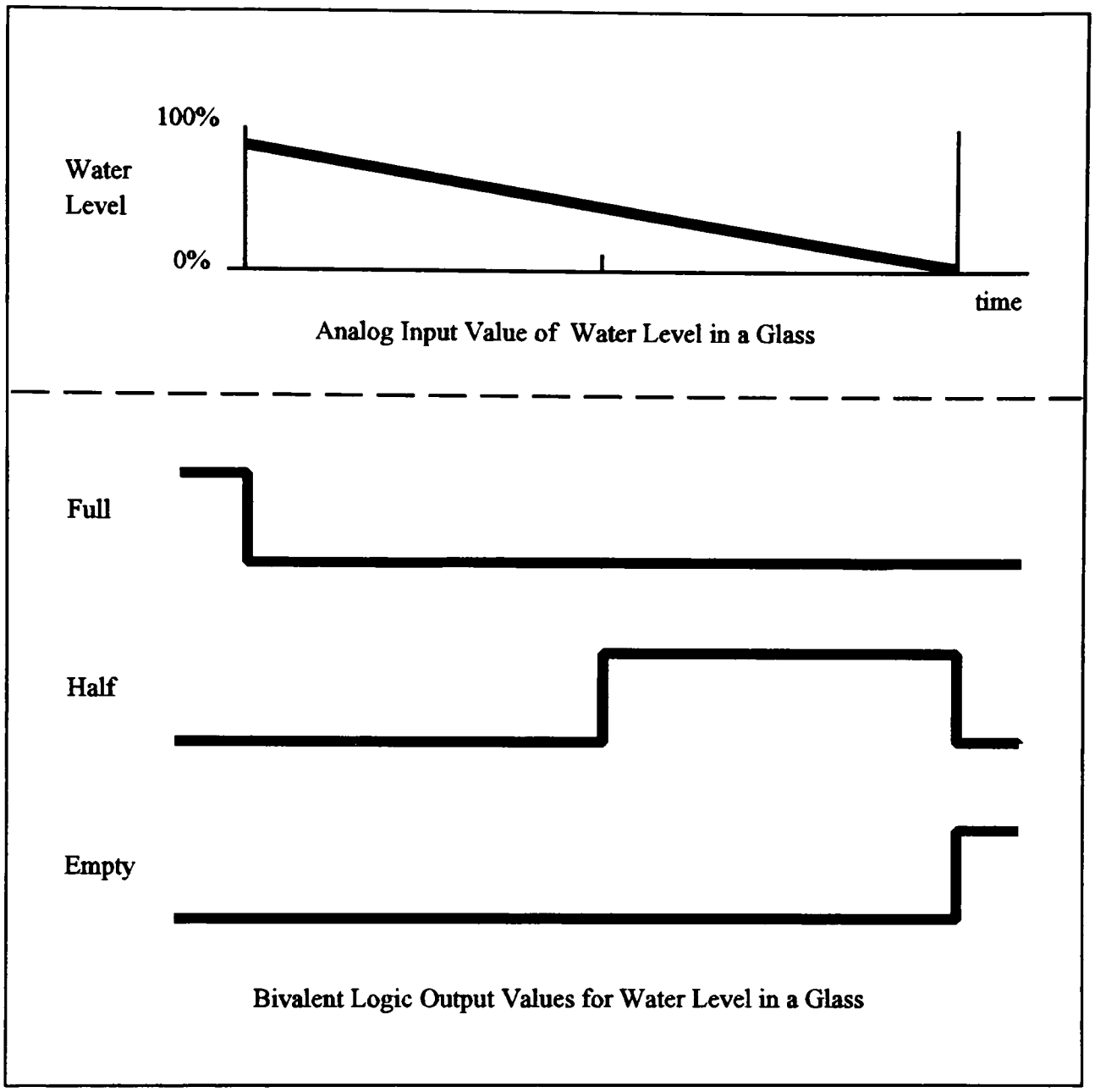


Figure 6. Bivalent Water Level Example

In a fuzzy logic system, the water level could be described by its varying degree of membership within the three levels. When the glass is *Full* then it is 100% part of the *Full* state, but it is also 0% in the *Half* and *Empty* states. As water evaporates, the level becomes less a part of the *Full* state and more a part of the *Half* state. When the water level reaches the 3/4 level. The fuzzy system will show the level to be 50% *Full*, 50% *Half* and 0% *Empty*. At half full, the system would be 0% *Full*, 100% *Half* and 0% *Empty*. At 1/4 full, the system would declare that the glass was 0% *Full*, 50% *Half* and

50% *Empty.* Finally, the glass would reach 0% *Full,* 0% *Half* and 100% *Empty.* When the water level was at the 1/2 marker, one output member may declare the glass as being "half-full," while another member might declare the glass as being "half-empty." Which output member is correct? Well, in fuzzy logic, they both are correct. These varying or sliding membership degrees allow a fuzzy system to make some very interesting output statements. Furthermore, a fuzzy system can even make some more interesting statements, such as, at a level of 5% *Full,* 95% *Half* and 0% *Empty,* the fuzzy output would show that the system was "slightly half-full," which the computer interprets as being 0.5 units *Full*, 0.95 units *Half* and 0.0 units *Empty.* When the level reached 0% *Full,* 95% *Half* and 5% *Empty,* the fuzzy output would declare that the system was "slightly half-empty." The computer interprets the level as being 0.0 units *Full*, 0.95 units *Half* and 0.5 units *Empty.* Since the fuzzy outputs are able to declare an output in human terms like, "slightly," "mostly," or "almost," the designer simply thinks about and describes the system in these imprecise terms. It is this ability to express human-like observations that makes fuzzy logic so powerful. The overlapping of each member function is shown in Figure 7.

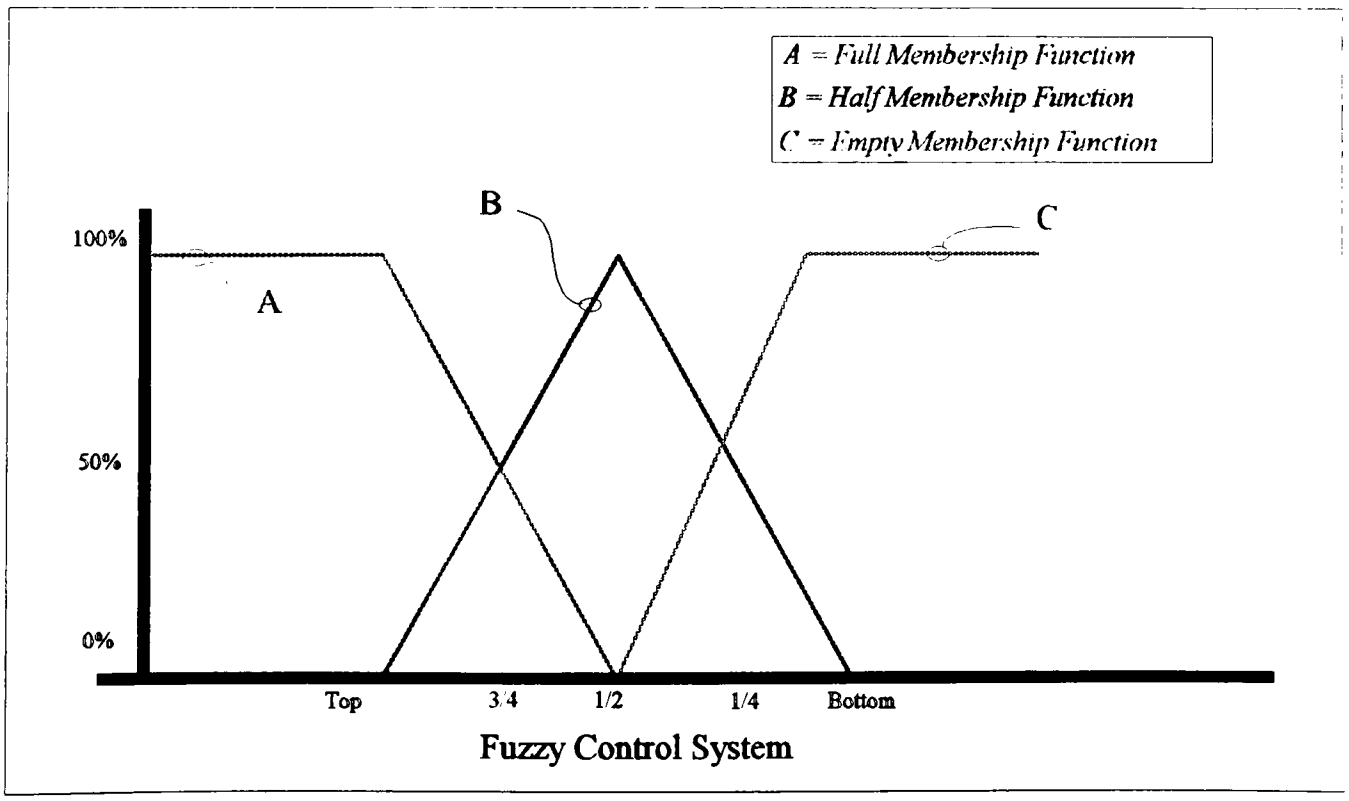


Figure 7. Water Level Membership Function

Current State of Fuzzy Logic

One reason that fuzzy logic has not been considered in many applications is because many design engineers lack the informational resources to design fuzzy systems. This is mainly because fuzzy logic lacks a well documented, proven and structured design process. Most design engineers are used to having several design tools readily at their disposal. each with a plethora of documents and examples to explain how the tool can best be used. However, such tools for fuzzy logic design development have only been available for a few years. Tools like Aptronix's Fuzzy Inference Development Environment (FIDE) and Motorola's FUzzy Design GEnerator (FUDGE) have since begun to gain industry acceptance.

Motorola's FUDGE software is an inexpensive but useful tool for developing simple fuzzy logic control systems, sometimes called fuzzy controllers or fuzzy engines. FUDGE provides the designer with an easy to use graphical interface. Within this environment the design engineer can graphically build a fuzzy control engine, then modify the system inputs and graphically view the resulting changes at the output, thus allowing the engineer to easily tweak the system into design specifications. Figure 8 shows a brief outline of FUDGE's parameters.

While the development environment of FUDGE lacks many of the sophisticated graphics capabilities of FIDE, it still allows an engineer to design, build and evaluate a surprisingly complex fuzzy control system. One of its features is the ability to export a fuzzy design into compilable C or assembly source code. The user simply selects the type of code to generate, (C-language or assembly code) and FUDGE generates the code for the fuzzy controller. This feature gives the designer the ability to implement a design without having to be an expert programmer. The assembly code is especially beneficial for developing software in many of the small, inexpensive and low processing power systems being built by engineers.

FUDGE Parameters

System Inputs:

- 8 inputs maximum
- 8 membership functions per input
- Trapezoidal or Triangular functions

System Outputs:

- 4 outputs maximum
- 8 membership functions per output
- Singleton functions

System Rules:

- 1000 rules maximum
- 8 antecedents per rule
- 4 consequences per rule

Figure 8. FUDGE Software Parameters

The Fuzzy Logic Engine

The FUDGE implementation of the fuzzy engine represent an example of a Binary Input-Output Fuzzy Associative Memory (BIOFAM). According to Kosko [2], A BIOFAM accepts binary (crisp) data as input. It converts this crisp input data to a set of fuzzy values, called fuzzy vectors. These fuzzy input vectors are processed by the fuzzy engine, and the results stored as a set of fuzzy output vectors. Finally, the fuzzy output vectors are converted back to binary (crisp) output data values.

The BIOFAM fuzzy engine implementation contains three basic processes. They are fuzzification, rule evaluation and defuzzification, as shown in Figure 9. During fuzzification process the Fuzzy Inference Processor (FIP) relates crisp input values to fuzzy input vectors by simple relational conversions. The FIP then uses rule evaluation to map fuzzy input vectors to fuzzy output vectors by Maximum/Minimum Composition[6], and then defuzzification of the fuzzy output vectors to crisp outputs is achieved by the COG (Center Of Gravity)[2] algorithm. Figure 10 shows the basic layout of the fuzzy engine.

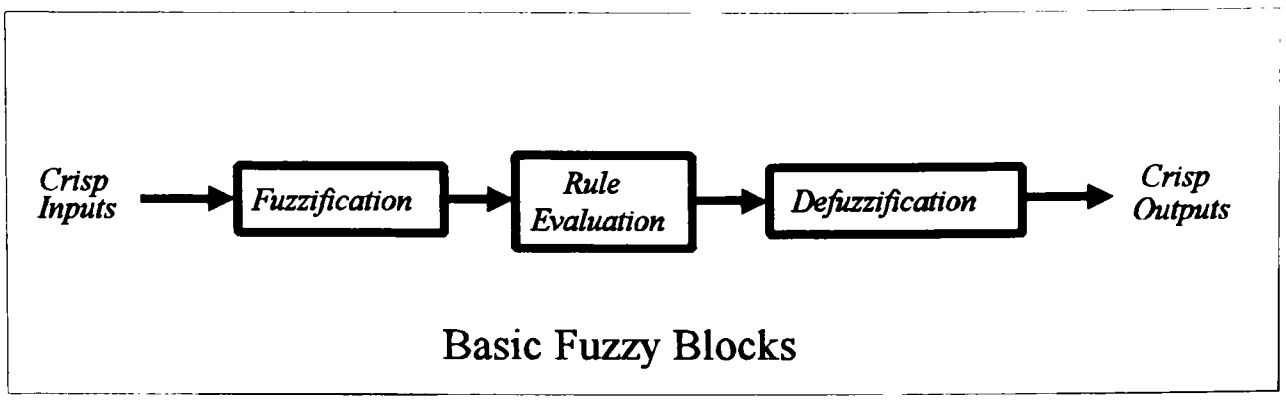


Figure 9. Fuzzy Engine Block Diagram

In Figure 10, the knowledge base section of the fuzzy engine is shown. Fuzzy logic allows a system expert to program or train the knowledge base for the fuzzy control system. This expert can be anyone or anything (consultant, database, neural network, etc.) that knows how the system's output should react to changes at its input. The knowledge base contains the unique description of a fuzzy engine. This description varies from one fuzzy controller to the next. It directly reflects the fuzzy system specifications as entered into FUDGE by the application expert or the system designer. Each of these specifications is represented by a constant data element. Therefore, these constants define the operation of each unique implementation of the fuzzy engine. In short, each fuzzy engine knowledge base contains a series of constant data structures that define the operation of the system. Thus, the design engineer only needs the design parameters for the knowledge base to calculate the memory and processing speed requirements for the control system.

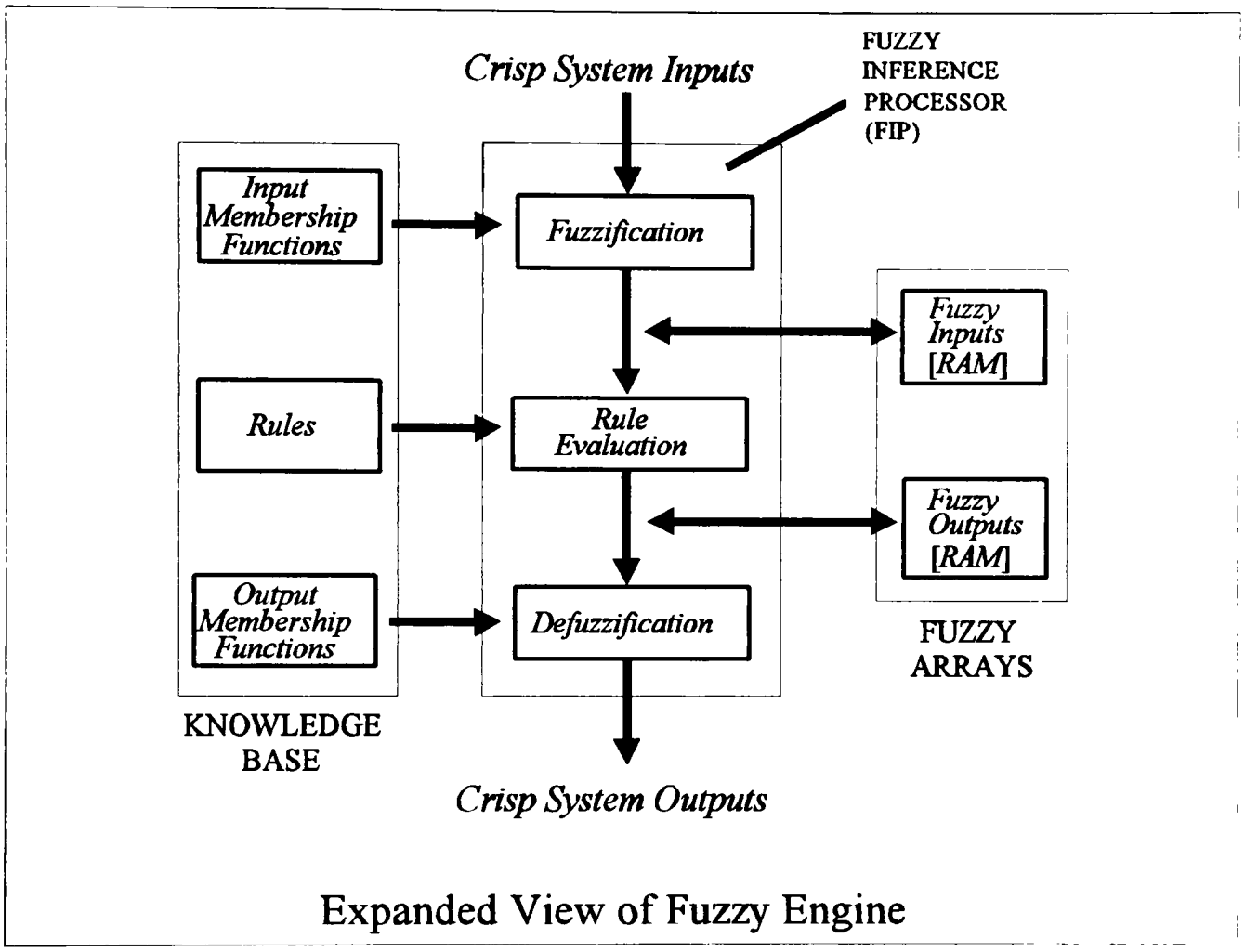


Figure 10. Fuzzy Engine Component Structure

CHAPTER III

RESEARCH METHODOLOGY

This section describes the methodologies used to develop the research addressed in this thesis. This research evaluates and predicts the hardware requirements for fuzzy logic control systems implemented with Motorola's FUzzy Design GEnerator (FUDGE) and expands the high level programming languages supported by FUDGE. This includes the required memory and processing power requirements for controller designs developed in FUDGE, as well as the inclusion of support for the C++ (object-oriented) programming language.

The research can be broken up into two major portions. The first portion consists of developing hardware models (memory allocation and processing power, respectively) for fuzzy logic control systems implemented with the FUDGE software. These models can be used by any engineer to predict the memory and processing power requirements needed to implement a proposed design. These models cover fuzzy systems implemented with Motorola's 68HC05 assembly language or systems implemented with ANSI C and C++ code. They will be especially beneficial to any designer trying to implement fuzzy logic control systems into inexpensive microcontroller applications.

The second portion relates to increasing the number of high level languages that are supported by the FUDGE tool. Since FUDGE is both a design and implementation tool, it will create the output code needed to implement a fuzzy system into some form of microprocessor. This microprocessor may be an Intel Pentium processor or one of Motorola's small self-contained microcontrollers.

The current version of FUDGE (Version 1.02) supports several of Motorola's assembly languages, as well as the ANSI C language. However, it does not produce object-oriented (C++) source code for implementing fuzzy control systems. So this thesis provides a translation program to convert the fuzzy engine (C source code) created by the FUDGE program into a functionally equivalent, object-oriented, fuzzy engine

(C++ code). This C++ code (a "fuzzy" class) allows a design engineer to implement a fuzzy engine in the popular C++ language.

Implementation of Research Methodology

Following is a description of the research methodologies used to develop the hardware models and the C++ object for the FUDGE development environment. This includes the memory and processing power models for fuzzy engines developed with FUDGE. It also covers the additional C++, object oriented translator developed to enhance the fuzzy engine high level language implementations.

Fuzzy Engine Hardware Models

Design engineers commonly need to make estimates about system hardware requirements (such as memory size and processing power) as soon as possible in the design process. With the formulas from this thesis, a designer will be able to make accurate predictions about the hardware requirements for a specific fuzzy control implementation. These projections are developed from the design parameters for the fuzzy engine. With the specification of these parameters, a designer can calculate the hardware requirements of the control system implementation.

With this information, a designer can play with the "what ifs" of their design. For example, they can determine the physical consequences of doubling the number of rules in the rule base or adding an addition input or membership function to the system, thus allowing the design engineer to make important design decisions early in the design process, decisions like trade-offs between application "extras" and the hard physical limitations of the design, which allows the designer to know the exact physical implications of adding more features to a control system. This helps prevents added features from causing problems later in the design process. Problems like getting to the prototyping stage and unexpectedly having the system run out of memory or processing power.

Fuzzy Logic Design Parameters:

- Number of desired crisp inputs in the control system.
- Number and shape of the fuzzy input member functions for each crisp input.
- The approximate number of rules in the fuzzy rule base.
- Number and shape of the fuzzy output member functions for each crisp output.
- Number of crisp outputs in the control system.

Figure 11. Fuzzy Logic Design Parameters

C++ High Level Language Support

This section describes the additional high level language support for the FUDGE environment. Currently, FUDGE only creates ANSI C compatible source code for high level applications. While C is undoubtedly an accepted and widely used high level language, the source code produced by this version (v1.02) of FUDGE is not exactly user friendly. The C code currently requires some hand manipulations before it can be used in the main program. Also, its structure and coding style are not immediately evident because all of the data members are not present or clearly labeled. For instance, the crisp inputs array for the fuzzy engine is not included in the C code implementation and must be added by the user. This C source code also requires that all shared data elements be declared as globals to the entire program. Thus, the programmer must then be careful not to accidentally violate any of the fuzzy data structures, this places an unnecessary burden on the fuzzy system programmer.

Object-oriented programming languages offer several benefits to the programmer, two of which include data encapsulation (protection) of the objects data and an easier

programmer interface to the object. Object-oriented languages also relate software design to natural real-world objects, and fuzzy logic relates natural real-world descriptions to control system designs. This illustrates how these two design methodologies complement each other and the designer's symbolic thought processes so well, which all boils down to the designer being able to quickly and easily develop robust control system applications.

The translator creates a "fuzzy" class structure for the fuzzy controller application. It also eliminates the need for global variables. Thus, this fuzzy class provides the data protection for the fuzzy engine's variables. But, most importantly, it provides an easy interface between the system designer and the fuzzy engine and enhances the natural symbolic interface between the designer and the design.

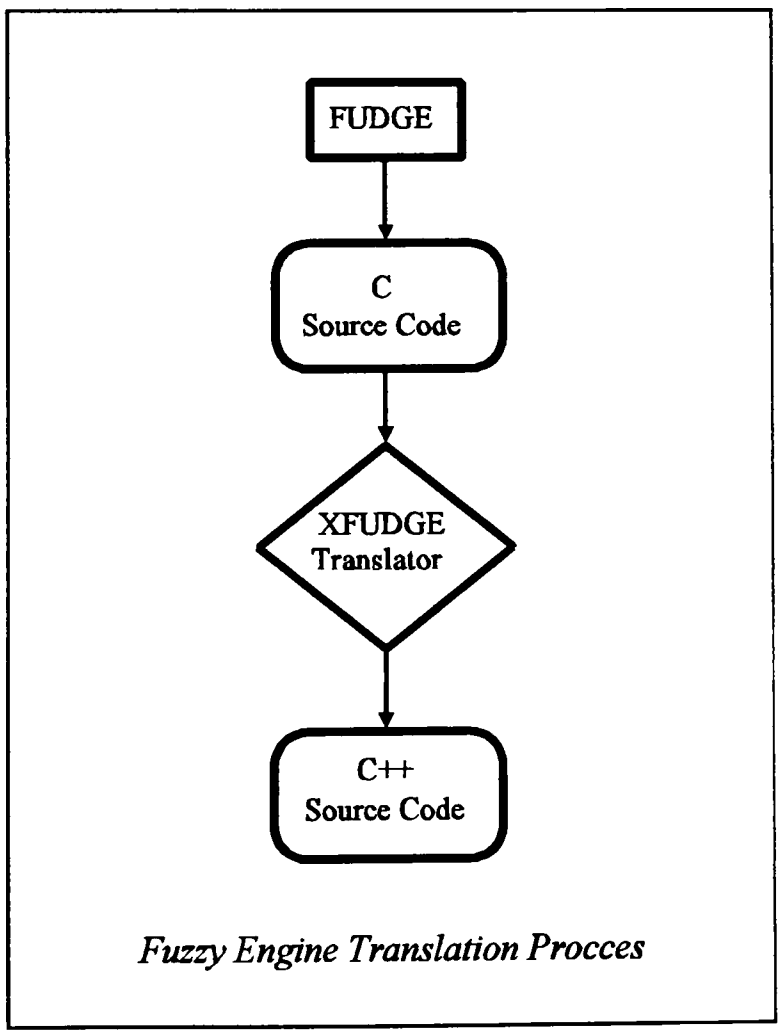


Figure 12.
XFUDGE Translator Block Diagram

Description of Examples

The hardware models and C++ translator developed in this thesis were developed by evaluating and experimenting with the various forms of fuzzy engines produced by the FUDGE development tool. These various forms include the low level assembly language version of the 68HC05 microcontroller and the high level C language version of several fuzzy engine implementations. These fuzzy engine implementations reflect three classic fuzzy logic applications. The specific application examples include: a washing machine controller [7], a traffic light controller [4] and a truck backing controller [2], all three of which were developed with the FUDGE environment. Then each individual output source code was evaluated to determine its structure. Once the fuzzy engine structure was determined, the memory and processing power models were developed, followed by the C++ fuzzy class object. These models and class were tested to insure that they represented an accurate model of the FUDGE fuzzy engine.

This section describes the research methodologies used to evaluate and specify the hardware requirements for implementing fuzzy engines created in the FUDGE development environment. FUDGE is both a design tool and a system evaluation tool. A designer can create, model, test and modify a system's parameters in the FUDGE environment. Afterwards, the designer can use FUDGE to create a software implementation of the fuzzy control engine. This engine is constructed by the FUDGE environment and is either implemented as a ANSI C language file (a high level language) or one of Motorola's assembly language files (a low level language).

Washing Machine Example

The washing machine example represents the first of the classic fuzzy logic control problems. It illustrates a design problem that is extremely difficult to solve with traditional forms of control implementation. Yet, can be easily solved by utilizing the fuzzy logic design method.

The object of this control problem is to implement a controller for a clothes washing machine. This controller should automatically determine the length of the wash time based on the amount (or degree) of dirtiness and the type of dirt on the clothes. The controller requires two inputs (Degree of Dirtiness and Type of Dirt) and one output (the length of the Wash cycle.), which are illustrated in Figure 13.

The degree of dirtiness can vary from some lightly soiled clothes to clothes that are heavily coated with dirt. The length of the wash cycle is determined by the amount and type of dirt present in the wash load. Obviously, the wash cycle will be short for lightly soiled loads and longer for heavily soiled loads. However, the type of dirt on the clothes can greatly effect the wash duration. Greasy dirt on clothes requires a great deal more effort to remove than does non-greasy dirt. So, as the amount of greasy dirt increases, the washer must wash the clothes longer to get them clean. Thus, the amount (dirtiness) and type of dirt are used as crisp inputs to the system and the length of the wash time becomes the system crisp output.

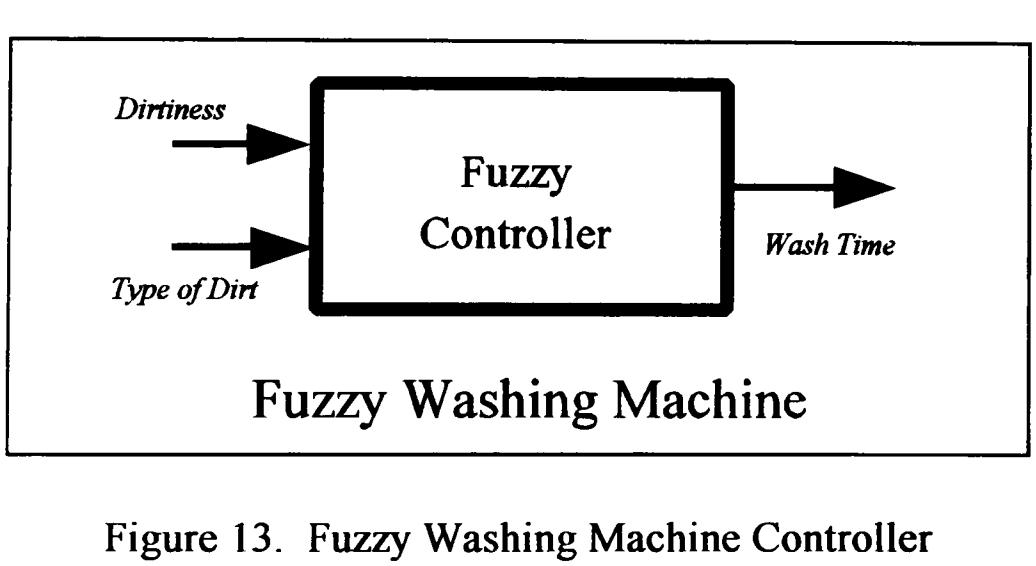
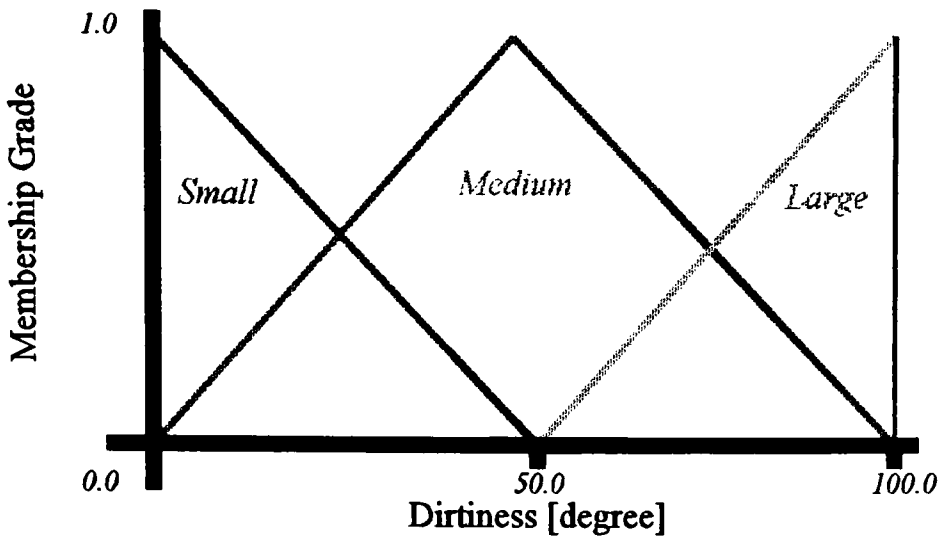
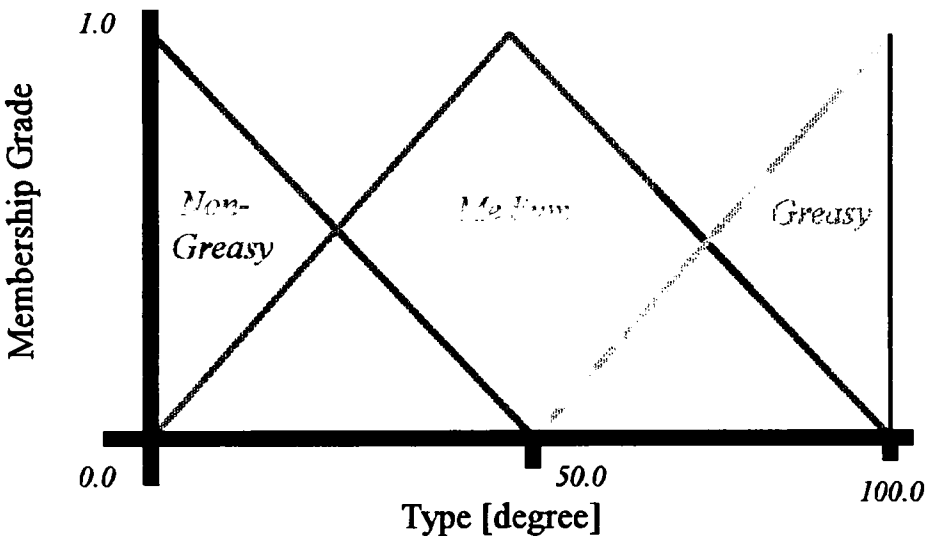


Figure 13. Fuzzy Washing Machine Controller

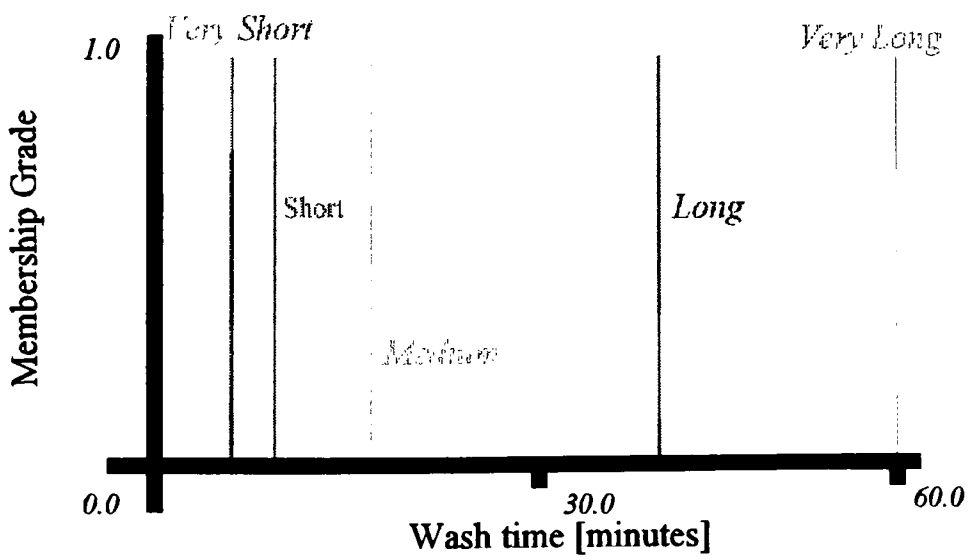
For this fuzzy system, the two crisp inputs receive numerical data from sensors. This input crisp data is a number between 0 and 100. Which represents the percent scale for Dirtiness (0 = light; 100 = Heavy) and of the Type of Dirt (0 = non-greasy; 100 = greasy). The input membership functions for Dirtiness are divided into the three input functions. They are Small, Medium and Large. The input functions for Type of Dirt are also divided into three input membership functions. They are NonGreasy, Medium and Greasy. The crisp output for wash time is scaled in minutes and will be a value from 0 to 60 minutes. It contains five output membership singletons. They are Very Short Time, Short Time, Medium Time, Long Time and Very Long Time. Figure 14 shows the washing machine functions. The Washing Machine example contains nine rules listed in Figure 15.



Input Membership Function for Dirtiness



Input Membership Function for Type of Dirt



Output Membership Function for Wash Time

Figure 14. Washing Machine Membership Functions

Washing Machine Rules

Rule #1
IF Dirtiness IS Large AND TypeOfDirt IS Greasy
THEN WashTime IS VeryLongTime

Rule #2
IF Dirtiness IS Medium AND TypeOfDirt IS Greasy
THEN WashTime IS LongTime

Rule #3
IF Dirtiness IS Small AND TypeOfDirt IS Greasy
THEN WashTime IS LongTime

Rule #4
IF Dirtiness IS Large AND TypeOfDirt IS Medium
THEN WashTime IS LongTime

Rule #5
IF Dirtiness IS Medium AND TypeOfDirt IS Medium
THEN WashTime IS MediumTime

Rule #6
IF Dirtiness IS Small AND TypeOfDirt IS Medium
THEN WashTime IS MediumTime

Rule #7
IF Dirtiness IS Large AND TypeOfDirt IS NonGreasy
THEN WashTime IS MediumTime

Rule #8
IF Dirtiness IS Medium AND TypeOfDirt IS NonGreasy
THEN WashTime IS ShortTime

Rule #9
IF Dirtiness IS Small AND TypeOfDirt IS NonGreasy
THEN WashTime IS VeryShortTime

Figure 15. Washing Machine Rules

Traffic Light Example

The traffic light controller is the next classic example of a fuzzy logic controller. This controller is designed to monitor the traffic flow through a busy intersection. Its purpose is to adjust the red and green lights to maximize the flow of traffic through the intersection. It monitors the number of cars waiting at red lights and the flow rate of cars traveling through the green lights. These rates are then compared by the fuzzy engine to determine if the states of the red and green lights need to change. This fuzzy engine allows the traffic light controller to compensate for varying traffic patters. It always allows the major streets, with the heaviest traffic patterns, to have the priority green light state. Yet, it keeps the cars at red lights from having to wait for long periods of time before they get a green light. The controller always adjusts the length of time for the red light/green light cycle based on the traffic flow and the length of time that cars have been waiting. As shown by Kelsey and Bisset [4], this fuzzy controller can provide significant improvements over traditional non-fuzzy controllers.

This fuzzy engine requires four fuzzy functions that describe the traffic densities for the red and green lights. Specifically, there are three input functions and one output function. The three input functions relate the number of cars waiting at red lights, the rate of cars traveling through green lights, and the amount of time each light has been in its current state. They are respectively named: RedLights, GreenLights and Time. The output function determines the need to change the current state of the lights at the intersection. It is named Degree of Change.

The input membership functions for red and green lights each have four membership states as shown in Figure 16. They represent the number of cars related to that lights state. The four states are: Zero, Low, Medium and High. The input function for time contains three membership states and each state measures the time in seconds. The states are: Short, Medium and Long. The output membership function for the Degree of Change contains five membership states. These states represent the probability that the light state is about to change. These states are: No, Probably No, Maybe, Probably Yes and Yes. The rule base in Figure 17 describes the relationship between the fuzzy inputs and the fuzzy output of the system.

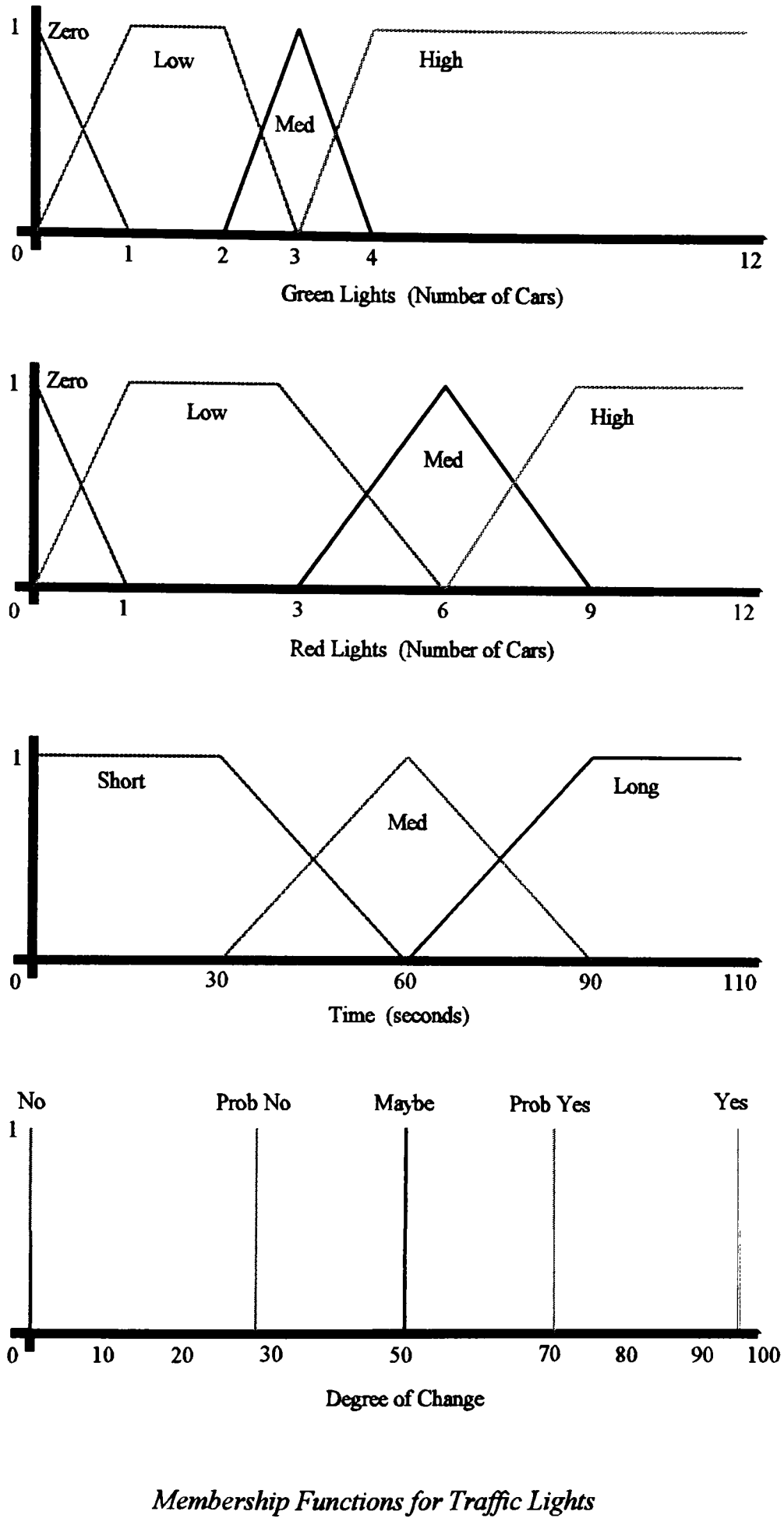


Figure 16. Traffic Light Membership Functions

Traffic Light Rules:

Rule #1
IF GreenLight IS Zero AND RedLight IS Zero THEN Change IS No

Rule #2
IF GreenLight IS Zero AND RedLight IS Low THEN Change IS Yes

Rule #3
IF GreenLight IS Zero AND RedLight IS Medium THEN Change IS Yes

Rule #4
IF GreenLight IS Zero AND RedLight IS High THEN Change IS Yes

Rule #5
IF RedLight IS Zero THEN Change IS No

Rule #6
IF GreenLight IS Low AND RedLight IS Low THEN Change IS No

Rule #7
IF GreenLight IS Medium AND RedLight IS Medium
THEN Change IS No

Rule #8
IF GreenLight IS High AND RedLight IS High THEN Change IS No

Rule #9
IF GreenLight IS Low AND RedLight IS Medium
AND CycleTime IS Short THEN Change IS Maybe

Rule #10
IF GreenLight IS Low AND RedLight IS Medium
AND CycleTime IS Medium THEN Change IS ProbYes

Rule #11
IF GreenLight IS Low AND RedLight IS Medium
AND CycleTime IS Long THEN Change IS Yes

Rule #12
IF GreenLight IS Low AND RedLight IS High
AND CycleTime IS Short THEN Change IS ProbNot

Rule #13
IF GreenLight IS Low AND RedLight IS High
AND CycleTime IS Medium THEN Change IS Maybe

Figure 17. Traffic Light Rules

Traffic Light Rules Continued

Rule #14
IF GreenLight IS Low AND RedLight IS High
AND CycleTime IS Long THEN Change IS ProbYes

Rule #15
IF GreenLight IS Medium AND RedLight IS Low
AND CycleTime IS Short THEN Change IS ProbNot

Rule #16
IF GreenLight IS Medium AND RedLight IS Low
AND CycleTime IS Medium THEN Change IS ProbNot

Rule #17
IF GreenLight IS Medium AND RedLight IS Low
AND CycleTime IS Long THEN Change IS Maybe

Rule #18
IF GreenLight IS Medium AND RedLight IS High
AND CycleTime IS Short THEN Change IS Maybe

Rule #19
IF GreenLight IS Medium AND RedLight IS High
AND CycleTime IS Medium THEN Change IS ProbYes

Rule #20
IF GreenLight IS Medium AND RedLight IS High
AND CycleTime IS Long THEN Change IS Yes

Rule #21
IF GreenLight IS High AND RedLight IS Low
AND CycleTime IS Short THEN Change IS Maybe

Figure 17. Continued

Traffic Light Rules Continues

Rule #22
IF GreenLight IS High AND RedLight IS Low
AND CycleTime IS Medium THEN Change IS ProbYes

Rule #23
IF GreenLight IS High AND RedLight IS Low
AND CycleTime IS Long THEN Change IS Yes

Rule #24
IF GreenLight IS High AND RedLight IS Medium
AND CycleTime IS Short THEN Change IS ProbNot

Rule #25
IF GreenLight IS High AND RedLight IS Medium
AND CycleTime IS Medium THEN Change IS ProbNot

Rule #26
IF GreenLight IS High AND RedLight IS Medium
AND CycleTime IS Long THEN Change IS Maybe

Figure 17. Continued

Truck Backing Example

The last of the classic fuzzy logic examples is the Truck Backer-Upper as described by Kosko [2]. The object of the system is to back the truck up perpendicular to the loading dock. This fuzzy system controls the truck that is being backed up to a loading dock. The fuzzy engine determines the position (theta) of the truck wheels with respect to the position (phi) of the truck body to the loading dock. This system is shown in Figure 18.

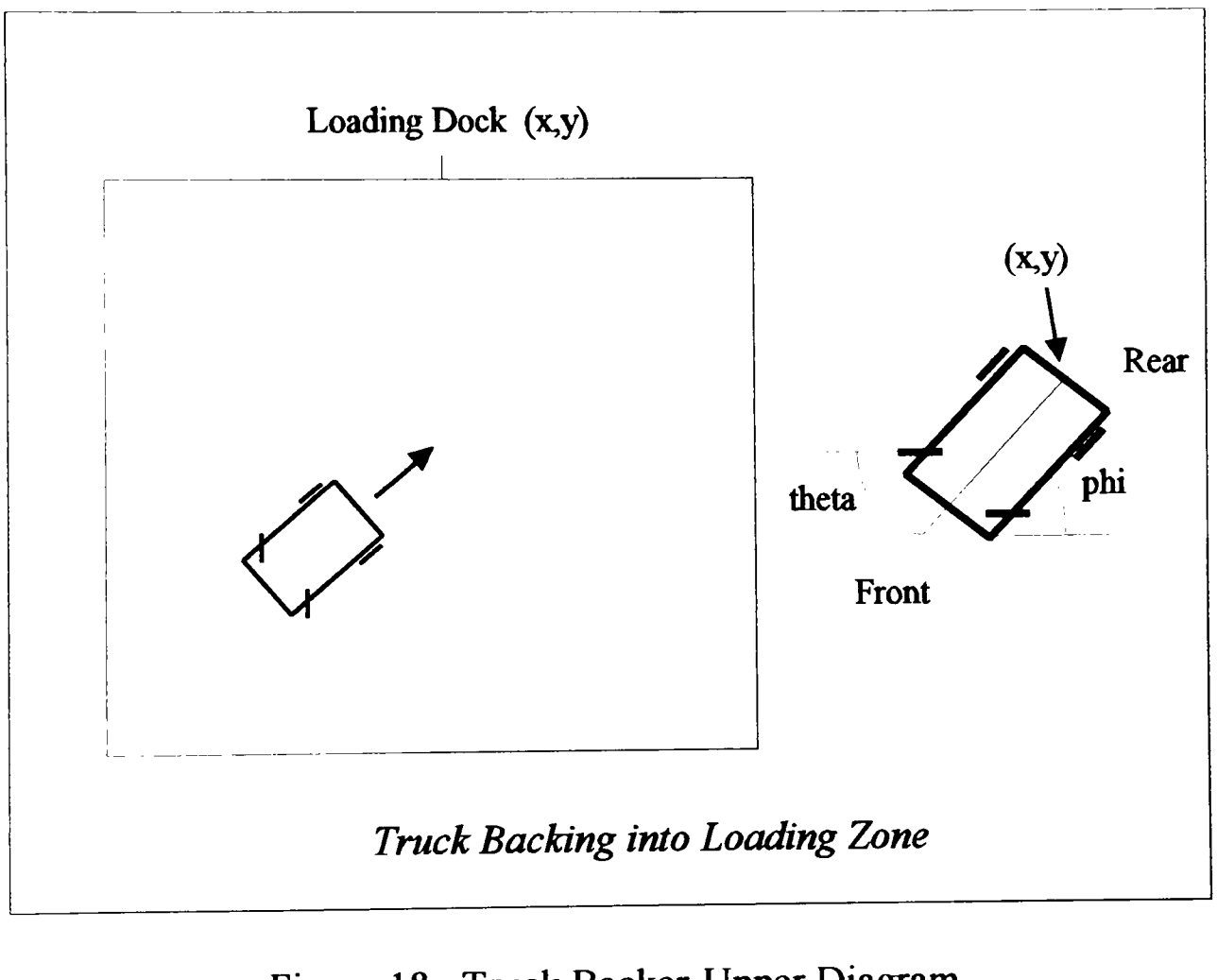
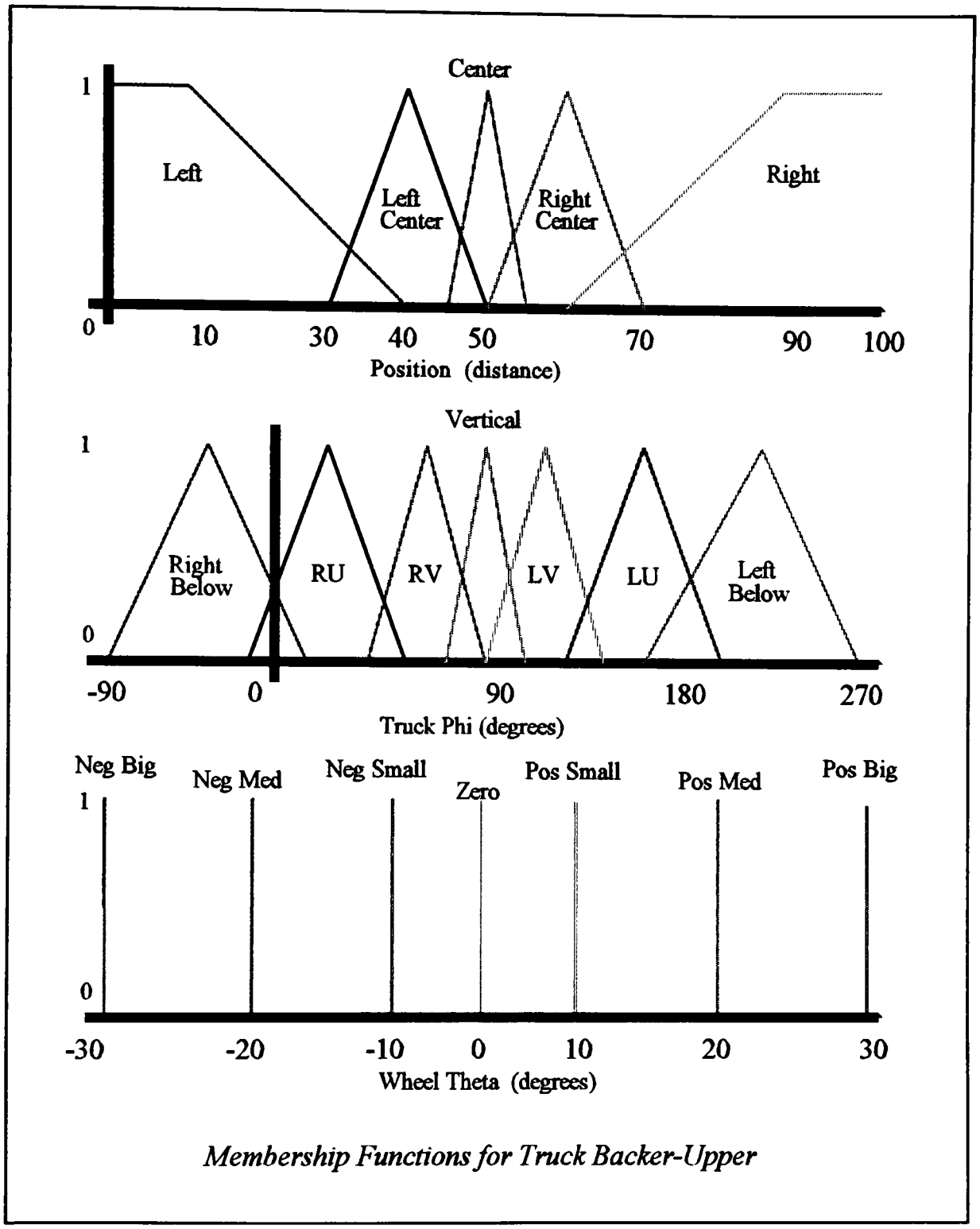


Figure 18. Truck Backer-Upper Diagram

The truck is originally placed at some random place and positioned in front of the loading dock. The fuzzy controller must then back the truck up to the loading dock. The controller must evaluate the current position and direction of the truck. Then it must determine the best positioning of the tires to steer the truck towards the dock.

The fuzzy controller contains three fuzzy functions that describe the system. Specifically, it contains two inputs and one output function. The two input functions relate the truck's current position to the fuzzy engine. They are: Position (x, y location) and Phi (truck body direction). The single output function determines the appropriate truck wheel direction (theta). The output singletons are: Negative Big, Negative Medium, Negative Small, Zero, Positive Small, Positive Medium and Positive Big. These fuzzy input and output functions are shown in Figure 19. The fuzzy system is described by the rules in Figure 20.



Membership Functions for Truck Backer-Upper

Figure 19. Truck Backer-Upper Membership Functions

Truck Rules:

Rule #1
IF Phi IS RightBelow AND Position IS Left
THEN Theta IS PosSmall

Rule #2
IF Phi IS RightUpper AND Position IS Left
THEN Theta IS NegSmall

Rule #3
IF Phi IS RightVertical AND Position IS Left
THEN Theta IS NegMedium

Rule #4
IF Phi IS Vertical AND Position IS Left
THEN Theta IS NegMedium

Rule #5
IF Phi IS LeftVertical AND Position IS Left
AND Position IS Left THEN Theta IS NegBig

Rule #6
IF Phi IS LeftUpper AND Position IS Left
THEN Theta IS NegBig

Rule #7
IF Phi IS LeftBelow AND Position IS Left
THEN Theta IS NegBig

Rule #8
IF Phi IS RightBelow AND Position IS LeftCenter
THEN Theta IS PosMedium

Rule #9
IF Phi IS RightUpper AND Position IS LeftCenter
THEN Theta IS PosSmall

Rule #10
IF Phi IS RightVertical AND Position IS LeftCenter
THEN Theta IS NegSmall

Figure 20. Truck Backer-Upper Rules

Truck Rules Continued:

Rule #11
IF Phi IS Vertical AND Position IS LeftCenter
THEN Theta IS NegMedium

Rule #12
IF Phi IS LeftVertical AND Position IS LeftCenter
THEN Theta IS NegMedium

Rule #13
IF Phi IS LeftUpper AND Position IS LeftCenter
THEN Theta IS NegBig

Rule #14
IF Phi IS LeftBelow AND Position IS LeftCenter
THEN Theta IS NegBig

Rule #15
IF Phi IS RightBelow AND Position IS Center
THEN Theta IS PosMedium

Rule #16
IF Phi IS RightUpper AND Position IS Center
THEN Theta IS PosMedium

Rule #17
IF Phi IS RightVertical AND Position IS Center
THEN Theta IS PosSmall

Rule #18
IF Phi IS Vertical AND Position IS Center
THEN Theta IS Zero

Rule #19
IF Phi IS LeftVertical AND Position IS Center
THEN Theta IS NegSmall

Rule #20
IF Phi IS LeftUpper AND Position IS Center
THEN Theta IS NegMedium

Figure 20. Continued

Truck Rules Continued:

Rule #21
IF Phi IS LeftBelow AND Position IS Center
THEN Theta IS NegMedium

Rule #22
IF Phi IS RightBelow AND Position IS RightCenter
THEN Theta IS PosBig

Rule #23
IF Phi IS RightUpper AND Position IS RightCenter
THEN Theta IS PosBig

Rule #24
IF Phi IS RightVertical AND Position IS RightCenter
THEN Theta IS PosMedium

Rule #25
IF Phi IS Vertical AND Position IS RightCenter
THEN Theta IS PosMedium

Rule #26
IF Phi IS LeftVertical AND Position IS RightCenter
THEN Theta IS PosSmall

Rule #27
IF Phi IS LeftUpper AND Position IS RightCenter
THEN Theta IS NegSmall

Rule #28
IF Phi IS LeftBelow AND Position IS RightCenter
THEN Theta IS NegMedium

Rule #29
IF Phi IS RightBelow AND Position IS Right
THEN Theta IS PosBig

Rule #30
IF Phi IS RightUpper AND Position IS Right
THEN Theta IS PosBig

Figure 20. Continued

Truck Rules Continued:

Rule #31
IF Phi IS RightVertical AND Position IS Right
THEN Theta IS PosBig

Rule #32
IF Phi IS Vertical AND Position IS Right
THEN Theta IS PosMedium

Rule #33
IF Phi IS LeftVertical AND Position IS Right
THEN Theta IS PosMedium

Rule #34
IF Phi IS LeftUpper AND Position IS Right
THEN Theta IS PosSmall

Rule #35
IF Phi IS LeftBelow AND Position IS Right
THEN Theta IS NegSmall

Figure 20. Continued

CHAPTER IV
RESEARCH RESULTS

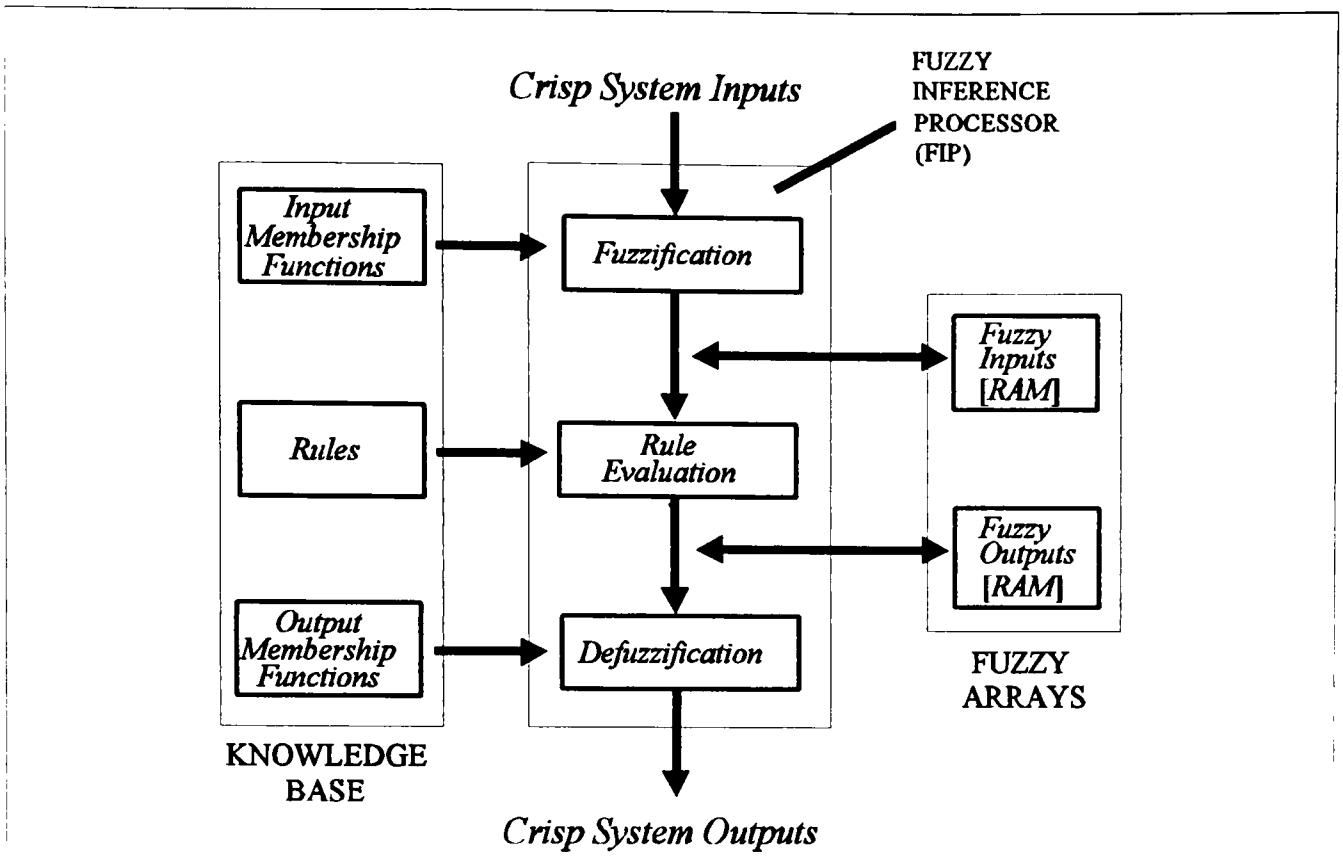
This chapter describes the hardware models developed in this thesis. This research includes the development of ANSI C, C++ and 68HC05 hardware models, plus a C++ translation program for Motorola's FUzzy Design GEnerator (FUDGE). These hardware models describe the memory and processing power requirements for fuzzy logic controllers developed with FUDGE, and the translation program expands the high level programming languages supported by FUDGE to include object-oriented C++.

Fuzzy Engine Structure

The common link between hardware models and the C++ translation program is the output source code produced by the FUDGE tool. This output source code represents the software module that will be executed by a microprocessor in some control system application. In a fuzzy logic controller this module is referred to the "fuzzy engine."

The fuzzy engine is the mechanism that maps system input states to the appropriate output states in a fuzzy logic controller. This fuzzy engine is represented by a series of software functions called procedures. These procedures perform the three basic operations of the fuzzy engine. These three fundamental operations are: fuzzification, rule evaluation and defuzzification. The fuzzy engine produced by the FUDGE and XFUDGE programs is constructed around these same three basic procedures.

All of the FUDGE fuzzy engine implementations contain the same program structure shown in Figure 21. So for simplicity, the following discussion will apply to all three code versions (assembly, C and C++) of the fuzzy engine. Otherwise, any specific structural differences between the code versions will be pointed out as the need arises. As shown in the figure, the fuzzy engine contains three primary sections: the Knowledge Base, the Fuzzy Inference Processor and the Fuzzy Arrays.



Structural View of Fuzzy Engine

Figure 21. Fuzzy Engine Structural Block Diagram

Knowledge Base

The Knowledge Base contains the unique description of a fuzzy engine. This description always varies from one fuzzy controller to the next. This is because it directly reflects the fuzzy system specifications as entered into FUDGE by the application expert. Each of these specifications represents a series of constant data structures. These constant structures define each fuzzy input with their related membership functions. The Rule Base and each fuzzy output with its related membership functions are also represented by these constants. In short, each fuzzy engine contains a series of constant data structures that define the operation of the fuzzy system, and the values in these data structures vary from engine to engine.

Fuzzy Inference Processor

The Fuzzy Inference Processor (FIP) contains the software procedures that calculate the crisp input to crisp output mappings for the fuzzy engine. These procedures implement the three processes of the fuzzy engine shown in the figure. First, they Fuzzify the Crisp System Inputs of the system and place the results in the Fuzzy Inputs Array. Next, they Infer the fuzzy inputs to the Fuzzy Outputs Array by evaluating the rules in the Fuzzy Rule Base. Finally, they convert the values in the Fuzzy Outputs Array into Crisp System Output values.

Fuzzy Arrays

The Fuzzy Arrays simply act as a temporary scratch pad for the Fuzzy Inference Processor. The Fuzzy Input Array contains the vector results that relate the Fuzzy Input Membership Functions to the Crisp System Inputs during the Fuzzification process, while the Fuzzy Outputs Array contains the vector results of the Min-Max correlation performed between the Fuzzy Inputs Array and the Fuzzy Outputs Array during the Rule Evaluation procedure. Then during the Defuzzification procedure, the fuzzy output values are combined with the Fuzzy Output Membership Functions to determine the appropriate Crisp System Output value. This is achieved by the Center of Gravity (COG) method of conveying fuzzy outputs to crisp outputs [2].

Hardware Requirement Models for the Fuzzy Engine

The hardware models contained here are divided into two separate but related categories. Both categories deal with the modeling and prediction of system hardware requirements for fuzzy logic systems designed with the FUDGE tool. However, the first model describes the control system memory requirements, and the second describes the processor execution delays for the control system.

The first model describes the amount of computer memory needed to implement a fuzzy engine. This model can be used to determine memory requirements for any of the three programming languages supported by FUDGE. This includes engines

implemented with assembly, C or C++ languages. These memory models are helpful in determining the amount of ROM and/or RAM memory for a fuzzy engine implementation. Secondly, the processing power models describe the expected processor delays (throughput) for a fuzzy engine implementation. The processor throughput is directly proportional to the execution delay of the processor. So, execution delay defined as the amount of time it takes for a system to produce an output response to a change at the system's input state.

Both types of models are based on the fuzzy logic system design specifications listed in Figure 21. These specifications list the maximum number of system components that a fuzzy design may incorporate and are limited by the FUDGE development software. The amount of required memory or processing power may increase or decrease if there is any change in a fuzzy system's specifications. However, please note that all hardware models are limited by the maximum values listed in Figure 22. Any deviation from these values will be addressed as the need arises. The specifications listed as "Varies" are implementation specific, and their value will be noted when appropriate. However, in all cases the maximum values will always be greater than or equal to 128.

Fuzzy System Specification Variables

Inputs:	*Quantity*
▸ Maximum Number of Crisp Inputs	8
▸ Maximum Number of Input Functions per Input	8
▸ Shape of Input Functions	Trapezoid
Outputs:	
▸ Maximum Number of Crisp Outputs	4
▸ Maximum Number of Output Functions per Output	8
▸ Shape of Output Functions	Singleton
Rules:	
▸ Maximum Number of Rules	Varies
▸ Maximum Number of Antecedents per Rule	Varies
▸ Maximum Number of Consequences per Rule	Varies

Figure 22. Fuzzy System Specifications

Fuzzy Engine Models

The fuzzy engine is simply a series of software procedures for calculating crisp output values for changes at the system's crisp inputs. It performs three basic operations, and each operation is implemented as a program procedure. Thus, the three major fuzzy engine program procedures are: fuzzification, rule evaluation and defuzzification. Remember that these procedures may be written in any of three programming languages. If a fuzzy engine is created by the FUDGE software, then it will be written in either assembly or the C programming languages. If the fuzzy engine was translated (from C source code) by the XFUDGE software, it will be written in the C++ language. Regardless of the language, the fuzzy engine requires RAM and/or ROM memory during its execution. The following sections contain the equations used to model the memory requirements for unique fuzzy engine implementations. Table 1 lists the implied number of bytes for each data type used in the memory model equations.

Table 1. Assumed Number of Bytes for Data Types

Data Type	Number of Bytes
Character	1
Integer	2
Float	4

Memory Models

68HC05 Fuzzy Engine

The fuzzy engine for the Motorola 68HC05 microcontroller is written in assembly language. This assembly code is enacted by a combination of software procedures and knowledge base definitions. So for simplicity, the 68HC05 fuzzy engine is divided up

into two separate sections. The first section is called the Fuzzy Inference Processor (FIP) and is shown in Figure 21. This section includes the common source code that implements the FIP procedures. The second section is referred to as the Knowledge Base and is also shown in the figure. The Knowledge Base section contains the data structures that create a unique fuzzy engine application. Note that the Fuzzy Arrays shown in the figure are directly defined by the Knowledge Base. So, they are also included with the Knowledge Base section.

The current version for the 68HC05 FIP is ver. 1.0 and is located in the "eng10.a" file. It is important to note that the FIP procedures do not vary between fuzzy engine implementations. This allows each specialized fuzzy engine to execute the same assembly code procedures for the FIP. Thus, the fuzzy engine is made unique by the data structures found in its Knowledge Base. These data structures form the ROM and RAM elements that define the specific input/output relationships of the engine. So, each fuzzy engine must have its own unique knowledge base to describe its operation. This knowledge base is created by the FUDGE tool and is placed in a output file with the "asm" extension. The exact knowledge base filename is determined by the system designer, but the default name is "fuzzy.asm". Therefore, both the FIP (eng10.a) and the knowledge base (fuzzy.asm) files are complied together to create the complete fuzzy engine implementation. The combined files for the three examples described in this thesis are shown in Table 2.

Table 2. Project Files for Thesis Examples

Example Name	Project Assembly Files
Washing Machine	eng10.a + wash.asm
Traffic Light	eng10.a + traffic.asm
Truck Backer-Upper	eng10.a + truck.asm

In order for the fuzzy engine to map crisp inputs to crisp outputs, it must perform all three steps of the Fuzzy Inference Processor. These steps are: fuzzification, rule

evaluation and defuzzification. The FIP does this by executing the processes in the "eng10.a" file. These processes interrogate the data structures in the "fuzzy.asm" knowledge base file. From these data structures the FIP determines the characteristics of the fuzzy engine. Meanwhile, it uses the Fuzzy Arrays to store the values of the fuzzy input or output variables.

Memory Requirements for Motorola 68HC05 Microcontrollers. The memory requirements for the fuzzy engine implementation of any version of the 68HC05 microcontroller can be determined by evaluating three memory elements. These elements are shown in the fuzzy engine diagram, Figure 21. They are the Knowledge Base, Fuzzy Inference Processor and the Fuzzy Arrays. The Knowledge Base and Fuzzy Array sizes are strictly determined by the unique design of the fuzzy control system, while the memory required for the Fuzzy Inference Processor is a constant 300 bytes of ROM for any controller implementation.

Both the Knowledge Base and the Fuzzy Array memory size are directly related to the controller design specifications, and they are both described by similar models. These memory equations are broken up into their respective ROM and RAM portions to simplify the memory calculations. The ROM model in equation (1) describes the constant memory required for the Knowledge Base. This is because the knowledge base data structures allocated in this model represent the constant specifications of the system.

This memory model assumes that A_{inputs} represents the number of fuzzy inputs in the system and that each input contains A_{msf} number of membership functions. Plus, the rules are represented by the number of rules (B_{rules}) in the system, each with B_{ants} number of antecedents and B_{cons} number of consequences. The fuzzy outputs are represented by $C_{outputs}$, each with C_{msf} number of output membership functions.

$$ROM\ Requirements = (A_{inputs} * A_{msf} * 6) + B_{rules}(B_{ants} + B_{cons}) + (C_{outputs} * C_{msf}) + 1.$$

Eq. (1) 68HC05 ROM Memory Requirements

The terms in equation (1) are determined by direct evaluation of data structures placed in the output knowledge base file by FUDGE. This knowledge base file contains all the data structures that describe a particular instance of a fuzzy engine. The first term calculates the ROM memory required to describe the input membership functions (A_{msf}) of each fuzzy system input (A_{inputs}). These two terms are multiplied by 6 to account for the six bytes of memory that describe each membership function. The second term calculates the amount of ROM memory required for the rule base of the fuzzy engine. Each rule antecedent and consequence are represented by separate byte of memory. Thus, every rule will require a single byte of memory for each antecedent and consequence. To figure the amount of memory for the rule base, the number of rules in the system (B_{rules}) is multiplied by the addition of the maximum number of rule antecedents (B_{ants}) and consequences (B_{cons}). Then one additional byte is added (the last term) to account for the end of rules marker. The third term represents the location of the singletons for each output membership function. So, each singleton will be represented by a single byte of data. Therefore, to calculate the number of bytes required for the outputs is determined by multiplying the number of membership functions (B_{msf}) times the number of outputs ($B_{outputs}$).

The Fuzzy Arrays are modeled by the RAM equations. These model equations calculate the required amount of memory needed by the Fuzzy Arrays and other temporary variables used by the FIP. These models assume that A_{inputs} is the number of fuzzy inputs in the system, with each input containing A_{msf} fuzzy membership functions. Plus, the fuzzy outputs are represented by the number of $C_{outputs}$, each with C_{msf}

number of output membership functions. Also, note that there are two RAM memory models. These models are mutually exclusive, and the design engineer must decide which model to use with the system. The equation in RAM Model 1 (Eq.(2)) represents the standard variable model for the FIP. This model utilizes the standard (default) version of the FIP code. The second equation in RAM Model 2 (Eq.(3)) represents the shared memory version of the FIP. This version of the FIP utilizes the temporary variables in a union data structure to save the number of bytes required for RAM. Please note that Model 2 is used to save precious RAM memory and that these two models only differ by six bytes.

$$RAM\ Requirements = A_{inputs}(1 + A_{msf}) + C_{outputs}(1 + C_{msf}) + 15.$$

Eq. (2) 68HC05 RAM Memory Requirements (Model 1)

$$RAM\ Requirements = A_{inputs}(1 + A_{msf}) + C_{outputs}(1 + C_{msf}) + 9.$$

Eq. (3) 68HC05 RAM Memory Requirements (Model 2)

In equations (2) and (3), the amount of RAM memory required for a 68HC05 fuzzy engine implementations are described. Both equations (the last terms) allow for the temporary variables needed for these implementations. The amount memory required for temporary variables is constant in both models. The fuzzy vectors are stored in two multidimensional arrays. The input array has one more than the maximum number of input membership functions (columns) times the number of fuzzy inputs (rows). Since all rules are stored in ROM, there is no need for extra RAM for rule evaluation. Finally, the fuzzy output array has one more than the maximum number of output membership functions (columns) times the number of fuzzy outputs (rows).

Memory Requirements for C and C++ Applications

As shown in Figure 21, the fuzzy engine is composed of three basic elements. They are: the Knowledge Base, Fuzzy Inference Processor and Fuzzy Arrays. The Fuzzy Inference Processor (FIP) consists only of the code needed to implement the fuzzy engine procedures. Thus, the memory required by the Inference Processor code does not change from engine to engine. So, these memory allocation models will only describe the memory requirements for the Knowledge Base and the Fuzzy Arrays. Equation (4) calculates the memory required to implement the C source code version of the fuzzy engine. While equation (5) calculates the memory needed to implement the C++ version of the fuzzy engine object.

In the C and C++ equations, A_{inputs} represents the number of crisp inputs in the system, and A_{msf} represents the maximum number of membership functions for any of the fuzzy inputs. B_{rules} is the number of rules in the rule base, while $C_{outputs}$ is the number of crisp outputs and C_{msf} is the maximum number of output membership functions for any of the fuzzy outputs.

$$C\ Memory = 126\,A_{inputs} + 36\,B_{rules} + 126\,C_{outputs} + 646.$$

Eq. (4) C Memory Requirements

$$C{+}{+}\ Memory = 126\,A_{inputs} + 36\,B_{rules} + 126\,C_{outputs} + 390.$$

Eq. (5) C++ Memory Requirements

Processor Power Models

68HC05 Processing Power Requirements

To evaluate a control system input state, the 68HC05 implementation of the fuzzy engine must perform each of the three (fuzzification, rule evaluation and defuzzification) of the Fuzzy Inference Processes. This means that the 68HC05's microprocessor must

execute each of the FIP processes. Of course, it takes a certain amount of time for the processor to execute this code, which results in an execution time delay for each crisp input state evaluated by the fuzzy engine. How much time (execution delay) depends on three factors. These factors are: processor clock speed, the fuzzy engine specifications and the particular input state of the controller.

The processor clock speed is determined by the oscillator connected to the microcontroller. For the 68HC05 series, the oscillator speed is typically less than 4.2 MHz. (However, note that the 68HC05 clock generating circuit always divides the oscillator frequency in half.) So, the minimum individual clock period is 0.476 micro seconds. In short, as the clock frequency decreases, the execution time increases.

The fuzzy engine specifications are determined by the design engineer and limited by Figure 22 and Table 3. Note that the total number of bytes for the 68HC05 fuzzy inputs is 256. This is due to the maximum fuzzy input array size limitation of the 68HC05. Therefore, each input membership function is represented by six bytes of data, and all the input membership functions are stored in the fuzzy input array, which limits the total number of bytes for these functions to be less than or equal to 256 bytes. In short, this means that there can either be a maximum of five fuzzy inputs (A_{input}), each with eight membership functions (A_{msf}) each or eight fuzzy inputs (A_{input}) with a maximum of five membership functions (A_{msf}) each.

As the number of specifications (crisp input, rules or crisp outputs) increase, so does execution delay. The input state (or Crisp Input) of the system can also affect the execution time of the fuzzy engine. Some input states may be easily and quickly evaluated by the FIP. This fact relates to short execution delays for some input states, while other input states may require lengthy evaluations by the FIP, and this relates to longer execution delays.

Table 3. 68HC05 Specification Limitations

Specification	Formula	<=
Total Number of Input Membership Functions	$A_{inputs} * A_{msf} * 6$	256
Total Number of Rule Antecedents and Consequences	$B_{rutes} * B_{ants} * B_{cons}$	256
Total Number of Output Membership Functions	$C_{outputs} * C_{msf}$	128

The exact execution delay of a fuzzy engine can be determined by the evaluation of each for the three steps of the Fuzzy Inference Process. If the specifications of a fuzzy system are known, then the exact number of clock cycles needed to map crisp inputs to crisp outputs with the FIP can be calculated. This is shown by equation (6). Once the exact number of clock cycles have been determined, then it is a trivial problem to calculate the execution delay of the system, as shown by equation (7).

$$Total\ FIP\ Clock\ Cycles = Fuzzification + Antecedents + Consequences + Defuzzification + 20.$$

Eq. (6) Total Fuzzy Engine Execution Cycles

$$Fuzzy\ Engine\ Execution\ Delay = [\ Total\ FIP\ Clock\ Cycles * \left(\frac{1}{(\ Oscillator\ freq.\ /\ 2)}\right)\ ].$$

Eq. (7) Fuzzy Engine Execution Delay Calculation

Fuzzification Cycles. The first step of the Fuzzy Inference Process is the fuzzification of the crisp inputs. This step converts every crisp input value into an equivalent fuzzy input vector. This is done by evaluating the degree of truthfulness of each crisp input. The degree of truthfulness is the relationship of each crisp input value to its corresponding fuzzy input membership functions. The result is a fuzzy input vector in the Fuzzy Input array of the Fuzzy Arrays portion of the FIP model. Refer to the model of the fuzzy engine in
Figure 21.

In the fuzzification process, crisp input values are converted to fuzzy input values. Each fuzzy input is represented by a vector. This vector indicates the amount of truthfulness applied to a fuzzy input by the current crisp input value. The amount of fuzzy input truthfulness is determined by the position of the crisp input value with respect to each of the fuzzy input's membership functions. Thus, the combined truthfulness values of all the input membership functions determines the fuzzy input vector for the crisp input value. To calculate the fuzzy input vector, the FIP applies the crisp input to all the membership functions for the fuzzy input.

The most complex membership function shape in the FUDGE environment is the trapezoid shown in Figure 23. All other membership shapes are just simpler versions of the trapezoid. Thus, the trapezoid represents the most complex shape available to input membership functions. Notice the four vertical lines (A, B, C and D) in the figure. They represent the intersection of the four defining points (P1, P2, P3 and P4) of a trapezoid input membership function. The lines represent the crisp input values that correspond to these defining membership points. In the fuzzification process, these points indicate a change in the amount of truthfulness a crisp input has in relationship to a fuzzy input. Thus, the truthfulness of a crisp input is determined by its relation to these points. This crisp input position also directly relates to the number of clock cycle needed to evaluate the input state. The four points divide the membership function into five separate sections. Thus, the crisp input value will always fall into one of the five sections of the membership function shown in the figure. Table 4 shows the number of clock cycles

required by the fuzzification process for crisp inputs that fall within any of the five sections.

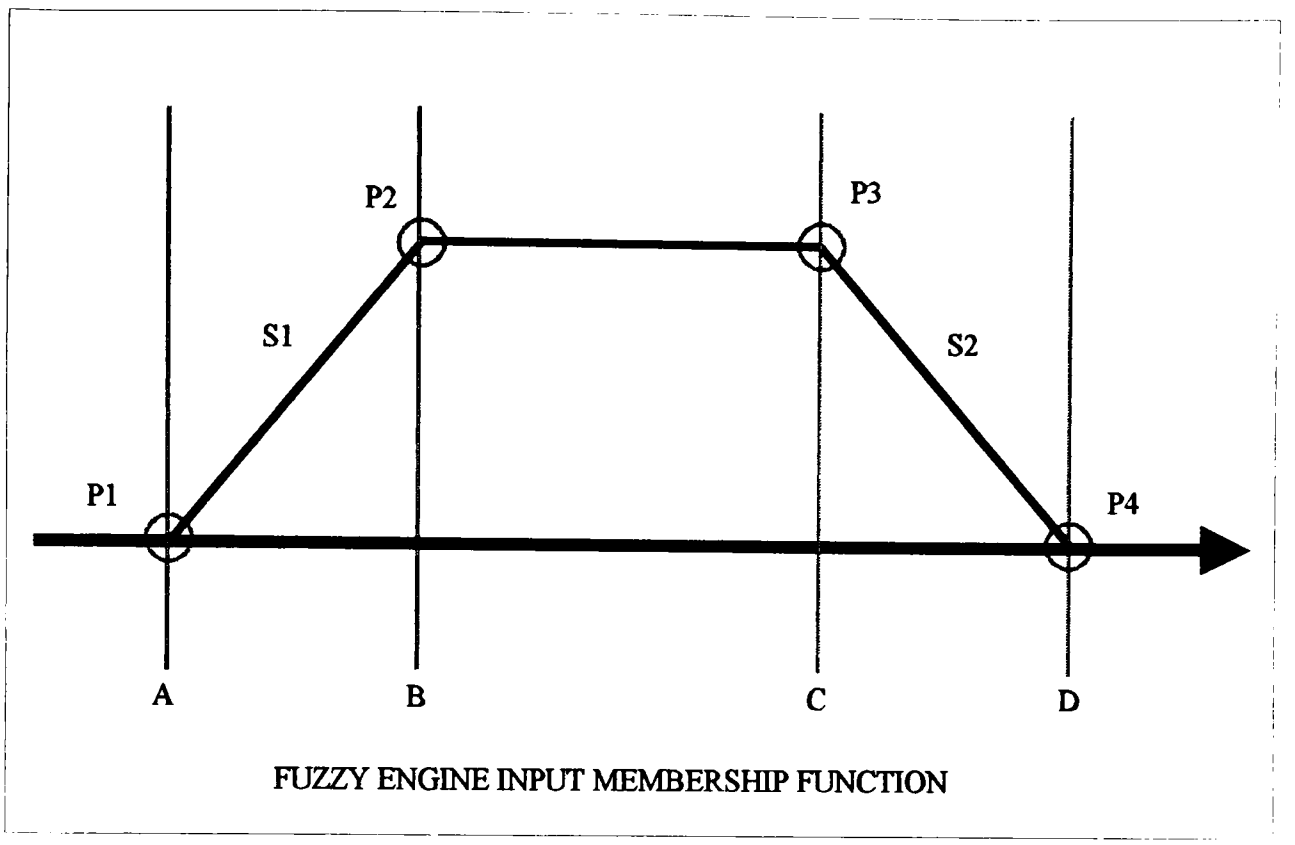


Figure 23. Fuzzy Input Membership Function

Table 4. Execution Cycles for Fuzzification of Crisp Inputs

Crisp Input Position ($I_{Crisp\ Input}$)	Clock Cycles
$I_{Crisp\ Input} <= A$	53
$A < I_{Crisp\ Input} < B$	91
$B <= I_{Crisp\ Input} <= C$	80
$C < I_{Crisp\ Input} > D$	102 **
$D <= I_{Crisp\ Input}$	61

** Worst Case Scenario

If the exact position of all the system's crisp inputs are known, then this table can be used to calculate the exact number of clock cycles required for the fuzzification process. Thus, the execution delay for the fuzzification process can also be determined. However, it is often impossible (and impractical) to predetermine the exact execution delays for all possible crisp input values. So, it is best to make a worst case estimation about the position of the crisp inputs when calculating the clock cycles for the fuzzification process.

Inspection of the table reveals that the $C < I_{Crisp\ Input} < D$ locations represent the worst possible position for all crisp inputs to be located. So, this worst case scenario is used to calculate the clock cycles for the fuzzification process in equation (8). A_{inputs} represents the number of crisp inputs in the system, and A_{msf} represents the maximum number of membership functions for any of the fuzzy inputs.

$$Fuzzification\ Cycles = A_{inputs}[\,(102\,A_{msf}) + 19\,].$$

Eq. (8) 68HC05 Clock Cycles for Fuzzification

Rule Evaluation Cycles. The next step in the FIP is rule evaluation. This specifically involves the calculation of fuzzy output vectors by the Min/Max correlation. The Min/Max correlation relates fuzzy input vectors to fuzzy output vectors by evaluating the engine's rule base.

For the FUDGE development environment, the fuzzy engine rule base is comprised of an arbitrary number of rules. With each rule containing one or more antecedents (the "if" part) and one or more consequences (the "then" part). The rules are not required have equal numbers of antecedents and consequences. Thus, one rule may contain five antecedents and two consequences, while another rule may have two antecedents and one consequence. Any combination of antecedents and consequences are allowed by the FUDGE tool and the Min/Max correlation. The only requirements are that each rule list all its antecedents first, followed by all its consequences, which is

standard practice for rules implementing the Min/Max correlation.

However, there are physical limitations to the overall number of antecedents and consequences allowed in the rule base. This number includes the addition of all antecedents plus all consequences for all the rules. This number must add up to some number less than or equal to 255. This limitation is due to the 256 addressable array size of the 68HC05 series of processors.

So, even though there is a physical size limitation to the rule base array, the actual implementation of the rules is quite flexible, and the FIP takes advantage of this flexible implementation style. The FIP evaluates the rule base array and applies the Min/Max correlation to determine the fuzzy inputs to fuzzy outputs mappings. In the rule base array the rules appear as a sequenced list of rule antecedents and consequences. So, the FIP simply evaluates each rule array element (one byte for each antecedent or consequence) in a sequential fashion. Therefore, the number of processor cycles needed to evaluate a rule antecedent or consequence depends on the value stored in the fuzzy input array corresponding to that antecedent or consequence. This fuzzy input value will have one of three possible conditions relating to the Min/Max correlation principle, as discussed by Kosko[2].

If the FIP is evaluating an antecedent, then the value will either be zero, the current minimum value or the current maximum value. The condition that is found will directly determine the number of processor clock cycles it takes to evaluate that antecedent. The relation between the antecedent value and the required processing cycles is listed in Table 5. Note that the worst case scenario for clock cycles is for the Minimum Antecedent condition.

Table 5. Execution Cycles for Antecedent Conditions

Antecedent Condition	Cycles
Zero	37
Minimum	68 **
Maximum	36

** Worst Case Scenario

If the FIP is evaluating a consequence, then the value will either contain the current minimum value or the current maximum value. The condition that is found will also directly determine the number of processor clock cycles it takes to evaluate a consequence. Unlike the antecedent evaluation, the zero possibility does not effect the processing cycles for consequence evaluation. The relation between the consequence value and the required processing cycles is listed in Table 6. Note that the worst case number of clock cycles is for the Maximum Consequence condition.

Table 6. Execution Cycles for Consequence Conditions

Consequence Condition	Cycles
Minimum	33
Maximum	39 **

** Worst Case Scenario

If the fuzzy input values for a system are known, then these tables can be used to calculate the exact number of processor clock cycles for the FIP rule evaluation step. However, this is not possible or practical for real-time system evaluations. So, equations (9) and (10) contain the worst case scenario formulas for calculating FIP antecedent and consequence processing cycles. Equation (9) assumes that B_{rules} is the number of rules

in the rule base and that each rule contains B_{ants} number of antecedents. Equation (10) assumes that B_{rules} is the number of rules in the rule base and that each rule contains B_{cons} number of consequences.

$$Antecedent\ Cycles = 68(B_{rules} * B_{ants}) + 17(B_{rules}) + 54.$$

Eq. (9) 68HC05 Clock Cycles for Antecedent Evaluation

$$Consequence\ Cycles = 36(B_{rules}) + 31(B_{rules} - 1) + 39(B_{cons}).$$

Eq. (10) 68HC05 Clock Cycles for Consequence Evaluation

Defuzzification Cycles. The final step of the Fuzzy Inference Process involves the defuzzification of the fuzzy output array. This step converts the vector representations of fuzzy outputs into crisp output values and utilizes the Center Of Gravity (COG) methodology described by Kosko [2]. The FUDGE environment allows the designer to implement fuzzy outputs with zero to eight membership functions. This number of output membership functions directly effects the number of processor clock cycles required to defuzzify an output.

The FIP will stop processing each fuzzy output after its last membership function has been evaluated. Thus, the number of clock cycles is directly related to the number of membership functions a fuzzy output contains. This relationship is shown in Table 7. This table assumes that C_{O_msf} is the number of membership functions for an individual fuzzy output. While C_{max_msf} is the fuzzy output with the maximum number of membership functions and is used to calculate the worst case scenario for the defuzzification step.

Table 7. Execution Cycles for Output Membership Functions

Defuzzify Condition	Cycles
Current Output has Less Than Max Number of Membership Functions	$[90\ (C_{0\_msf}-1) + 1112]$
Current Output has Maximum Number of Membership Functions	$[90(C_{max\_msf}) + 1069]^{**}$

** Worst Case Scenario

$$\textit{Defuzzification Cycles} = (33 * C_{outputs}) + (C_{outputs} * [(90\ C_{maxmsf}) + 1069]) + 12.$$

Eq. (11) Clock Cycles for Defuzzification

XFUDGE Translator

The XFUDGE program is a C++ code generator for Motorola's FUzzy Design GEnerator (FUDGE) tool. XFUDGE translates the C source code produced by the FUDGE tool into a functionally equivalent C++ class (object). This new "fuzzy" class provides the designer with a fuzzy engine that benefits from C++ object-oriented design.

The name XFUDGE comes from the fact that it "translates" the C code produced by FUDGE into new C++ code. So the "X" represents "trans" and the acronym for FUDGE is kept the same. Thus, the name comes from the program's purpose: to translate FUDGE code. So, the name is shortened to "XFUDGE."

Benefits of XFUDGE

Unfortunately, the C source code produced by the FUDGE tool is poorly commented and certainly not straightforward C coding. So, first time users may have to extensively study the C source code before using it in their own control system

implementation. In contrast, the design of the C++ fuzzy class is easy to learn and incorporate. A system designer can easily understand the fuzzy object's input/output interface without having to learn the details of the entire fuzzy class' code structure. So, the designer can then focus on the design of the control system, not the interface to the fuzzy engine, thus enabling the designer to produce more readable and maintainable code.

The C++ fuzzy class is designed to be used as a template for any generic fuzzy engine. So, each implementation of the fuzzy class is made unique by the addition of a fuzzy engine knowledge base (*.KNB) file. This knowledge base file contains the eleven control system specifications (data structures) as implemented by the system designer in the FUDGE development tool.

The XFUDGE program reads (as input) the C source code file produced by the FUDGE tool. It extracts the fuzzy engine's design specifications embodied in the C input file and creates (as output) a new knowledge base file for the C++ fuzzy class implementation. Afterwards, only one small change must be made to the "fuzzy.hpp" file in order to implement a new fuzzy object. This change (to the fuzzy.hpp file) is to the filename listed for the KNB include statement. Figure 24 contains the list of steps needed to create a new C++ fuzzy engine implementation:

Steps to Create a C++ Fuzzy Engine:

1) Create and test the fuzzy engine design in the FUDGE development software.
2) Use FUDGE to create the C source code (fuzzy.c) for the fuzzy engine implementation.
3) Use the XFUDGE translation software to create a KNB file from the C source code.
4) Include the KNB filename in the fuzzy class definition (fuzzy.hpp) as an '#include "filename.knb"' statement.
5) Compile the fuzzy class (fuzzy.cpp) implementation of the fuzzy engine.

Figure 24. XFUDGE Translation Procedure

XFUDGE Programs

There are three programs developed for the XFUDGE translation software. The first two programs are identical except that they have been compiled for different operating systems. While the third program is simply a GUI (Graphical User Interface) for the Windows version of XFUDGE. The first program, xfudge_d.exe, was compiled to run as a DOS application. While the xfudge_w.exe program was compiled to run as an Easy Windows application in Microsoft's Windows 3.1. As previously mentioned, the third program runs as a simple GUI interface to the xfudge_w.exe program. Refer to Figure 25 for descriptions of the three programs.

XFUDGE Program Descriptions:

XFUDGE_D.EXE	-	DOS executable
XFUDGE_W.EXE	-	Windows 3.1 executable
XFUDGE.EXE	-	GUI interface for the XFUDGE_W program.

Figure 25. Available XFUDGE Program Descriptions

XFUDGE_D.EXE

This program translates FUDGE (C source) code into a XFUDGE (C++ knowledge base) source file. This knowledge base file (.KNB extension) is the heart of the fuzzy class definition. The KNB file contains eleven data structures that define the fuzzy control system's operation. In short, these data structures contain the fuzzy engine specifications as implemented by the system designer in the FUDGE development tool. The command line options for the DOS version of XFUDGE are shown in Figure 26.

XFUDGE Command Line Parameters:

```
        xfudge_d.exe [<input filename> <output filename>] [show translation:
1|0]

[OPTIONAL] command line parameters:
   input filename     =   filename of the C source code created by FUDGE.
                          Default is "fuzzy.c"
   output filename    =   filename of output knowledge base (KNB) filename.
                          Default is "fuzzy.knb"
   show transition    =
      0 - creates KNB file without showing the translation results on the screen.
      1 - shows the translation results on the screen. (Helpful for debugging.)

      Default is 1.

xfudge_d.exe with no command line parameters =
    The program will prompt user for all required input/output filenames.
```

Figure 26. XFUDGE Command Line Options

XFUDGE_W.EXE

It performs the same functions as xfudge_d.exe, except it runs in Windows 3.1. It also has the same command line options as the xfudge_d.exe shown in Figure 26.

XFUDGE.EXE

An easy to use Graphical User Interface (GUI) for the xfudge_w.exe program, it automatically creates the command line parameters for the xfudge_w program. It calls and executes the xfudge_w program from the graphical interface. This is the preferred and recommended method for executing the XFUDGE translation software.

XFUDGE C++ Implementation

The XFUDGE software creates the knowledge base for the C++ implementation of the fuzzy engine. The output knowledge base created by the XFUDGE program is

stored in a KNB (KNowledge Base) file. This file can be given any name by the user, but its default filename is "fuzzy.knb." This KNB file contains all of the constant data structures that make a fuzzy engine unique. These data structures are shown in Figure 27 and represent the washing machine example developed in this thesis. The unique KNB's data structures describe the fuzzy inputs (including input membership functions), fuzzy rule base and the fuzzy outputs (including output membership functions) for the Fuzzy Inference Processor of the fuzzy engine. Refer to Figure 21 to review the fuzzy engine block diagram. Remember that the data structures themselves are common to all fuzzy engines. However, it is the constant values contained in these data structures that make the fuzzy engine's operation unique.

The C++ class developed for the fuzzy engine is contained in the "fuzzy.hpp" file and shown in Figure 28. This class structure does not vary from one engine implementation to the next engine implementation. It has been designed this way to increase the C++ code reusability of the fuzzy class. Note that the fuzzy.hpp file contains a KNB include (#include "fuzzy.knb") statement. This statement is required and must contain the filename of the unique fuzzy engine knowledge base. The filename for the KNB file is assumed to be "fuzzy.knb". However, this filename may be different if the user created a KNB file with a different name.

This included KNB file contains the unique data structures that define the FIPs operation. Thus, the constructor for the fuzzy class object initializes the private data structures with the appropriate values from the KNB file.

```
//DEFINES
#define MIN(A,B)  (A < B) ? A : B
#define MAX(A,B)  (A > B) ? A : B
#define NUM_INPUTS 2
#define NUM_OUTPUTS 1
#define NUM_RULES 9

//STRUCTURES
struct Rule
{           int   antecedent[8];
            int   consequent[8];
};

//CONSTANTS
const struct Rule Rules[NUM_RULES] =
{           { { 0x10, 0x11 }, { 0xa0 } },
            { { 0x08, 0x11 }, { 0x98 } },
            { { 0x00, 0x11 }, { 0x98 } },
            { { 0x10, 0x09 }, { 0x98 } },
            { { 0x08, 0x09 }, { 0x90 } },
            { { 0x00, 0x09 }, { 0x90 } },
            { { 0x10, 0x01 }, { 0x90 } },
            { { 0x08, 0x01 }, { 0x88 } },
            { { 0x00, 0x01 }, { 0x80 } }
};
const int num_input_mfs[NUM_INPUTS] =  { 3, 3 };
const int num_output_mfs[NUM_OUTPUTS] =  { 5 };
const int num_rule_ants[NUM_RULES] =  { 2, 2, 2, 2, 2, 2, 2, 2, 2};
const int num_rule_cons[NUM_RULES] =  { 1, 1, 1, 1, 1, 1, 1, 1, 1};
const int inmem_points[NUM_INPUTS][7][4] =
{           {
                        { 0.000000, 0.000000, 0.000000, 50.000000 },
                        { 0.000000, 50.000000, 50.000000, 100.000000 },
                        { 50.000000, 100.000000, 100.000000, 100.000000 }
            },          {
                        { 0.000000, 0.000000, 0.000000, 50.000000 },
                        { 0.000000, 50.000000, 50.000000, 100.000000 },
                        { 50.000000, 100.000000, 100.000000, 100.000000 }
            }
};
const int outmem_points[NUM_OUTPUTS][7][4] =
{           {
                        { 8.000000 },
                        { 12.000000 },
                        { 20.000000 },
                        { 45.000000 },
                        { 60.000000 }
            }
};
```

Figure 27. Example Fuzzy Engine Knowledge Base File

```
//INCLUDE FUZZY ENGINE KNOWLEDGE BASE
#include "fuzzy.knb"   // NOTE: Change to match your KNB filename.

// FUZZY CLASS DECLARATION
class fuzzy
{
 public:
        fuzzy();                        // constructor
        void  fuzzy_step();             // call to execute the fuzzy engine
        float &operator[](int);         // [ ] operator for crisp input and outputs

 private:
        // KNOWLEDGE BASE DATA VALUES
        int    num_inputs;              // number of crisp inputs
        int    num_outputs;             // number of crisp outputs
        int    num_rules;               //  number of rules

        // FUZZY VARIABLES
        float  crisp_inputs[NUM_INPUTS];  // holds crisp value for each input
        float  crisp_outputs[NUM_OUTPUTS]; //holds crisp val for each output
        float  fuzzy_inputs[8][8];        // 8 inputs each with 8 msf; max
        float  fuzzy_outputs[4][8];       // 4 outputs each with 8 msf; max

        // PRIVATE FUNCTIONS
        void   fuzzify_input(int, float);       // converts from crisp to fuzzy values
        float  get_membership_value(int, int, float);  // create fuzzy vector

        void   correlation(int);                // min/max correlation of fuzzy values
        float  defuzzify_output(int, float *);  // COG calculations for crisp outputs
}; //end fuzzy class

#endif
```

Figure 28. Fuzzy Class Definition

Example Testing and Evaluation of Results

The three example fuzzy logic systems developed in this thesis are the Washing Machine, Traffic Light, and Truck Backer-Upper controllers. Each control system was created, tested and implemented with the FUDGE tool. During this process they were each implemented as 68HC05, C and C++ versions of the fuzzy engine. The C and C++ engines were each extensively tested for correct crisp input to crisp output calculations. Afterwards, all three were tested against their appropriate hardware models.

C and C++ Fuzzy Engine Comparisons

Both the C and C++ versions of the three fuzzy engine implementations were extensively tested for correctness. Each engine was incorporated into a test program. This test program incremented each crisp input by a tenth (0.10) of a point and executed the fuzzy engine. The crisp outputs for each example were then stored in an output file. Then a systematic comparison was done between the C and C++ versions of each fuzzy engine. Afterwards, a random check of the values in the output file were also tested against the crisp output values calculated by the FUDGE environment. The test was declared a failure if any output values did not match among the C, C++ and FUDGE results. As a result of this extensive testing, none of the tests failed, thus proving the C and C++ implementations of the fuzzy engine.

68HC05, C and C++ Hardware Model Tests

The hardware models for the FUDGE tool were evaluated against the three examples developed in this thesis, each example was implemented as a C, C++ and 68HC05 fuzzy engine. The applicable hardware models for memory and processing power requirements were applied to these example fuzzy engines. The memory and processing requirements were then hand evaluated from the source code of each engine implementation. After comparing the hand evaluation to the results derived from the hardware models. Each model was determined to be correct.

All three fuzzy engine examples have been evaluated for their hardware requirements in the 68HC05 series of microcontrollers. Figure 29 shows the number of RAM bytes required by each example. For each example case, the RAM memory model 1 (Eq. (2)) was used to predetermine the amount of memory required. Then the assembly code of each example was inspected to determine the true number of RAM bytes required. As shown in the figure, the actual memory required by each example exactly matched the RAM model predictions.

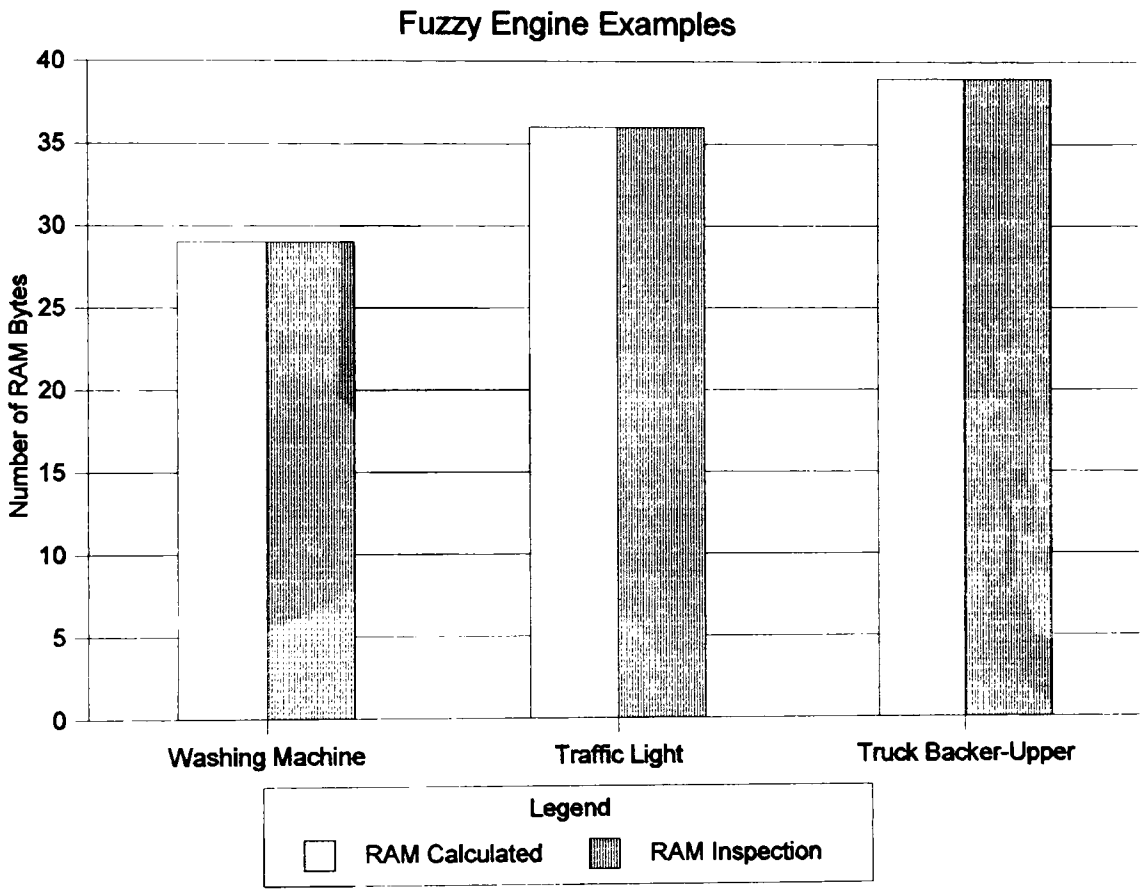


Figure 29. 68HC05 RAM Bytes for Three Fuzzy Engine Examples

Next, the ROM model for the 68HC05 series of microcontrollers was tested. As shown in Figure 30 . The amount of required ROM was calculated by Eq. (1), the worst case ROM memory model. Afterwards, the code for each example was inspected to determine the actual number of ROM bytes required to implement the engines.

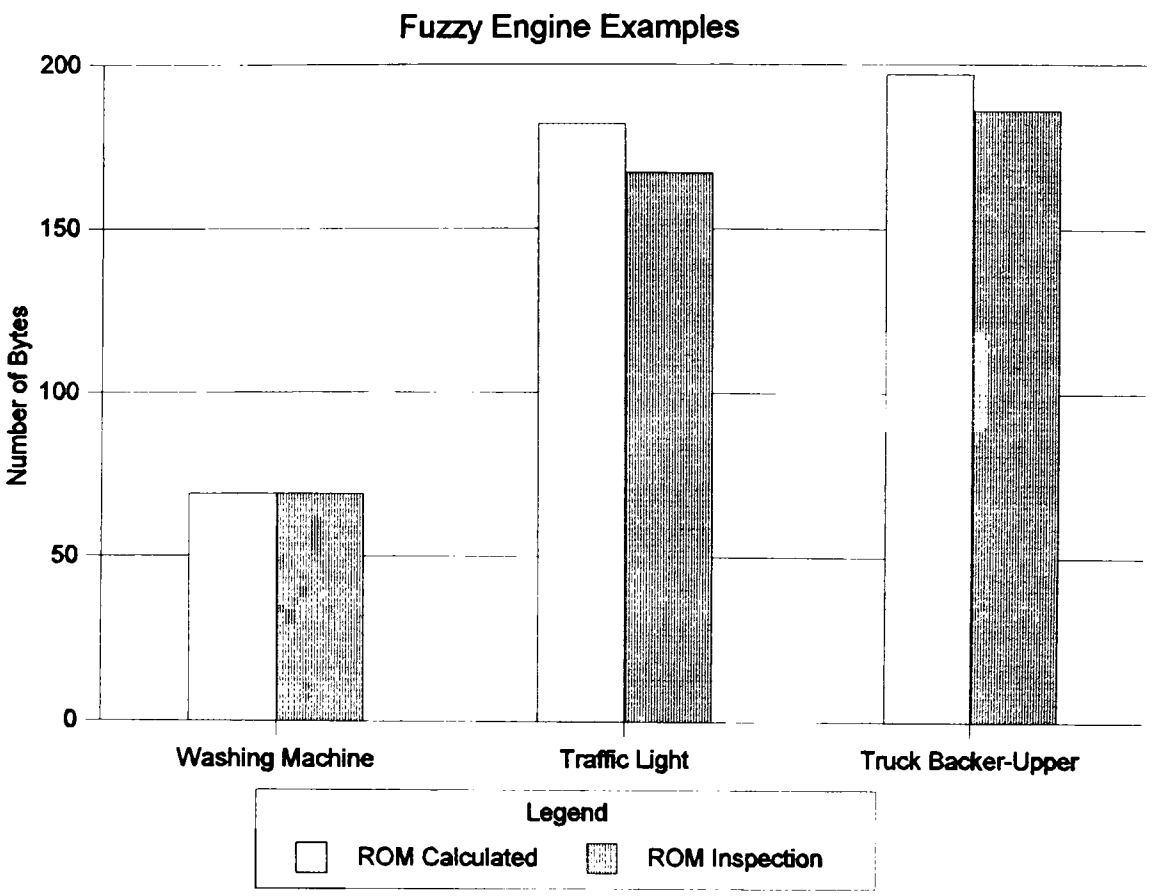


Figure 30. 68HC05 ROM Bytes for Fuzzy Engine Examples

The ROM model in Eq. (1) is a worst case model. It assumes that all inputs have A_{msf} number of membership functions and that all outputs contain C_{msf} number of output functions. It also assumes that each rule contains $[B_{ants} + B_{cons}]$ number of antecedents and consequences and that B_{ants} and B_{cons} represent the maximum number of each respective rule antecedent or consequence.

In the simple washing machine example, the amount of predicted ROM and actual ROM requirements were exactly equal. In the other two examples, the predicted ROM amount was slightly higher than the actual amount of ROM. This discrepancy can be explained by evaluating the rule base of each example. In the Washing Machine example, each rule contains the exact maximum number of rule antecedents and consequences (2 :1), and each input/output also has the same number of membership

functions. Thus, its actual ROM requirements match the expected number of bytes determined by the model.

In the last two examples, the actual number of membership functions for each input/output may be less than A_{msf} or C_{msf}. Plus, each rule may or may not contain the maximum number of rule antecedents or consequences. These facts account for the lesser amount of ROM required by the last two examples. Note that these models determine the greatest amount of memory needed implement the fuzzy engine design. The actual amount will always be equal to or less than these predicted values.

Each of the three fuzzy engine examples were also evaluated to determine their predicted execution cycles and processing delays for a 68HC05 series microcontroller with a 4.0 MHz oscillator. Table 8 contains the expected number of cycles for each process of the FIP and the processing delay for the entire fuzzy engine's execution. These values were obtained from the models for Fuzzification (Eq.(8)), Rule Evaluation (Eq(9) and (10)), Defuzzification (Table 7.), Total Execution (Eq. (6)) and Execution Delay (Eq(7)).

Table 8. Execution Cycles and Time Delays for Fuzzy Engine Examples

Example	Fuzzify Cycles	Rule Cycles	Defuzzify Cycles	Total Cycles	Execution Delay
Wash	650	2047	1564	4276	2.14 msec
Traffic	1281	7550	1564	10415	5.21 msec
Truck	1466	7762	1744	10992	5.50 msec

CHAPTER V
CONCLUSION

The research in this thesis developed models to predict the hardware requirements for fuzzy logic control systems implemented with Motorola's FUzzy Design GEnerator (FUDGE) and also expands the high level programming languages supported by the FUDGE environment. These hardware models specify the memory and processing power requirements for control system designs developed with the FUDGE tool. Also, the high level languages supported by FUDGE now include the C++, object-oriented, programming language, which is implemented by a C to C++ translation program called XFUDGE.

Hardware Prediction Models

The first portion of this thesis develops two basic fuzzy logic hardware models. The first model describes the fuzzy engine memory requirements. While the second describes the processing delays for fuzzy logic control systems implemented with the FUDGE software. These models predict the memory and processing power requirements needed to implement a proposed fuzzy logic design, and they can be used with Motorola's 68HC05 line of microcontrollers or the larger more complex microprocessors found in personal computers.

Microcontroller Support

A microcontroller is a stand alone processing element, and it is designed for small memory/low processing power applications. They are marketed as "all-in-one" controllers and provide a simple encapsulated package with microprocessor, memory and a basic input/output interface already integrated into a single package. They are especially practical in small, inexpensive, control system applications. Therefore, the models in this thesis are especially beneficial to any designer trying to implement fuzzy logic control systems with these inexpensive microcontrollers.

Hardware Model Benefits

Since design engineers commonly need to make estimates about system hardware requirements (such as memory size and processing power), as soon as possible in the design process. The formulas described in this thesis help designers to make quick and accurate predictions about the hardware requirements of their fuzzy control implementation. With this information, designers can play with the "what ifs" of their design. For example, they can determine the physical consequences of doubling the number of rules in the rule base or adding an addition input or membership function to the system, thus allowing the design engineer to make important design decisions early in the design process, decisions like trade-offs between application "extras" and the hard physical limitations of the design. This prevents these extra features from causing problems later in the design process, problems like getting to the prototyping stage and unexpectedly having the system to runs out of memory or processing power.

XFUDGE: C++ Object-Oriented Support

The second portion of this thesis relates to increasing the number of high level languages supported by the FUDGE tool. The current version of FUDGE (Version 1.02) supports several of Motorola's assembly languages, as well as the ANSI C language. Now, a translation program (called XFUDGE) can convert the fuzzy engine (C source code) created by the FUDGE program into a functionally equivalent object-oriented, C++ fuzzy engine. This C++ code (a "fuzzy" class) allows the design engineer to implement a fuzzy engine in the popular C++ language.

Object-Oriented Benefits

Object-oriented programming languages offer several benefits to the programmer, two of which include data encapsulation (protection) and an easier programmer interface to the object. Object-oriented languages also relate software design to natural real-world objects, and fuzzy logic relates natural real-world descriptions to control system designs. This illustrates how these two design methodologies complement each other and the

designer's symbolic thought processes so well. This all boils down to the designer being able to quickly and easily develop robust control system applications. In short, it provides an easy interface between the system engineer, the fuzzy engine and the control system design, which simplifies and enhances the natural symbolic interface between the designer and the design.

Further Fuzzy Logic Research Topics

The Fuzzy Design Generator (FUDGE) is certainly still in its infancy stage, and the FUDGE environment still needs further improvements. First of all, it should support the hardware model predictions, as well as the C++ language from within its own application. Secondly, it should have several improvements made to the FIP (Fuzzy Inference Processor) implementation of the fuzzy engine.

Extended FUDGE Hardware Model Support

Currently, the FUDGE development environment does not directly support the hardware models developed in this thesis. These models are definitely an important part of the design process, and they should be incorporated into the FUDGE environment beginning with the creation of similar models for some of the other microcontrollers and microprocessors supported by the FUDGE tool, which include the Motorola 68HC11, 68HC16 and 68000 series of processing elements.

These processing elements (excluding the powerful 68000 series) have relatively limited available memory and processing throughput. So, it is important to always make these memory and processing throughput estimations prior to implementing a control system design. Therefore, the FUDGE environment should be able to make these basic hardware predictions based on a fuzzy control system's preliminary design parameters.

Incorporate FUDGE C++ Support

The popular C++, object oriented programming language naturally fits with the real-world (symbolic) object descriptions used in fuzzy logic design. Thus, it will

certainly play an ever increasing role in fuzzy logic design. The "fuzzy" engine class has been designed to simplify the interface between the system designer, the software and the controller implementation. This enables a system designer to create controller software with object-oriented procedures that relate to the fuzzy logic system design. This fuzzy class also provides data encapsulation and operation overloading to help to create a robust and maintainable software application.

Improvements to FUDGE Fuzzy Inference Processor

The current version of the FIP only allows for singletons as output membership functions. This prevents any fine tuning of the output by adjusting the shape of the system's output membership functions. As a result, to increase output sensitivity the designer is forced to increase the number of output membership functions. Thus, a system designer must at least increase the number of rules in the system, which has a direct negative affect on the hardware models of the system. In contrast, a fuzzy logic system with trapezoidal membership functions would allow the system designer more control over the system and would not greatly increase the amount of required memory or produce lengthy processing delays.

Summation

This research benefits any person using the FUDGE development system to implement fuzzy logic control systems. It also provides helpful design and development information during the fuzzy controller implementation process. While the XFUDGE translator program provides the user with another (better) high level language option for implementing their fuzzy control designs.

REFERENCES

[1] Sibigtroth, J. (1992). Implementing Fuzzy Expert Rules in Hardware. AI Expert, April.

[2] Kosko, B., (1992). Neural Networks and Fuzzy Systems: A dynamical Systems Approach to Machine Intelligence. Englewood Cliffs, N J: Prentice-Hall, Inc.

[3] Sibigtroth, J. (1991). Creating Fuzzy Micros. Embedded Systems: Programming, v4, 12.

[4] Jamshidi, M., Vadiee, N., Ross, T., (1993). Fuzzy Logic and Control: Software and Hardware Applications. Englewood Cliffs, N J: Prentice-Hall, Inc. 263-278.

[5] Weiss, D. (1994). Description of the 68HC05 Fuzzy Inference Engine (Technical Brief). Austin, TX: Motorola Corporation, Motorola SPS Sector Technology PSP/CECT.

[6] Klir, G. J., & Forger, T. A., Fuzzy Sets, Uncertainty, and Information, Englewood Cliffs, NJ: Prentice Hall, 1988.

[7] Weijing, Z., (1992). Washing Machine (FIDE Application Note No. 001-920727). San Jose, CA: Aptronix Incorporated.

[8] Mendel, J. (1995). Fuzzy Logic Systems for Engineering: A Tutorial. IEEE: Periodicals, 4, 345-375.

APPENDIX A
XFUDGE TRANSLATOR MAIN SOURCE CODE

```
/*-------------------------------------------------------------------------
PROGRAM: fudgeXpp.cpp
AUTHOR: Mark Workman
DATE: 8/21/96
VERSION: 1.0
DESCRIPTION:This program translates the C language fuzzy engine created by
Motorola's Fuzzy Development GEnerator (FUDGE) into an equivalent C++ fuzzy
class structure. It will query the C code implementation for information
needed to build the C++ knowledgebase.

INPUT PARAMETERS:
xFudge [input C code filename][output knb filename][show translation{1 true}|{0 false}]
-------------------------------------------------------------------------*/

//INCLUDES
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include "xfudge.hpp"
#include "string.hpp"
#include <conio.h>

//DEFINES
#define VER "1.0"
#define PROG_NAME "\nxFUDGE - C to C++ translation program."
#define INPUT_FILE   "fuzzy.c"   //default input file
#define HEADER_FILE  "header.in" //default header file
#define KNB_FILE     "fuzzy.knb" //default output file

int main(int argc, char *argv[])
{       String szfIn = INPUT_FILE;
        String szfHeader = HEADER_FILE;
        String szfKnb = KNB_FILE;
        String szShow = "1";
        ifstream pfFuzzyIn;
        ifstream pfHeaderIn;
        ofstream pfKnbOut;
        char szLine[126];
```

```
//IN ALL INPUT PARAMs ARE USED
if(argc==4)
{
        szfIn = argv[1];    //first param contains the input file string
        szfKnb = argv[2];   //sec param contains the output file string
        szShow = argv[3];   //third param determines if to show KNB construction
        if(szShow!="0")
        {       clrscr();
                //OPENING HEADER
                cout << PROG_NAME << "  Version #" << VER << endl;
         }
}//end if all input params used
//IF TOO MANY INPUT PARAMETERS ARE USED
else if(argc>4)
{
        clrscr();
        //OPENING HEADER
        cout << PROG_NAME << "  Version #" << VER << endl;
        cout << "\nError: Invalid or to many parameters\n";
        exit(1);
}//end else if argc>=4
//IF NO SHOW PARAM IS SPECIFIED
else if(argc==3)
{
        szfIn = argv[1];    //first param contains the input file string
        szfKnb = argv[2];   //sec param contains the output file string
        clrscr();
        //OPENING HEADER
        cout << PROG_NAME << "  Version #" << VER << endl;
}//end if 3 input params used
//ALL OTHER
else    {
        char cAns=' ';
        clrscr();
        //OPENING HEADER
        cout << PROG_NAME << "  Version #" << VER << endl;
        cout << "\n Do you want to translate the C file \""
                << szfIn << "\" : ";
        cAns=getche();
        cAns=toupper(cAns);
        if(cAns!='Y')
        {  cout << "\nEnter another input filename: ";
                cin >> szfIn;
        }
        cout << "\n Do you want to create the C++ KNB file \""
                << szfKnb << "\" : ";
        cAns=getche();
        cAns=toupper(cAns);
        if(cAns!='Y')                   {
                cout << "\nEnter another ouput KNB filename: ";
                cin >> szfKnb;
        }
}//end else no input params
```

```
//Declare XLation object
xfudge Xlate(szShow);

//Test for all file opened
pfFuzzyIn.open(szfIn.charStrg());
if(!pfFuzzyIn)
{       cerr << "\nCouldn't open file: " << szfIn << endl;
        exit(2);
}
pfHeaderIn.open(szfHeader.charStrg());
if(!pfHeaderIn)
{       cerr << "\nCouldn't open file: " << szfHeader << endl;
        exit(3);
}
pfKnbOut.open(szfKnb.charStrg());
if(!pfKnbOut)
{       cerr << "\nCouldn't open file: " << szfKnb << endl;
        exit(4);
}

//Remove knb data structures from FUDGE (C code) fuzzy engine file
pfFuzzyIn >> Xlate;

//Search through C file for data structs
if(szShow!= "0")
        cout << "\n\nCreating new knowledge base...";
while(!pfFuzzyIn.eof())
{
        pfFuzzyIn >> Xlate; //build knb strings
}//end FuzzyIn while

//WRITE THE KNB FILE
if(Xlate.Complete())        //Make sure that the knb has been completed
{  if(szShow!="0")
                cout << "\nWriting new knowledge base file!!!";

        //Create the header for the output knowledge base file
        pfHeaderIn.getline(szLine, 125);
        while(!pfHeaderIn.eof())
        {
                pfKnbOut << szLine << endl;
                pfHeaderIn.getline(szLine, 125);
        }//end pfHeader while
        pfKnbOut << "\n\n#ifndef _FUZZY_KNB"
                                << "\n#define _FUZZY_KNB";
        pfKnbOut << Xlate; //save the new knowledge base to the output file
        pfKnbOut << "\n#endif\n";

        if(szShow!="0")
                cout << "\nNew KNB file \"" << szfKnb
                  << "\" created from \"" << szfIn << "\"."
                  << "\nDone.";
}
```

```
        else
        {
                cout << "\nERROR:\n" << "Could not find one or more portions of the\n"
                        << "    fuzzy engine data stucts in the input file!!\n";
                exit(5);
        }
        pfFuzzyIn.close();
        pfHeaderIn.close();
        pfKnbOut.close();
return 0;
}//end main
```

APPENDIX B
XFUDGE TRANSLATOR CLASS DEFINITIONS

```
/*-------------------------------------------------------------------------
PROGRAM: xFUDGE.hpp
AUTHOR:  Mark Workman
DATE :   8/22/96
VERSION: 1.0
DESCRIPTION:This file contains the objects that will interpret the fuzzy
engine knowlege base data structures found in the C files created by the
FUDGE (v1.04) development program. This C++ code translates the C language
fuzzy engine created by FUDGE into an equivalent C++ fuzzy class structure.
It will query the C code implementation for information needed to build
the C++ knowledgebase.

Elements of the knowledge base are:
        NUM_INPUTS, NUM_OUTPUTS, NUM_RULES,
        RULE, RULES, num_input_mfs, num_output_mfs, num_rule_ants,
        num_rule_cons, inmem_points, outmem_points
-------------------------------------------------------------------------*/
#ifndef _XFUDGE_H
#define _XFUDGE_H

//INCLUDE
#include "string.hpp"

//DEFINES
enum bool {FALSE, TRUE};   //Boolean definitions

//Each portion of the knowledge base is considered an 'element' of the engine.
//element class
class elem
{
        String strg;
        bool complete;

        public:
        elem() {complete = FALSE;}   //constructor

        //accessors
        bool Comp() {return complete;}       //return if elem complete

        //overloaded functions              //inline functions
        bool &operator= (const bool in)       {complete = in; return complete;}
        char * operator= (char *in)           {strg = in; return in;}
        String & operator= (const String &in) {strg = in; return strg;}
        String & operator+= (char *in)        {strg += in; return strg;}
        String & operator+=(const String &in) {strg += in; return strg;}

        friend ostream & operator<< (ostream &out, const elem &element)
                {out<<element.strg; return out;}
};//end element
```

```
//TRANSLATOR CLASS
class xfudge : private elem
{
        public:
        xfudge(const String &szShow = "1");     //constructor
        bool Complete(); //are all the data element strings complete?
        String Show;


        private:      //each kb element contains a string with the fuzzy engine
                                                        //  definitions from the FUDGE development tool.
        elem NUM_INPUTS;
        elem NUM_OUTPUTS;
        elem NUM_RULES;
        elem Rule;
        elem Rules;
        elem num_in_mfs;
        elem num_out_mfs;
        elem num_rule_ants;
        elem num_rule_cons;
        elem inmem_points;
        elem outmem_points;

        //MEMBERSHIP FUNCTIONS
        void varName(String &inStrg);

        friend ostream & operator<< (ostream &output, xfudge &element);
        friend istream & operator>> (istream &input, xfudge &element);
};//end xfudge class

#endif
```

APPENDIX C
XFUDGE TRANSLATOR CLASS FUNCTIONS

```
/*-------------------------------------------------------------------------
PROGRAM  :xFUDGE.cpp
AUTHOR   :Mark Workman
DATE     :8/21/96
VERSION  :1.0
DESCRIPTION: This file contains the implementations for the xfudge class.
the xfudge object translates the C language fuzzy engine created by Motorola's
Fuzzy Development GEnerator (FUDGE) into an equivalent C++ fuzzy class
structure. It will query the C code implementation for the information needed
to build the C++ knowledge base.

Elements of the knowledge base are:
        NUM_INPUTS, NUM_OUTPUTS, NUM_RULES,
        RULE, RULES, num_input_mfs, num_output_mfs, num_rule_ants,
        num_rule_cons, inmem_points, outmem_points
-------------------------------------------------------------------------*/
#ifndef _XFUDGE_CPP
#define _XFUDGE_CPP

//INCLUDE
#include "xfudge.hpp"
#include <ctype.h>


//CONSTRUCTOR
xfudge::xfudge(const String &szShow)
{
  Show = szShow;    //determines if the output will be shown on screen
        //INITIALIZE all string elements of the knowledge base
        //strings for defines
        NUM_INPUTS = "\n#define NUM_INPUTS ";
        NUM_OUTPUTS = "\n#define NUM_OUTPUTS ";
        NUM_RULES = "\n#define NUM_RULES ";

        //string for rule structure
        Rule =
        "\n\n\n//STRUCTURES\
        \nstruct Rule\n{\n\
          int   antecedent[8];\n\
          int   consequent[8];\
        \n};\n";
        Rule = TRUE;

        //Strings for Constants
        Rules =
        "\n\n//CONSTANTS\
        \nconst struct Rule Rules[NUM_RULES] = ";
        num_in_mfs = "\nconst int num_input_mfs[NUM_INPUTS] = ";
        num_out_mfs = "\nconst int num_output_mfs[NUM_OUTPUTS] = ";
        num_rule_ants = "\nconst int num_rule_ants[NUM_RULES] = ";
```

```
        num_rule_cons = "\nconst int num_rule_cons[NUM_RULES] = ";
        inmem_points = "\nconst int inmem_points[NUM_INPUTS][7][4] = ";
        outmem_points = "\nconst int outmem_points[NUM_OUTPUTS][7][4] = ";

}//end constructor

bool xfudge::Complete()
{
 if( NUM_INPUTS.Comp()
                && NUM_OUTPUTS.Comp()
                && NUM_RULES.Comp()
                && Rule.Comp()
                && Rules.Comp()
                && num_in_mfs.Comp()
                && num_out_mfs.Comp()
                && num_rule_ants.Comp()
                && num_rule_cons.Comp()
                && inmem_points.Comp()
                && outmem_points.Comp())
        return TRUE;
 return FALSE;
}//end Complete

//PRIVATE MEMBERSHIP FUNCTIONS
//*********************************************************************
// varName function - removes any extra characters from the String
//  containing a variable name. (ie. "Rules[9] " becomes "Rules")
//  Name can't begin with spaces, punct or numbers. It receives the inStrg
//  by reference. So all changes to inStrg will be reflected in the
//  original copy of the string.
void xfudge::varName(String &inStrg)
{
        char szTemp[256];
        int i=0, iLength = inStrg.length();

        //loop until an index contains something other than an alphanum or '_' char
        while((isalpha(inStrg[i]) || inStrg[i]=='_') && i<=iLength)
        {
                szTemp[i] = inStrg[i];
                i++;
        }
        szTemp[i] = '\0'; //replace anything else with null
        inStrg = szTemp;
}//end varName
```

```
//FRIENDS
//*************************************************************************
// operator << - overloads the inserter for the xfudge class. It outputs
//    each element string in the xfudge class to the output stream.
ostream & operator << (ostream &output, xfudge &element)
{
        output << element.NUM_INPUTS << element.NUM_OUTPUTS
                        << element.NUM_RULES << element.Rule << element.Rules
                        << element.num_in_mfs << element.num_out_mfs
                        << element.num_rule_ants << element.num_rule_cons
                        << element.inmem_points << element.outmem_points;
        return output;
}//end <<

//*************************************************************************
// operator >> - overloads the extractor for the xfudge class. It expects
//    the instream to be from a file. The file should be the fuzzy engine
//    (C code) output file created by FUDGE. This function locates the
//    required data structures needed to build the CPP knowledge base.
//    It then appends the assignment statements to the appropriate xfudge
//    data element string.
istream & operator >> (istream &input, xfudge &element)
{
        String inStrg;    //temp input string holder

        input >> inStrg;  //get a string from the stream

        //search the input stream for the required data elements
        if(inStrg=="int")      //check all data elementts of type int
        {
                input >> inStrg;      //get the variable name from the stream
                element.varName(inStrg);      //remove any extra chars from the variable name
                //test for all the possible int variables
                if(inStrg=="num_inputs")
                {
                 input >> inStrg >> inStrg; //skip the assignment stmt '='
                 inStrg[1] = '\0';
                 element.NUM_INPUTS += inStrg; //append the num of inputs in the system
                 element.NUM_INPUTS = TRUE;    //mark it as complete
                 if(element.Show!="0")
                                cout << "\n\tElement 'num_inputs' is COMPLETE!";

                }
                else if(inStrg=="num_outputs")
                {
                 input >> inStrg >> inStrg;
                 inStrg[1] = '\0';
                 element.NUM_OUTPUTS += inStrg; //append the num of outputs in the system
                 element.NUM_OUTPUTS = TRUE;   //mark it as complete
                 if(element.Show!="0")
                                cout << "\n\tElement 'num_outputs' is COMPLETE!";

                }
```

```
else if(inStrg=="num_rules")
{
 input >> inStrg >> inStrg;
 //look for num_rules > 9
 for(int k=0; k<3; k++)
          if(inStrg[k] == ';')
           inStrg[k]= '\0';
 element.NUM_RULES += inStrg; //save the  num of rules in the sys
 element.NUM_RULES = TRUE;    //mark it as complete
 if(element.Show!="0")
                    cout << "\n\tElement 'num_rules' is COMPLETE!";

}
else if(inStrg=="num_input_mfs")
{ char szTemp[256];       //a temp string to save the assignment stmt
 input >> inStrg;
 input.get(szTemp, 255);  //retrieve all data between the braces { and }
 inStrg = szTemp;
 element.num_in_mfs += inStrg; //append the assignment stmt to the string
 element.num_in_mfs = TRUE;    //mark it as complete
 if(element.Show!="0")
                    cout << "\n\tElement 'num_inputs_mfs' is COMPLETE!";

}
else if(inStrg=="num_output_mfs")
{ char szTemp[256];
 input >> inStrg;
 input.get(szTemp, 255);    //retrieve  data between { and }
 inStrg = szTemp;
 element.num_out_mfs += inStrg; //append the assignment stmt to the string
 element.num_out_mfs = TRUE;    //mark it as complete
 if(element.Show!="0")
                    cout << "\n\tElement 'num_output_mfs' is COMPLETE!";

}
else if(inStrg=="num_rule_ants")
{ char szTemp[256];
 input >> inStrg;
 input.get(szTemp, 255);    //retrieve  data between { and }
 inStrg = szTemp;
 element.num_rule_ants += inStrg; //append the assignment stmt to the string
 element.num_rule_ants = TRUE;    //mark it as complete
 if(element.Show!="0")
                    cout << "\n\tElement 'num_rule_ants' is COMPLETE!";

}
else if(inStrg=="num_rule_cons")
{ char szTemp[256];
 input >> inStrg;
 input.get(szTemp, 255);    //retrieve  data between { and }
 inStrg = szTemp;
 element.num_rule_cons += inStrg; //append the assignment stmt to the string
 element.num_rule_cons = TRUE;    //mark it as complete
```

```
            if(element.Show!="0")
                        cout << "\n\tElement 'num_rule_cons' is COMPLETE!";

        }
}//end if int
else if(inStrg=="float")  //check all data elements of type float
{
        input >> inStrg;      //get string containing the element name
        element.varName(inStrg);      //remove any extra chars from the variable name

        if(inStrg=="inmem_points")
        {
                char szTemp[2048];
                input >> inStrg;
                input.get(szTemp, 2047, ';'); //retrieve  data between { };'s
                inStrg = szTemp;
                inStrg += ";";
                element.inmem_points += inStrg; //append the assignment stmt to the string
                element.inmem_points = TRUE;    //mark it as complete
         if(element.Show!="0")
                        cout << "\n\tElement 'inmem_points' is COMPLETE!";

        }
        else if(inStrg=="outmem_points")
        {
                char szTemp[2048];
                input >> inStrg;
                input.get(szTemp, 2047, ';'); //retrieve  data between { };'s
                inStrg = szTemp;
                inStrg += ";";
                element.outmem_points += inStrg; //append the assignment stmt to the string
                element.outmem_points = TRUE;    //mark it as complete
          if(element.Show!="0")
                        cout << "\n\tElement 'outmem_points' is COMPLETE!";

        }
}//end else if float
else if(inStrg=="struct")   //check all data elementts of type struct
{
        input >> inStrg >> inStrg; //get next string containing the data elem name
        element.varName(inStrg);          //remove any extra chars from the variable name

        if(inStrg=="Rules")
        {
                char szTemp[2048];
                input >> inStrg;
                input.get(szTemp, 2047, ';'); //retrieve  data between { };'s
                inStrg = szTemp;
                inStrg += ";";
                element.Rules += inStrg; //append the assignment stmt to the string
                element.Rules = TRUE;    //mark it as complete
```

```
            if(element.Show!="0")
                                        cout << "\n\tElement 'Rules' is COMPLETE!";

                    }
            }//end else if struct

            return input;
}//end >>

#endif
```

APPENDIX D
XFUDGE STRING CLASS DEFINITIONS

```
/*-------------------------------------------------------------
PROGRAM  :string.hpp
AUTHOR   :Mark Workman
DATE     :8/21/96
VERSION  :1.0
DESCRIPTION:This is a string operations class. It can be very helpful when
working with strings. It takes advantage of both object oriented design and
dynamic memory allocation.

String functions include:
            -constructor/destructor
            -determine string length
            -return String char pointer
            -string character retrieval str[]
            -string assignment   str2 = str1;
            -clear string
            -string concatination  str1 + str2
~I/O operations:
            -overloaded >> (extractor) string retrieval from the iostream
            -overloaded << (inserter) string insertion into the iostream
~Logical operations:
            -equivalent  str1==str2
            -not equal   str1!=str2
            -less than   str1<str2
            -greater than  str1>str2
-------------------------------------------------------------*/
#include <iostream.h>
#ifndef _STRG
#define _STRG

class String
{  public:
            // Constructors
            String ( int numChars = 0 );          // Create an empty string
            String ( const char *charSeq );       // Initialize using char*
            // Destructor
            ~String ();

            // String operations
            int length () const;                   // # characters
           char *charStrg()const;                 //pointer to char array
            char &operator [] ( int n )const;              // Subscript
            String & operator = ( const String &rightString );  // Assignment
            void clear ();                         // Clear string
            String & operator += ( const String &rightString);   // cat
            int operator == ( String &rightString ) const;
            int operator == ( char * rightString ) const;
            int operator != ( char * rightString ) const;
            int operator <  ( String &rightString ) const;
            int operator >  ( String &rightString ) const;
 private:
            // Data members
            int bufferSize;  // Size of the string buffer
            char *buffer;    // String buffer containing a null-terminated sequence of characters

 // Friends
            friend istream & operator >> ( istream &input, String &inputString );
            friend ostream & operator << ( ostream &output, const String &outputString );
}; //end string class
#endif
```

APPENDIX E
XFUDGE STRING CLASS FUNCTIONS

```
/*-----------------------------------------------------------------
PROGRAM  :string.cpp
AUTHOR   :Mark Workman
DATE     :8/21/96
VERSION  :1.0
DESCRIPTION:This file contains the implementation of the string operations
class. It can be very helpful when working with strings. It takes advantage
of both object oriented design and dynamic memory allocation.

String functions include:
            -constructor/destructor
            -determine string length
            -return char pointer to string
            -string character retrieval str[]
            -string assignment   str2 = str1
            -string concatination  str1+str2
            -clear string

            ~I/O operations:
            -overloaded >> (extractor) string retrieval from the iostream
            -overloaded << (inserter) string insertion into the iostream

            ~Logical operations:
            -equivalent  str1==str2
             -not equal   str1!=str2
            -less than   str1<str2
            -greater than  str1>str2
-----------------------------------------------------------------*/
#ifndef _STRG_CPP
#define _STRG_CPP

//INCLUDES
#include "string.hpp"
#include <string.h>
#include <iostream.h>


String::String ( int numChars )          // Create an empty string
{
            bufferSize = numChars +1;
            buffer = new char [bufferSize];

            for(int i=0; i<=bufferSize; i++)
                        buffer[i] = '\0';
}//end constructor

String::String ( const char *charSeq )       // Initialize using char*
{
            bufferSize = (strlen(charSeq)+1);
            buffer = new char [bufferSize];
            strcpy(buffer, charSeq);
}//end initialize

// Destructor
String::~String ()
{
            delete []buffer;
}//end destructor
```

```
// String operations
int String::length () const                    // # characters
{
        return strlen(buffer);
}//end length

char *String::charStrg() const      //returns a constant pointer to the
{                                   //string in the buffer
        return buffer;
}//end StrgChar

char &String::operator [] (int n) const        // Subscript
{                                  //can be used on both the left
        if(n<bufferSize && n>=0)            //  and right side of the assignment
                return buffer[n];                                                    //  statement
        return buffer[0];
}//end operator[]

String & String::operator = ( const String &rightString )  // Assignment
{
        delete []buffer;
        bufferSize = rightString.bufferSize;
        buffer = new char [bufferSize];
        strcpy(buffer, rightString.buffer);
        return *this;
}//end operator=

void String::clear ()                          // Clear string
{
        buffer[0] = '\0';
}//end clear

String &String::operator+= (const String &rightString)
{
        char *Temp;

        Temp = new char[bufferSize];
        strcpy(Temp, buffer);

        bufferSize = bufferSize + rightString.bufferSize + 1;
        delete []buffer;

        buffer = new char [bufferSize];
        strcpy(buffer, Temp);
        strcat(buffer, rightString.buffer);
        delete []Temp;

        return *this;
}//end operator +

int String::operator == (String &rightString )const
{
        int i = strcmp(buffer, rightString.buffer);
        if(!i)
                return 1;
        return i;
}//end operator==
```

```
int String::operator == (char *rightString)const
{
        int i = strcmp(buffer, rightString);
        if(!i)
                return 1;
        return 0;
}//end operator==

int String::operator != ( char * rightString ) const
{
        int i = strcmp(buffer, rightString);
        if(!i)
                return 0;
        return 1;
}//end operator !=

int String::operator < (String &rightString )const
{
        if(strcmp(buffer, rightString.buffer) < 0)
                return 1;
        return 0;
}//end operator <

int String::operator > (String &rightString )const
{
        if(strcmp(buffer, rightString.buffer) > 0)
                return 1;
        return 0;
}//end operator >


//FRIENDS
istream &operator >> (istream &input, String &inputString)
{
        char szTemp[256];

        input >> szTemp;
        delete []inputString.buffer;

        inputString.bufferSize = strlen(szTemp)+1;
        inputString.buffer = new char[inputString.bufferSize];
        strcpy(inputString.buffer,szTemp);
        return input;
}// end >>

ostream &operator << (ostream &output, const String &outputString)
{
        output << outputString.buffer;
        return output;
}// end <<

#endif
```

APPENDIX F
XFUDGE HEADER FILE

```
/*-----------------------------------------------------------------------
PROGRAM: fuzzy.knb
AUTHOR:  Mark Workman
DATE:    8-10-96
VERSION: 1.0
DESCRIPTION: This file contains the fuzzy knowlege base constants for the
fuzzy class definition. These constants define the operation of your fuzzy engine.

Three new global constants have been added to indicate NUM_INPUTS, NUM_OUTPUTS
and NUM_RULES. Respectively, they indicate the number of crisp inputs,
crisp outputs and rules.

The original knowlege base was created by the Fuzzy Design Generator (FUDGE), a
fuzzy design and development tool. Fudge was created by the Motorola Corporation.
The FUDGE software was written by Alex DeCastro and Jason Spielman. The current
version of FUDGE is v.1.04

FILES:
The "fuzzy.hpp" file contains the (C++) fuzzy class definition.
The "fuzzy.cpp" file contains the fuzzy class member functions.
The "fuzzy.knb" file contains the unique knowledge base created by FUDGE for
your application.

NOTE: "fuzzy.knb" is the default filename created by the
xFUDGE translation program. If you changed the KNB filename during the
translation process, your filename will differ from this name.
-----------------------------------------------------------------------*/

//DEFINES
#ifndef _FUZZY_DEFS
#define _FUZZY_DEFS
#define TRACE    0    //set to display fuzzy parameters
#define NO_RULES 0    //set to display inputs when no rules fire
#define MIN(A,B)  (A < B) ? A : B
#define MAX(A,B)  (A > B) ? A : B
#endif
```

APPENDIX G
ANSI C IMPLEMENTATION OF WASHING MACHINE FUZZY ENGINE

```
/*
        Application name:  FUzzy Development and Generation Environment (FUDGE) Version V1.02
        File name:   Fuzzy.c
        Written by:          Alex DeCastro & Jason Spielman

        Copyright Motorola 1994
        SCALE    1
*/

#include <stdio.h>
#include "Fuzzy.h"

int     num_inputs = 2;
int     num_outputs = 1;
int     num_rules = 9;

int     num_input_mfs[2] = { 3, 3 };

struct  In      Inputs[] =
{
        { 0.000000, 100.000000 },
        { 0.000000, 100.000000 }
};

float   inmem_points[2][7][4] =
{
        {
                { 0.000000, 0.000000, 0.000000, 50.000000 },
                { 0.000000, 50.000000, 50.000000, 100.000000 },
                { 50.000000, 100.000000, 100.000000, 100.000000 }
        },
        {
                { 0.000000, 0.000000, 0.000000, 50.000000 },
                { 0.000000, 50.000000, 50.000000, 100.000000 },
                { 50.000000, 100.000000, 100.000000, 100.000000 }
        }
};
int     num_output_mfs[1] = { 5 };

struct  Out     Outputs[] =
{
        { 0.000000, 60.000000 }
};

float   outmem_points[1][7][4] =
{
        {
                { 8.000000 },
                { 12.000000 },
                { 20.000000 },
                { 45.000000 },
                { 60.000000 }
        }
};
float   crisp_outputs[1] = { 0};

int     num_rule_ants[9] = { 2, 2, 2, 2, 2, 2, 2, 2, 2};
int     num_rule_cons[9] = { 1, 1, 1, 1, 1, 1, 1, 1, 1};
```

```
struct	Rule	Rules[9] =
{
		{ { 0x10, 0x11 }, { 0xa0 } },
		{ { 0x08, 0x11 }, { 0x98 } },
		{ { 0x00, 0x11 }, { 0x98 } },
		{ { 0x10, 0x09 }, { 0x98 } },
		{ { 0x08, 0x09 }, { 0x90 } },
		{ { 0x00, 0x09 }, { 0x90 } },
		{ { 0x10, 0x01 }, { 0x90 } },
		{ { 0x08, 0x01 }, { 0x88 } },
		{ { 0x00, 0x01 }, { 0x80 } }
};

void fuzzy_step(float *crisp_inputs, float *crisp_outputs)
{
		int		in_index,rule_index,out_index;
		float	in_val;
		for (in_index = 0;in_index < num_inputs;in_index++)
		{
			fuzzify_input(in_index,crisp_inputs[in_index]);
		}
		for (rule_index = 0;rule_index < num_rules;rule_index++)
		{
			eval_rule(rule_index);
		}
		for (out_index = 0;out_index < num_outputs;out_index++)
		{
			crisp_outputs[out_index] = defuzzify_output(out_index, crisp_inputs);
			if (TRACE) printf("crisp_output[%d] = %f\n", out_index, crisp_outputs[out_index]);
		}
}
void fuzzify_input(int in_index,float in_val)
{
		int i;
		if (TRACE) printf("Fuzzify: input #%d crisp value %f\n", in_index, in_val);
		for (i = 0;i < num_input_mfs[in_index];i++)
		{
			fuzzy_inputs[in_index][i] = get_membership_value(in_index,i,in_val);
			if (TRACE) printf("Membership function #%d grade %f\n", i, fuzzy_inputs[in_index][i]);
		}
}
float get_membership_value(int in_index,int mf_index,float in_val)
{
		if (in_val < inmem_points[in_index][mf_index][0]) return 0;
		if (in_val > inmem_points[in_index][mf_index][3]) return 0;
		if (in_val <= inmem_points[in_index][mf_index][1])
		{
			if (inmem_points[in_index][mf_index][0] == inmem_points[in_index][mf_index][1])
				return 1;
			else
				return ((in_val - inmem_points[in_index][mf_index][0]) /
					(inmem_points[in_index][mf_index][1] -
inmem_points[in_index][mf_index][0]));
		}
		if (in_val >= inmem_points[in_index][mf_index][2])
		{		if (inmem_points[in_index][mf_index][2] == inmem_points[in_index][mf_index][3])
				return 1;
			else
				return ((inmem_points[in_index][mf_index][3] - in_val) /
					(inmem_points[in_index][mf_index][3] -
inmem_points[in_index][mf_index][2]));
		}
		return 1;
}
```

```
void eval_rule(int rule_index)
{
        int         in_index,out_index,mf_index,ant_index,con_index;
        int    val;
        float   rule_strength = 1;
        for    (ant_index = 0;ant_index < num_rule_ants[rule_index];ant_index++)
        {
                val = Rules[rule_index].antecedent[ant_index];
                in_index = (val & 0x07);
                mf_index = ((val & 0x38) >> 3);
                rule_strength = MIN(rule_strength,fuzzy_inputs[in_index][mf_index]);
        }
        rule_strengths[rule_index] = rule_strength;
        if (TRACE) printf("Rule  #%d strength %f\n", rule_index, rule_strength);
        for (con_index = 0;con_index < num_rule_cons[rule_index];con_index++)
        {
                val = Rules[rule_index].consequent[con_index];
                out_index = (val & 0x03);
                mf_index = ((val & 0x38) >> 3);
                fuzzy_outputs[out_index][mf_index] = MAX(fuzzy_outputs[out_index][mf_index],
                        rule_strengths[rule_index]);
        }
}
float defuzzify_output(int out_index,float *inputs)
{
        float       summ = 0;
        float       product = 0;
        float       temp1,temp2;
        int         mf_index,in_index;
        if (TRACE) printf("Defuzzify: output #%d\n", out_index);
        for (mf_index = 0;mf_index < num_output_mfs[out_index];mf_index++)
        {
                temp1 = fuzzy_outputs[out_index][mf_index];
                temp2 = outmem_points[out_index][mf_index][0];
                summ = summ + temp1;
                product = product + (temp1 * temp2);
                if (TRACE) printf("Membership function #%d grade %f\n", mf_index, fuzzy_outputs[out_index][mf_index]);
                fuzzy_outputs[out_index][mf_index] = 0;
        }
        if (summ > 0)
        {
                crisp_outputs[out_index] = product / summ;
                return crisp_outputs[out_index];
        }
        else
        {
                if (NO_RULES){
                printf("No rules fire for:\n");
                        for (in_index = 0;in_index < num_inputs;in_index++)
                           printf("Input #%d=%f ", in_index,inputs[in_index]);
                           printf("\n");}
                return crisp_outputs[out_index];
        }
}
```

APPENDIX H
ANSI C IMPLEMENTATION OF TRAFFIC LIGHT EXAMPLE OF FUZZY ENGINE

```
/*
        Application name:  FUzzy Development and Generation Environment (FUDGE) Version V1.02
        File name:  Fuzzy.c
        Written by:             Alex DeCastro & Jason Spielman

        Copyright Motorola 1994
        SCALE   1
*/

#include <stdio.h>
#include "Fuzzy.h"

int     num_inputs = 3;
int     num_outputs = 1;
int     num_rules = 26;

int     num_input_mfs[3] = { 4, 4. 3 };

struct  In      Inputs[] =
{
        { 0.000000, 12.000000 },
        { 0.000000, 12.000000 },
        { 0.000000, 110.000000 }
};

float   inmem_points[3][7][4] =
{
        {
                { 0.000000, 0.000000, 0.000000, 1.000000 },
                { 0.000000, 1.000000, 2.000000, 3.000000 },
                { 2.000000, 3.000000, 3.000000, 4.000000 },
                { 3.000000, 4.000000, 12.000000, 12.000000 }
        },
        {
                { 0.000000, 0.000000, 0.000000, 1.000000 },
                { 0.000000, 1.000000, 3.000000, 6.000000 },
                { 3.000000, 6.000000, 6.000000, 9.000000 },
                { 6.000000, 9.000000, 12.000000, 12.000000 }
        },
        {
                { 0.000000, 0.000000, 30.000000, 60.000000 },
                { 30.000000, 60.000000, 60.000000, 90.000000 },
                { 60.000000, 90.000000, 110.000000, 110.000000 }
        }
};
int     num_output_mfs[1] = { 5 };

struct  Out     Outputs[] =
{
        { 0.000000, 100.000000 }
};
```

```
float	outmem_points[1][7][4] =
{
		{
			{ 0.000000 },
			{ 30.000000 },
			{ 50.000000 },
			{ 70.000000 },
			{ 100.000000 }
		}
};
float	crisp_outputs[1] = { 0};

int	num_rule_ants[26] = { 2, 2, 2, 2, 1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3};
int	num_rule_cons[26] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

struct	Rule	Rules[26] =
{
		{ { 0x00, 0x01 }, { 0x80 } },
		{ { 0x00, 0x09 }, { 0xa0 } },
		{ { 0x00, 0x11 }, { 0xa0 } },
		{ { 0x00, 0x19 }, { 0xa0 } },
		{ { 0x01 }, { 0x80 } },
		{ { 0x08, 0x09 }, { 0x80 } },
		{ { 0x10, 0x11 }, { 0x80 } },
		{ { 0x18, 0x19 }, { 0x80 } },
		{ { 0x08, 0x11, 0x02 }, { 0x90 } },
		{ { 0x08, 0x11, 0x0a }, { 0x98 } },
		{ { 0x08, 0x11, 0x12 }, { 0xa0 } },
		{ { 0x08, 0x19, 0x02 }, { 0x88 } },
		{ { 0x08, 0x19, 0x0a }, { 0x90 } },
		{ { 0x08, 0x19, 0x12 }, { 0x98 } },
		{ { 0x10, 0x09, 0x02 }, { 0x88 } },
		{ { 0x10, 0x09, 0x0a }, { 0x88 } },
		{ { 0x10, 0x09, 0x12 }, { 0x90 } },
		{ { 0x10, 0x19, 0x02 }, { 0x90 } },
		{ { 0x10, 0x19, 0x0a }, { 0x98 } },
		{ { 0x10, 0x19, 0x12 }, { 0xa0 } },
		{ { 0x18, 0x09, 0x02 }, { 0x90 } },
		{ { 0x18, 0x09, 0x0a }, { 0x98 } },
		{ { 0x18, 0x09, 0x12 }, { 0xa0 } },
		{ { 0x18, 0x11, 0x02 }, { 0x88 } },
		{ { 0x18, 0x11, 0x0a }, { 0x88 } },
		{ { 0x18, 0x11, 0x12 }, { 0x90 } }
};

void fuzzy_step(float *crisp_inputs, float *crisp_outputs)
{
		int		in_index,rule_index,out_index;
		float	in_val;
		for (in_index = 0;in_index < num_inputs;in_index++)
		{
			fuzzify_input(in_index,crisp_inputs[in_index]);
		}
		for (rule_index = 0;rule_index < num_rules;rule_index++)
		{
			eval_rule(rule_index);
		}
		for (out_index = 0;out_index < num_outputs;out_index++)
		{
			crisp_outputs[out_index] = defuzzify_output(out_index, crisp_inputs);
			if (TRACE) printf("crisp_output[%d] = %f\n", out_index, crisp_outputs[out_index]);
		}
}
```

```
void fuzzify_input(int in_index,float in_val)
{
        int i;
        if (TRACE) printf("Fuzzify: input #%d crisp value %f\n", in_index, in_val);
        for (i = 0;i < num_input_mfs[in_index];i++)
        {
                fuzzy_inputs[in_index][i] = get_membership_value(in_index,i,in_val);
                if (TRACE) printf("Membership function #%d grade %f\n", i, fuzzy_inputs[in_index][i]);
        }
}
float get_membership_value(int in_index,int mf_index,float in_val)
{
        if (in_val < inmem_points[in_index][mf_index][0]) return 0;
        if (in_val > inmem_points[in_index][mf_index][3]) return 0;
        if (in_val <= inmem_points[in_index][mf_index][1])
        {
                if (inmem_points[in_index][mf_index][0] == inmem_points[in_index][mf_index][1])
                        return 1;
                else
                        return ((in_val - inmem_points[in_index][mf_index][0]) /
                                        (inmem_points[in_index][mf_index][1] -
inmem_points[in_index][mf_index][0]));
        }
        if (in_val >= inmem_points[in_index][mf_index][2])
        {
                if (inmem_points[in_index][mf_index][2] == inmem_points[in_index][mf_index][3])
                        return 1;
                else
                        return ((inmem_points[in_index][mf_index][3] - in_val) /
                                        (inmem_points[in_index][mf_index][3] -
inmem_points[in_index][mf_index][2]));
        }
        return 1;
}
void eval_rule(int rule_index)
{       int     in_index,out_index,mf_index,ant_index,con_index;
        int   val;
        float  rule_strength = 1;
        for   (ant_index = 0;ant_index < num_rule_ants[rule_index];ant_index++)
        {
                val = Rules[rule_index].antecedent[ant_index];
                in_index = (val & 0x07);
                mf_index = ((val & 0x38) >> 3);
                rule_strength = MIN(rule_strength,fuzzy_inputs[in_index][mf_index]);
        }
        rule_strengths[rule_index] = rule_strength;
        if (TRACE) printf("Rule  #%d strength %f\n", rule_index, rule_strength);
        for (con_index = 0;con_index < num_rule_cons[rule_index];con_index++)
        {
                val = Rules[rule_index].consequent[con_index];
                out_index = (val & 0x03);
                mf_index = ((val & 0x38) >> 3);
                fuzzy_outputs[out_index][mf_index] = MAX(fuzzy_outputs[out_index][mf_index],
                        rule_strengths[rule_index]);
        }
}
float defuzzify_output(int out_index,float *inputs)
{       float       summ = 0;
        float       product = 0;
        float       temp1,temp2;
        int         mf_index,in_index;
        if (TRACE) printf("Defuzzify: output #%d\n", out_index);
        for (mf_index = 0;mf_index < num_output_mfs[out_index];mf_index++)
        {
```

```
                temp1 = fuzzy_outputs[out_index][mf_index];
                temp2 = outmem_points[out_index][mf_index][0];
                summ = summ + temp1;
                product = product + (temp1 * temp2);
                if (TRACE) printf("Membership function #%d grade %f\n", mf_index, fuzzy_outputs[out_index][mf_index]);
                fuzzy_outputs[out_index][mf_index] = 0;
        }
        if (summ > 0)
        {
                crisp_outputs[out_index] = product / summ;
                return crisp_outputs[out_index];
        }
        else    {
                if (NO_RULES){
                printf("No rules fire for:\n");
                        for (in_index = 0;in_index < num_inputs;in_index++)
                           printf("Input #%d=%f ", in_index,inputs[in_index]);
                           printf("\n");}
                return crisp_outputs[out_index];
        }
}
```

APPENDIX I
ANSI C VERSION OF TRUCK BACKER-UPPER EXAMPLE FUZZY ENGINE

```
/*
        Application name:  FUzzy Development and Generation Environment (FUDGE) Version V1.02
        File name:  Fuzzy.c
        Written by:             Alex DeCastro & Jason Spielman

        Copyright Motorola 1994
        SCALE   1
*/

#include <stdio.h>
#include "Fuzzy.h"

int     num_inputs = 2;
int     num_outputs = 1;
int     num_rules = 35;

int     num_input_mfs[2] = { 7, 5 };

struct  In      Inputs[] =
{
        { -90.000000, 270.000000 },
        { 0.000000, 100.000000 }
};

float   inmem_points[2][7][4] =
{
        {
                { -90.000000, -45.000000, -45.000000, 8.000000 },
                { -12.000000, 24.000000, 24.000000, 65.000000 },
                { 45.000000, 65.000000, 65.000000, 90.000000 },
                { 80.000000, 90.000000, 90.000000, 100.000000 },
                { 90.000000, 112.000000, 112.000000, 134.000000 },
                { 115.000000, 155.000000, 155.000000, 195.000000 },
                { 175.000000, 225.000000, 225.000000, 270.000000 }
        },
        {
                { 0.000000, 0.000000, 12.000000, 35.000000 },
                { 30.000000, 40.000000, 40.000000, 50.000000 },
                { 45.000000, 50.000000, 50.000000, 55.000000 },
                { 50.000000, 60.000000, 60.000000, 70.000000 },
                { 65.000000, 88.000000, 100.000000, 100.000000 }
        }
};
int     num_output_mfs[1] = { 7 };

struct  Out     Outputs[] =
{
        { -30.000000, 30.000000 }
};
```

```
float       outmem_points[1][7][4] =
{
            {
                        { -30.000000 },
                        { -15.000000 },
                        { -5.000000 },
                        { 0.000000 },
                        { 5.000000 },
                        { 15.000000 },
                        { 30.000000 }
            }
};
float       crisp_outputs[1] = { 0};

int         num_rule_ants[35] = { 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2};
int         num_rule_cons[35] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

struct      Rule        Rules[35] =
{
            { { 0x00, 0x01 }, { 0xa0 } },
            { { 0x08, 0x01 }, { 0x90 } },
            { { 0x10, 0x01 }, { 0x88 } },
            { { 0x18, 0x01 }, { 0x88 } },
            { { 0x20, 0x01, 0x01 }, { 0x80 } },
            { { 0x28, 0x01 }, { 0x80 } },
            { { 0x30, 0x01 }, { 0x80 } },
            { { 0x00, 0x09 }, { 0xa8 } },
            { { 0x08, 0x09 }, { 0xa0 } },
            { { 0x10, 0x09 }, { 0x90 } },
            { { 0x18, 0x09 }, { 0x88 } },
            { { 0x20, 0x09 }, { 0x88 } },
            { { 0x28, 0x09 }, { 0x80 } },
            { { 0x30, 0x09 }, { 0x80 } },
            { { 0x00, 0x11 }, { 0xa8 } },
            { { 0x08, 0x11 }, { 0xa8 } },
            { { 0x10, 0x11 }, { 0xa0 } },
            { { 0x18, 0x11 }, { 0x98 } },
            { { 0x20, 0x11 }, { 0x90 } },
            { { 0x28, 0x11 }, { 0x88 } },
            { { 0x30, 0x11 }, { 0x88 } },
            { { 0x00, 0x19 }, { 0xb0 } },
            { { 0x08, 0x19 }, { 0xb0 } },
            { { 0x10, 0x19 }, { 0xa8 } },
            { { 0x18, 0x19 }, { 0xa8 } },
            { { 0x20, 0x19 }, { 0xa0 } },
            { { 0x28, 0x19 }, { 0x90 } },
            { { 0x30, 0x19 }, { 0x88 } },
            { { 0x00, 0x21 }, { 0xb0 } },
            { { 0x08, 0x21 }, { 0xb0 } },
            { { 0x10, 0x21 }, { 0xb0 } },
            { { 0x18, 0x21 }, { 0xa8 } },
            { { 0x20, 0x21 }, { 0xa8 } },
            { { 0x28, 0x21 }, { 0xa0 } },
            { { 0x30, 0x21 }, { 0x90 } }
};
```

```
void fuzzy_step(float *crisp_inputs, float *crisp_outputs)
{
        int         in_index,rule_index,out_index;
        float   in_val;
        for (in_index = 0;in_index < num_inputs;in_index++)
        {
                fuzzify_input(in_index,crisp_inputs[in_index]);
        }
        for (rule_index = 0;rule_index < num_rules;rule_index++)
        {
                eval_rule(rule_index);
        }
        for (out_index = 0;out_index < num_outputs;out_index++)
        {
                crisp_outputs[out_index] = defuzzify_output(out_index, crisp_inputs);
                if (TRACE) printf("crisp_output[%d] = %f\n", out_index, crisp_outputs[out_index]);
        }
}
void fuzzify_input(int in_index,float in_val)
{
        int i;
        if (TRACE) printf("Fuzzify: input #%d crisp value %f\n", in_index, in_val);
        for (i = 0;i < num_input_mfs[in_index];i++)
        {
                fuzzy_inputs[in_index][i] = get_membership_value(in_index,i,in_val);
                if (TRACE) printf("Membership function #%d grade %f\n", i, fuzzy_inputs[in_index][i]);
        }
}
float get_membership_value(int in_index,int mf_index,float in_val)
{
        if (in_val < inmem_points[in_index][mf_index][0]) return 0;
        if (in_val > inmem_points[in_index][mf_index][3]) return 0;
        if (in_val <= inmem_points[in_index][mf_index][1])      {
                if (inmem_points[in_index][mf_index][0] == inmem_points[in_index][mf_index][1])
                        return 1;
                else
                        return ((in_val - inmem_points[in_index][mf_index][0]) /
                                                (inmem_points[in_index][mf_index][1] -
inmem_points[in_index][mf_index][0]));
        }
        if (in_val >= inmem_points[in_index][mf_index][2])      {
                if (inmem_points[in_index][mf_index][2] == inmem_points[in_index][mf_index][3])
                        return 1;
                else
                        return ((inmem_points[in_index][mf_index][3] - in_val) /
                                                (inmem_points[in_index][mf_index][3] -
inmem_points[in_index][mf_index][2]));
        }
        return 1;
}
void eval_rule(int rule_index){
        int         in_index,out_index,mf_index,ant_index,con_index;
        int    val;
        float   rule_strength = 1;
        for    (ant_index = 0;ant_index < num_rule_ants[rule_index];ant_index++)       {
                val = Rules[rule_index].antecedent[ant_index];
                in_index = (val & 0x07);
                mf_index = ((val & 0x38) >> 3);
                rule_strength = MIN(rule_strength,fuzzy_inputs[in_index][mf_index]);
        }
        rule_strengths[rule_index] = rule_strength;
        if (TRACE) printf("Rule  #%d strength %f\n", rule_index, rule_strength);
        for (con_index = 0;con_index < num_rule_cons[rule_index];con_index++)       {
                val = Rules[rule_index].consequent[con_index];
```

```
                out_index = (val & 0x03);
                mf_index = ((val & 0x38) >> 3);
                fuzzy_outputs[out_index][mf_index] = MAX(fuzzy_outputs[out_index][mf_index],
                        rule_strengths[rule_index]);
        }
}
float defuzzify_output(int out_index,float *inputs){
        float   summ = 0;
        float   product = 0;
        float   temp1,temp2;
        int     mf_index,in_index;
        if (TRACE) printf("Defuzzify: output #%d\n", out_index);
        for (mf_index = 0;mf_index < num_output_mfs[out_index];mf_index++)      {
                temp1 = fuzzy_outputs[out_index][mf_index];
                temp2 = outmem_points[out_index][mf_index][0];
                summ = summ + temp1;
                product = product + (temp1 * temp2);
                if (TRACE) printf("Membership function #%d grade %f\n", mf_index, fuzzy_outputs[out_index][mf_index]);
                fuzzy_outputs[out_index][mf_index] = 0;
        }
        if (summ > 0)           {
                crisp_outputs[out_index] = product / summ;
                return crisp_outputs[out_index];
        }
        else    {
                if (NO_RULES){
                printf("No rules fire for:\n");
                        for (in_index = 0;in_index < num_inputs;in_index++)
                           printf("Input #%d=%f ", in_index,inputs[in_index]);
                           printf("\n");}
                return crisp_outputs[out_index];
        }
}
```

APPENDIX J
ANSI C VERSION OF FUZZY.H HEADER FILE

```
/*
        Application name:     FUzzy Development and Generation Environment (FUDGE) Version V1.00
        File name:              Fuzzy.h
        Written by:             Alex DeCastro & Jason Spielman

        Copyright Motorola 1994
*/

#define TRACE       1     /*set to display fuzzy parameters */
#define NO_RULES    1     /*set to display inputs when no rules fire*/
#define MIN(A,B)    (A < B) ? A : B
#define MAX(A,B)    (A > B) ? A   B

struct In {
        float       min;
        float       max;
};
struct Out {
        float       min;
        float       max;
};
struct Rule {
        int         antecedent[8];
        int         consequent[8];
};

float       fuzzy_inputs[8][8];
float       fuzzy_outputs[4][8];
float       rule_strengths[64];

void   fuzzy_step(float *, float *);
void   fuzzify_input(int, float);
float  get_membership_value(int, int, float);
void   eval_rule(int);
float  defuzzify_output(int, float *);
```

APPENDIX K
C++ VERSION OF WASHING MACHINE KNOWLEDGE BASE FILE

```
/*--------------------------------------------------------------------------
PROGRAM: fuzzy.knb
AUTHOR:  Mark Workman
DATE:    8-10-96
VERSION: 1.0
DESCRIPTION: This file contains the fuzzy knowlege base constants for the
fuzzy class definition. These constants define the operation of your fuzzy engine.

Three new global constants have been added to indicate NUM_INPUTS, NUM_OUTPUTS
and NUM_RULES. Respectively, they indicate the number of crisp inputs,
crisp outputs and rules.

The original knowlege base was created by the Fuzzy Design Generator (FUDGE), a
fuzzy design and development tool. Fudge was created by the Motorola Corporation.
The FUDGE software was written by Alex DeCastro and Jason Spielman. The current
version of FUDGE is v.1.02

FILES:
The "fuzzy.hpp" file contains the (C++) fuzzy class definition.
The "fuzzy.cpp" file contains the fuzzy class member functions.
The "fuzzy.knb" file contains the unique knowledge base created by FUDGE for
your application.

NOTE: "fuzzy.knb" is the default filename created by the
xFUDGE translation program. If you changed the KNB filename during the
translation process, your filename will differ from this name.
--------------------------------------------------------------------------*/

//DEFINES
#ifndef _FUZZY_DEFS
#define _FUZZY_DEFS
#define TRACE    0    //set to display fuzzy parameters
#define NO_RULES 0    //set to display inputs when no rules fire
#define MIN(A,B)  (A < B) ? A : B
#define MAX(A,B)  (A > B) ? A : B
#endif


#ifndef _FUZZY_KNB
#define _FUZZY_KNB
#define NUM_INPUTS 2
#define NUM_OUTPUTS 1
#define NUM_RULES 9
```

```
//STRUCTURES
struct Rule
{
        int   antecedent[8];
        int   consequent[8];
};


//CONSTANTS
const struct Rule Rules[NUM_RULES] =
{
        { { 0x10, 0x11 }, { 0xa0 } },
        { { 0x08, 0x11 }, { 0x98 } },
        { { 0x00, 0x11 }, { 0x98 } },
        { { 0x10, 0x09 }, { 0x98 } },
        { { 0x08, 0x09 }, { 0x90 } },
        { { 0x00, 0x09 }, { 0x90 } },
        { { 0x10, 0x01 }, { 0x90 } },
        { { 0x08, 0x01 }, { 0x88 } },
        { { 0x00, 0x01 }, { 0x80 } }
};
const int num_input_mfs[NUM_INPUTS] =  { 3, 3 };
const int num_output_mfs[NUM_OUTPUTS] =  { 5 };
const int num_rule_ants[NUM_RULES] =  { 2, 2, 2, 2, 2, 2, 2, 2, 2};
const int num_rule_cons[NUM_RULES] =  { 1, 1, 1, 1, 1, 1, 1, 1, 1};
const int inmem_points[NUM_INPUTS][7][4] =
{
        {
                { 0.000000, 0.000000, 0.000000, 50.000000 },
                { 0.000000, 50.000000, 50.000000, 100.000000 },
                { 50.000000, 100.000000, 100.000000, 100.000000 }
        },
        {
                { 0.000000, 0.000000, 0.000000, 50.000000 },
                { 0.000000, 50.000000, 50.000000, 100.000000 },
                { 50.000000, 100.000000, 100.000000, 100.000000 }
        }
};
const int outmem_points[NUM_OUTPUTS][7][4] =
{
        {
                { 8.000000 },
                { 12.000000 },
                { 20.000000 },
                { 45.000000 },
                { 60.000000 }
        }
};
#endif
```

APPENDIX L
C++ VERSION OF TRAFFIC LIGHT KNOWLEDGE BASE FILE

```
/*----------------------------------------------------------------------
PROGRAM: fuzzy.knb
AUTHOR:  Mark Workman
DATE:    8-10-96
VERSION: 1.0
DESCRIPTION: This file contains the fuzzy knowlege base constants for the
fuzzy class definition. These constants define the operation of your fuzzy engine.

Three new global constants have been added to indicate NUM_INPUTS, NUM_OUTPUTS
and NUM_RULES. Respectively, they indicate the number of crisp inputs,
crisp outputs and rules.

The original knowlege base was created by the Fuzzy Design Generator (FUDGE), a
fuzzy design and development tool. Fudge was created by the Motorola Corporation.
The FUDGE software was written by Alex DeCastro and Jason Spielman. The current
version of FUDGE is v.1.04

FILES:
The "fuzzy.hpp" file contains the (C++) fuzzy class definition.
The "fuzzy.cpp" file contains the fuzzy class member functions.
The "fuzzy.knb" file contains the unique knowledge base created by FUDGE for
your application.

NOTE: "fuzzy.knb" is the default filename created by the
xFUDGE translation program. If you changed the KNB filename during the
translation process, your filename will differ from this name.
----------------------------------------------------------------------*/

//DEFINES
#ifndef _FUZZY_DEFS
#define _FUZZY_DEFS
#define TRACE    0    //set to display fuzzy parameters
#define NO_RULES 0    //set to display inputs when no rules fire
#define MIN(A,B)  (A < B) ? A : B
#define MAX(A,B)  (A > B) ? A . B
#endif


#ifndef _FUZZY_KNB
#define _FUZZY_KNB
#define NUM_INPUTS 3
#define NUM_OUTPUTS 1
#define NUM_RULES 2
```

```
//STRUCTURES
struct Rule
{
        int   antecedent[8];
        int   consequent[8];
};


//CONSTANTS
const struct Rule Rules[NUM_RULES] =
{
        { { 0x00, 0x01 }, { 0x80 } },
        { { 0x00, 0x09 }, { 0xa0 } },
        { { 0x00, 0x11 }, { 0xa0 } },
        { { 0x00, 0x19 }, { 0xa0 } },
        { { 0x01 }, { 0x80 } },
        { { 0x08, 0x09 }, { 0x80 } },
        { { 0x10, 0x11 }, { 0x80 } },
        { { 0x18, 0x19 }, { 0x80 } },
        { { 0x08, 0x11, 0x02 }, { 0x90 } },
        { { 0x08, 0x11, 0x0a }, { 0x98 } },
        { { 0x08, 0x11, 0x12 }, { 0xa0 } },
        { { 0x08, 0x19, 0x02 }, { 0x88 } },
        { { 0x08, 0x19, 0x0a }, { 0x90 } },
        { { 0x08, 0x19, 0x12 }, { 0x98 } },
        { { 0x10, 0x09, 0x02 }, { 0x88 } },
        { { 0x10, 0x09, 0x0a }, { 0x88 } },
        { { 0x10, 0x09, 0x12 }, { 0x90 } },
        { { 0x10, 0x19, 0x02 }, { 0x90 } },
        { { 0x10, 0x19, 0x0a }, { 0x98 } },
        { { 0x10, 0x19, 0x12 }, { 0xa0 } },
        { { 0x18, 0x09, 0x02 }, { 0x90 } },
        { { 0x18, 0x09, 0x0a }, { 0x98 } },
        { { 0x18, 0x09, 0x12 }, { 0xa0 } },
        { { 0x18, 0x11, 0x02 }, { 0x88 } },
        { { 0x18, 0x11, 0x0a }, { 0x88 } },
        { { 0x18, 0x11, 0x12 }, { 0x90 } }
};
const int num_input_mfs[NUM_INPUTS] =  { 4, 4, 3 };
const int num_output_mfs[NUM_OUTPUTS] =  { 5 };
const int num_rule_ants[NUM_RULES] =  { 2, 2, 2, 2, 1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3};
const int num_rule_cons[NUM_RULES] =  { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1};
```

```
const int inmem_points[NUM_INPUTS][7][4] =
{
        {
                { 0.000000, 0.000000, 0.000000, 1.000000 },
                { 0.000000, 1.000000, 2.000000, 3.000000 },
                { 2.000000, 3.000000, 3.000000, 4.000000 },
                { 3.000000, 4.000000, 12.000000, 12.000000 }
        },
        {
                { 0.000000, 0.000000, 0.000000, 1.000000 },
                { 0.000000, 1.000000, 3.000000, 6.000000 },
                { 3.000000, 6.000000, 6.000000, 9.000000 },
                { 6.000000, 9.000000, 12.000000, 12.000000 }
        },
        {
                { 0.000000, 0.000000, 30.000000, 60.000000 },
                { 30.000000, 60.000000, 60.000000, 90.000000 },
                { 60.000000, 90.000000, 110.000000, 110.000000 }
        }
};
const int outmem_points[NUM_OUTPUTS][7][4] =
{
        {
                { 0.000000 },
                { 30.000000 },
                { 50.000000 },
                { 70.000000 },
                { 100.000000 }
        }
};
#endif
```

APPENDIX M
C++ VERSION OF TRUCK BACKER-UPPER KNOWLEDGE BASE FILE

```
/*-----------------------------------------------------------------------
PROGRAM: fuzzy.knb
AUTHOR:  Mark Workman
DATE:    8-10-96
VERSION: 1.0
DESCRIPTION: This file contains the fuzzy knowlege base constants for the
fuzzy class definition. These constants define the operation of your fuzzy engine.

Three new global constants have been added to indicate NUM_INPUTS, NUM_OUTPUTS
and NUM_RULES. Respectively, they indicate the number of crisp inputs,
crisp outputs and rules.

The original knowlege base was created by the Fuzzy Design Generator (FUDGE), a
fuzzy design and development tool. Fudge was created by the Motorola Corporation.
The FUDGE software was written by Alex DeCastro and Jason Spielman. The current
version of FUDGE is v.1.04

FILES:
The "fuzzy.hpp" file contains the (C++) fuzzy class definition.
The "fuzzy.cpp" file contains the fuzzy class member functions.
The "fuzzy.knb" file contains the unique knowledge base created by FUDGE for
your application.

NOTE: "fuzzy.knb" is the default filename created by the
xFUDGE translation program. If you changed the KNB filename during the
translation process, your filename will differ from this name.
-----------------------------------------------------------------------*/

//DEFINES
#ifndef _FUZZY_DEFS
#define _FUZZY_DEFS
#define TRACE     0     //set to display fuzzy parameters
#define NO_RULES  0     //set to display inputs when no rules fire
#define MIN(A,B)  (A < B) ? A : B
#define MAX(A,B)  (A > B) ? A   B
#endif


#ifndef _FUZZY_KNB
#define _FUZZY_KNB
#define NUM_INPUTS 2
#define NUM_OUTPUTS 1
#define NUM_RULES 35
```

```
//STRUCTURES
struct Rule
{
        int   antecedent[8];
        int   consequent[8];
};


//CONSTANTS
const struct Rule Rules[NUM_RULES] =
{
        { { 0x00, 0x01 }, { 0xa0 } },
        { { 0x08, 0x01 }, { 0x90 } },
        { { 0x10, 0x01 }, { 0x88 } },
        { { 0x18, 0x01 }, { 0x88 } },
        { { 0x20, 0x01, 0x01 }, { 0x80 } },
        { { 0x28, 0x01 }, { 0x80 } },
        { { 0x30, 0x01 }, { 0x80 } },
        { { 0x00, 0x09 }, { 0xa8 } },
        { { 0x08, 0x09 }, { 0xa0 } },
        { { 0x10, 0x09 }, { 0x90 } },
        { { 0x18, 0x09 }, { 0x88 } },
        { { 0x20, 0x09 }, { 0x88 } },
        { { 0x28, 0x09 }, { 0x80 } },
        { { 0x30, 0x09 }, { 0x80 } },
        { { 0x00, 0x11 }, { 0xa8 } },
        { { 0x08, 0x11 }, { 0xa8 } },
        { { 0x10, 0x11 }, { 0xa0 } },
        { { 0x18, 0x11 }, { 0x98 } },
        { { 0x20, 0x11 }, { 0x90 } },
        { { 0x28, 0x11 }, { 0x88 } },
        { { 0x30, 0x11 }, { 0x88 } },
        { { 0x00, 0x19 }, { 0xb0 } },
        { { 0x08, 0x19 }, { 0xb0 } },
        { { 0x10, 0x19 }, { 0xa8 } },
        { { 0x18, 0x19 }, { 0xa8 } },
        { { 0x20, 0x19 }, { 0xa0 } },
        { { 0x28, 0x19 }, { 0x90 } },
        { { 0x30, 0x19 }, { 0x88 } },
        { { 0x00, 0x21 }, { 0xb0 } },
        { { 0x08, 0x21 }, { 0xb0 } },
        { { 0x10, 0x21 }, { 0xb0 } },
        { { 0x18, 0x21 }, { 0xa8 } },
        { { 0x20, 0x21 }, { 0xa8 } },
        { { 0x28, 0x21 }, { 0xa0 } },
        { { 0x30, 0x21 }, { 0x90 } }
};
const int num_input_mfs[NUM_INPUTS] =  { 7, 5 };
const int num_output_mfs[NUM_OUTPUTS] =  { 7 };
const int num_rule_ants[NUM_RULES] =  { 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2};
const int num_rule_cons[NUM_RULES] =  { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1};
```

```
const int inmem_points[NUM_INPUTS][7][4] =
{
        {
                { -90.000000, -45.000000, -45.000000, 8.000000 },
                { -12.000000, 24.000000, 24.000000, 65.000000 },
                { 45.000000, 65.000000, 65.000000, 90.000000 },
                { 80.000000, 90.000000, 90.000000, 100.000000 },
                { 90.000000, 112.000000, 112.000000, 134.000000 },
                { 115.000000, 155.000000, 155.000000, 195.000000 },
                { 175.000000, 225.000000, 225.000000, 270.000000 }
        },
        {
                { 0.000000, 0.000000, 12.000000, 35.000000 },
                { 30.000000, 40.000000, 40.000000, 50.000000 },
                { 45.000000, 50.000000, 50.000000, 55.000000 },
                { 50.000000, 60.000000, 60.000000, 70.000000 },
                { 65.000000, 88.000000, 100.000000, 100.000000 }
        }
};
const int outmem_points[NUM_OUTPUTS][7][4] =
{
        {
                { -30.000000 },
                { -15.000000 },
                { -5.000000 },
                { 0.000000 },
                { 5.000000 },
                { 15.000000 },
                { 30.000000 }
        }
};
#endif
```

APPENDIX N
C++ FUZZY ENGINE CLASS DEFINITION

```
/*--------------------------------------------------------------------------
PROGRAM: fuzzy.hpp
AUTHOR: Mark Workman
DATE: 7-10-96
VERSION: 1.0
DESCRIPTION: This file contains the fuzzy class definition. It implements the fuzzy
engine originally created by the Motorola FUDGE fuzzy development tool. FUDGE was written
by Alex DeCastro and Jason Spielman. There may be unspecified changes to these functions
and/or data structures. However, the basic structure is still intact. The current version
of FUDGE is v.1.04.

Version 1.0
The crisp_inputs and crisp_outputs data element arrays are now encapsulated within the
fuzzy class. The operator[] has been overloaded to facilitate these two arrays. To access
values in the crisp_inputs/crisp_outputs arrays; just use the corresponding input/output
number as the index. All crisp inputs are indexed by positive integer values greater than
zero. All crisp outputs are indexed by negative integer values less than zero.

        Valid crisp_input values will be integers 1,2,3, ...,N. Where N is last crisp input.
        Example to change crisp_input #2:    fuzzy_engine[2] = 3.4567;

    Valid crisp_output values will be integers -1,-2,-3, ...,-M. Where M is last crisp input.
    Example to retrieve crisp_output #2:    fValue = fuzzy_engine[-2];

Three new global constants have been added to indicate NUM_INPUTS, NUM_OUTPUTS and
NUM_RULES
Respectively, They indicate the number of crisp inputs, crisp outputs and rules.

Other structures and/or arrays are now constant or have been initialized by the constructor
to zero.

FILES:
The "fuzzy.hpp" file contains the (C++) fuzzy class definition.
The "fuzzy.cpp" file contains the fuzzy class member functions.
The "fuzzy.knb" file contains the unique knowledge base created by FUDGE for
your application.

NOTE: "fuzzy.knb" is the default filename created by the xFUDGE translation
program. If you changed the KNB filename during the translation process, your filename
will differ from this name.
--------------------------------------------------------------------------*/
#ifndef _FUZZY_HPP
#define _FUZZY_HPP

//INCLUDE FUZZY ENGINE KNOWLEGE BASE
#include"fuzzy.knb"    //Change to match your KNB filename.
```

```
//FUZZY CLASS DECLARATION
class fuzzy
{
 public:
        fuzzy();
        void   fuzzy_step();              //constructor
        float &operator[](int);           //overload[] operator

 private:
        //KNOWLEDGE BASE DATA VALUES
        int    num_inputs;
        int    num_outputs;
        int    num_rules;

        //FUZZY VARIABLES
        float  crisp_inputs[NUM_INPUTS];
        float  crisp_outputs[NUM_OUTPUTS];
        float  fuzzy_inputs[8][8];
        float  fuzzy_outputs[4][8];

         //FUNCTIONS
        void   fuzzify_input(int, float);
        float  get_membership_value(int, int, float);
        void   eval_rule(int);
        float  defuzzify_output(int, float *);
}; //end fuzzy class

#endif
```

APPENDIX O
C++ FUZZY CLASS DEFINITIONS

```
/*----------------------------------------------------------------------------
PROGRAM: fuzzy.cpp
AUTHOR: Mark Workman
DATE:  8-20-96
VERSION: 1.0
DESCRIPTION: This file contains the fuzzy class member functions. These fuzzy functions
(no pun intentended) were originally built by the Motorola FUDGE fuzzy development
tool. FUDGE was written by Alex DeCastro and Jason Spielman. The current version of
FUDGE is v.1.04

Version 1.0
The crisp_inputs[] and crisp_outputs[] data arrays are now encapsulated within the
fuzzy class. The operator[] has been overloaded to facilitate these two arrays. Now, to
access values in the crisp_inputs/crisp_outputs arrays; just use the corresponding
input/output number as the array index. All crisp inputs are indexed by positive integer
values greater than zero. All crisp outputs are indexed by negative integer values
less than zero.

        Valid crisp_input values will be integers 1,2,3, ...,N. Where N is last crisp input.
        Example to change crisp_input #2:   fuzzy_engine[2] = 3.4567;

        Valid crisp_output values will be integers -1,-2,-3, ...,-M. Where M is last crisp input.
        Example to retrieve crisp_output #2:   fValue = fuzzy_engine[-2];

Three new global constants have been added to indicate NUM_INPUTS, NUM_OUTPUTS and
NUM_RULES
Respectively, They indicate the number of crisp inputs, crisp outputs and rules.

INCLUDE FILES:
The "fuzzy.hpp" file contains the (C++) fuzzy class definition.
The "fuzzy.cpp" file contains the fuzzy class member functions.
The "fuzzy.knb" file contains the unique knowledge base created by FUDGE for
your application. NOTE: "fuzzy.knb" is the default filename created by the xFUDGE translation
program. If you changed the KNB filename during the translation process, your filename
will differ from this name.
----------------------------------------------------------------------------*/
#ifndef _FUZZY_CPP
#define _FUZZY_CPP

//INCLUDE FUZZY ENGINE CLASS DEFINITION
#include <iostream.h>
#include "fuzzy.hpp"
```

```
//PUBLIC FUNCTIONS
fuzzy::fuzzy() //CONSTRUCTOR initializes fuzzy arrays
{       //initialize private data structures
        num_inputs  = NUM_INPUTS;
        num_outputs = NUM_OUTPUTS;
        num_rules   = NUM_RULES;

        for(int i=0; i<NUM_OUTPUTS; i++)
                crisp_outputs[i]=0.0;

        for(int j=0; j<NUM_INPUTS; j++)
                crisp_inputs[j] =0.0;

        for(i=0; i<8 ; i++)
                for(j=0; j<8; j++)
                        fuzzy_inputs[i][j] = 0.0;

        for(i=0; i<4 ; i++)
                for(j=0; j<8; j++)
                        fuzzy_outputs[i][j] = 0.0;
}//end default constructor

//FUZZY FUNCTIONS
// fuzzy_step() - Runs the fuzzy engine; evaluates crisp inputs and
//          creates crisp outputs
//   INPUT:  Crisp input values must be preloaded in the crisp_inputs array
//   OUTPUT: Crisp outputs are left in the crisp_outputs array
void fuzzy::fuzzy_step()  //Runs the fuzzy engine
{
        int in_index, rule_index, out_index;
        //Fuzzify all inputs
        for (in_index = 0;in_index < num_inputs;in_index++)
        {
                fuzzify_input(in_index,crisp_inputs[in_index]);
        }
        //Evaluate each rule
        for (rule_index = 0;rule_index < num_rules;rule_index++)
        {
                eval_rule(rule_index);
        }
        //Create crisp outputs
        for (out_index = 0;out_index < num_outputs;out_index++)
        {
                crisp_outputs[out_index] = defuzzify_output(out_index, crisp_inputs);
        #if TRACE
                cout << "crisp_output[" << out_index << "] = "
                        << crisp_outputs[out_index] << endl;
        #endif
        }
}// end fuzzy_step
```

```
// operator[] - overload function for the crisp_inputs and crisp_outputs arrays
//  INPUT: positive index values relate to the crisp_inputs[]
//         negative index values relate to the crisp_outputs[]
//         index zero is unused
//  OUTPUT: the value of the corresponding crisp_input/crisp_output at position index
float &fuzzy::operator[](int index)
{
        if(index==0 || index>NUM_INPUTS || index<-NUM_OUTPUTS)
                cout << "\nINVALID INDEX!!\n";
        else if(index<0 && index>= -NUM_OUTPUTS)
                return crisp_outputs[(-1*index)-1];
        return crisp_inputs[index-1];
}//end operator[]



//PRIVATE FUNCTIONS
//fuzzify_input - Evaluates each crisp input and determines its fuzzy input values.
//    Fuzzy values are determined by the relationship of the crisp input to each
//       of the fuzzy input membership functions. The fuzzy input results are stored
//       in the fuzzy_inputs[] array.
void fuzzy::fuzzify_input(int in_index,float in_val)
{
        #if TRACE
        cout << "Fuzzify: input #" << in_index
                        <<"crisp value " << in_val << endl;
        #endif

        for (int i = 0; i<num_input_mfs[in_index]; i++)
        {
                fuzzy_inputs[in_index][i] = get_membership_value(in_index,i,in_val);
                #if TRACE
                cout << "Membership function #" << i
                                << " grade" << fuzzy_inputs[in_index][i] << endl;
                #endif
        }
} // end fuzzify_input
```

```
//get_membership_value - This function returns the degree of truthfulness for a crisp
//        input vs. an inputs fuzzy membership function. The value returned will always
//        be a floating point number between zero and one.
float fuzzy::get_membership_value(int in_index,int mf_index,float in_val)
{  //evaluate points outside the membership function
        if (in_val < inmem_points[in_index][mf_index][0]) return 0;
        if (in_val > inmem_points[in_index][mf_index][3]) return 0;
        //evaluate the first half of the functions upward slope
        if (in_val <= inmem_points[in_index][mf_index][1])      {
                if (inmem_points[in_index][mf_index][0] == inmem_points[in_index][mf_index][1])
                        return 1;
                else
                        return ((in_val - inmem_points[in_index][mf_index][0]) /
                                (inmem_points[in_index][mf_index][1] -
inmem_points[in_index][mf_index][0]));
        }
        //evaluate the second half of the functions downward slope
        if (in_val >= inmem_points[in_index][mf_index][2])      {
                if (inmem_points[in_index][mf_index][2] == inmem_points[in_index][mf_index][3])
                        return 1;
                else
                        return ((inmem_points[in_index][mf_index][3] - in_val) /
                                (inmem_points[in_index][mf_index][3] -
inmem_points[in_index][mf_index][2]));
        }
        return 1;
}// end get_membership_value

//eval_rule -  This function uses the min/max technique of rule evaluation. All
//    antecedents of a rule are compared and the min truth value taken. That min
//    antecedent value is then applied to all the rules conequences. If the value
//    is the maximum value of a given cons then it is taken; else it is thrown away.
//    All ants and cons are index into the fuzzy_inputs[] array by their respective
//    element value in the Rules[] array. The results of the rule evaluation are stored
//    in the fuzzy_outputs[] array.
void fuzzy::eval_rule(int rule_index) {
        int   in_index, out_index, mf_index, ant_index, con_index;
        int   val;
        float rule_strength = 1;
        //evaluate all ants
        for (ant_index = 0; ant_index<num_rule_ants[rule_index]; ant_index++)          {
                val = Rules[rule_index].antecedent[ant_index];
                in_index = (val & 0x07);
                mf_index = ((val & 0x38) >> 3);
                rule_strength = MIN(rule_strength,fuzzy_inputs[in_index][mf_index]);
        }
        #if TRACE
        cout << "Rule  #"<<rule_index << " strength = " << rule_strength << endl;
        #endif
```

```
        //evaluate all cons
        for (con_index = 0;con_index < num_rule_cons[rule_index];con_index++)          {
                val = Rules[rule_index].consequent[con_index];
                out_index = (val & 0x03);
                mf_index = ((val & 0x38) >> 3);
                fuzzy_outputs[out_index][mf_index] = MAX(fuzzy_outputs[out_index][mf_index],
                                rule_strength);
        }
}// end eval_rule

//defuzzify_output - This function evaluates the fuzzy_output[] array vs. each crisp
//      output. It determins the correct crisp value for each fuzzy output with respect
//      to an outputs membership functions. Each crisp output value is stored in the
//      crisp_outputs[] array.
float fuzzy::defuzzify_output(int out_index,float *inputs) {
        float  summ = 0;
        float  product = 0;
        float  temp1,temp2;
        int    mf_index,in_index;

        #if TRACE
        cout << "Defuzzify: output #" << out_index << endl;
        #endif
        //Determine the crisp out[ut values
        for (mf_index = 0;mf_index < num_output_mfs[out_index];mf_index++)          {
                temp1 = fuzzy_outputs[out_index][mf_index];
                temp2 = outmem_points[out_index][mf_index][0];
                summ = summ + temp1;
                product = product + (temp1 * temp2);

                #if TRACE
                cout << "Membership function #" << mf_index
                                << "grade " << fuzzy_outputs[out_index][mf_index] << endl;
                #endif

                fuzzy_outputs[out_index][mf_index] = 0;
        }
        //check to see if any rules fired
        if (summ > 0)       {
                crisp_outputs[out_index] = product / summ;
                return crisp_outputs[out_index];
        }
        else        {
                #if NO_RULES
                cout << "No rules fire for:\n";
                #endif

                for (in_index = 0;in_index < num_inputs;in_index++)
                                cout << "Input #" << in_index << '='<< inputs[in_index] << endl;
                return crisp_outputs[out_index];
        }
} // end defuzzify_output
#endif
```

APPENDIX P
68HC05 CODE FOR WASHING MACHINE
EXAMPLE FUZZY ENGINE

```
*       FUzzy Development and Generation Environment (FUDGE) Version V1.02
*       MC68HC05 assembly file
*       John Dumas & Jason Spielman & Alex DeCastro
*       Copyright Motorola 1994

INPUT_MFS       EQU     *       ; Input Membership Functions
IN0MF           EQU     *       ;       Dirtiness
                FCB     $00,$00,$00,$80,$00,$02         ;       Small
                FCB     $00,$80,$80,$ff,$02,$02         ;       Medium
                FCB     $80,$ff,$ff,$ff,$02,$00         ;       Large
IN1MF           EQU     *       ;       TypeOfDirt
                FCB     $00,$00,$00,$80,$00,$02         ;       NonGreasy
                FCB     $00,$80,$80,$ff,$02,$02         ;       Medium
                FCB     $80,$ff,$ff,$ff,$02,$00         ;       Greasy
SGLTN_POS       EQU     *       ; Output Membership Functions
OUT0MF          EQU     *       ;       WashTime
                FCB     $22             ;       VeryShortTime
                FCB     $33             ;       ShortTime
                FCB     $55             ;       MediumTime
                FCB     $bf             ;       LongTime
                FCB     $ff             ;       VeryLongTime
RULE_START      EQU     *       ; Rules follow:
                FCB     $02
                FCB     $05
                FCB     $84
                FCB     $01
                FCB     $05
                FCB     $83
                FCB     $00
                FCB     $05
                FCB     $83
                FCB     $02
                FCB     $02
                FCB     $04
                FCB     $83
                FCB     $01
                FCB     $04
                FCB     $82
                FCB     $00
                FCB     $04
                FCB     $82
                FCB     $02
                FCB     $03
                FCB     $82
                FCB     $01
                FCB     $03
                FCB     $81
                FCB     $00
                FCB     $03
                FCB     $80
END_OF_RULE     FCB     $ff
NUMINP  EQU     $2
NUMOUT          EQU     $1
LPI     EQU     3
LPO     EQU     5
DEFVER  FCC     '1.02'
```

APPENDIX Q
68HC05 CODE FOR TRAFFIC LIGHT EXAMPLE FUZZY ENGINE

```
*       FUzzy Development and Generation Environment (FUDGE) Version V1.02
*       MC68HC05 assembly file
*       John Dumas & Jason Spielman & Alex DeCastro
*       Copyright Motorola 1994

INPUT_MFS       EQU     *               ; Input Membership Functions
IN0MF           EQU     *               ;       GreenLight
                FCB     $00,$00,$00,$15,$00,$0c         ;       Zero
                FCB     $00,$15,$2b,$40,$0c,$0c         ;       Low
                FCB     $2b,$40,$40,$55,$0c,$0c         ;       Medium
                FCB     $40,$55,$ff,$ff,$0c,$00         ;       High
IN1MF           EQU     *               ;       RedLight
                FCB     $00,$00,$00,$15,$00,$0c         ;       Zero
                FCB     $00,$15,$40,$80,$0c,$04         ;       Low
                FCB     $40,$80,$80,$bf,$04,$04         ;       Medium
                FCB     $80,$bf,$ff,$ff,$04,$00         ;       High
IN2MF           EQU     *               ;       CycleTime
                FCB     $00,$00,$46,$8b,$00,$04         ;       Short
                FCB     $46,$8b,$8b,$d1,$04,$04         ;       Medium
                FCB     $8b,$d1,$ff,$ff,$04,$00         ;       Long
SGLTN_POS       EQU     *               ; Output Membership Functions
OUT0MF          EQU     *               ;       Change
                FCB     $00                     ;       No
                FCB     $4d                     ;       ProbNot
                FCB     $80                     ;       Maybe
                FCB     $b3                     ;       ProbYes
                FCB     $ff                     ;       Yes
RULE_START      EQU     *               ; Rules follow:
                FCB     $00
                FCB     $04
                FCB     $80
                FCB     $00
                FCB     $05
                FCB     $84
                FCB     $00
                FCB     $06
                FCB     $84
                FCB     $00
                FCB     $07
                FCB     $84
                FCB     $04
                FCB     $80
                FCB     $01
                FCB     $05
                FCB     $80
                FCB     $02
                FCB     $06
                FCB     $80
                FCB     $03
                FCB     $07
                FCB     $80
                FCB     $01
                FCB     $06
                FCB     $08
                FCB     $82
                FCB     $01
                FCB     $06
                FCB     $09
```

```
FCB     $83
FCB     $01
FCB     $06
FCB     $0a
FCB     $84
FCB     $01
FCB     $07
FCB     $08
FCB     $81
FCB     $01
FCB     $07
FCB     $09
FCB     $82
FCB     $01
FCB     $07
FCB     $0a
FCB     $83
FCB     $02
FCB     $05
FCB     $08
FCB     $81
FCB     $02
FCB     $05
FCB     $09
FCB     $81
FCB     $02
FCB     $05
FCB     $0a
FCB     $82
FCB     $02
FCB     $07
FCB     $08
FCB     $82
FCB     $02
FCB     $07
FCB     $09
FCB     $83
FCB     $02
FCB     $07
FCB     $0a
FCB     $84
FCB     $03
FCB     $05
FCB     $08
FCB     $82
FCB     $03
FCB     $05
FCB     $09
FCB     $83
FCB     $03
FCB     $05
FCB     $0a
FCB     $84
FCB     $03
FCB     $06
FCB     $08
FCB     $81
FCB     $03
FCB     $06
FCB     $09
FCB     $81
FCB     $03
FCB     $06
FCB     $0a
```

```
                        FCB     $82
END_OF_RULE             FCB     $ff
NUMINP  EQU             $3
NUMOUT                  EQU     $1
LPI     EQU             4
LPO     EQU             5
DEFVER  FCC             '1.02'
```

APPENDIX R
68HCO5 CODE FOR TRUCK BACKER-UPPER EXAMPLE FUZZY ENGINE

```
*       FUzzy Development and Generation Environment (FUDGE) Version V1.02
*       MC68HC05 assembly file
*       John Dumas & Jason Spielman & Alex DeCastro
*       Copyright Motorola 1994

INPUT_MFS       EQU     *       ; Input Membership Functions
IN0MF           EQU     *       ;       Phi
                FCB     $00,$20,$20,$45,$08,$07         ;       RightBelow
                FCB     $37,$51,$51,$6e,$0a,$09         ;       RightUpper
                FCB     $60,$6e,$6e,$80,$12,$0e         ;       RightVertical
                FCB     $78,$80,$80,$87,$24,$24         ;       Vertical
                FCB     $80,$8f,$8f,$9f,$10,$10         ;       LeftVertical
                FCB     $91,$ae,$ae,$ca,$09,$09         ;       LeftUpper
                FCB     $bc,$df,$df,$ff,$07,$08         ;       LeftBelow
IN1MF           EQU     *       ;       Position
                FCB     $00,$00,$1f,$59,$00,$04         ;       Left
                FCB     $4d,$66,$66,$80,$0a,$0a         ;       LeftCenter
                FCB     $73,$80,$80,$8c,$14,$14         ;       Center
                FCB     $80,$99,$99,$b3,$0a,$0a         ;       RightCenter
                FCB     $a6,$e0,$ff,$ff,$04,$00         ;       Right
SGLTN_POS       EQU     *       ; Output Membership Functions
OUT0MF          EQU     *       ;       Theta
                FCB     $00             ;       NegBig
                FCB     $40             ;       NegMedium
                FCB     $6a             ;       NegSmall
                FCB     $80             ;       Zero
                FCB     $95             ;       PosSmall
                FCB     $bf             ;       PosMedium
                FCB     $ff             ;       PosBig
RULE_START      EQU     *       ; Rules follow:
                FCB     $00
                FCB     $07
                FCB     $84
                FCB     $01
                FCB     $07
                FCB     $82
                FCB     $02
                FCB     $07
                FCB     $81
                FCB     $03
                FCB     $07
                FCB     $81
                FCB     $04
                FCB     $07
                FCB     $07
                FCB     $80
                FCB     $05
                FCB     $07
                FCB     $80
                FCB     $06
                FCB     $07
                FCB     $80
                FCB     $00
                FCB     $08
                FCB     $85
                FCB     $01
                FCB     $08
                FCB     $84
```

```
FCB     $02
FCB     $08
FCB     $82
FCB     $03
FCB     $08
FCB     $81
FCB     $04
FCB     $08
FCB     $81
FCB     $05
FCB     $08
FCB     $80
FCB     $06
FCB     $08
FCB     $80
FCB     $00
FCB     $09
FCB     $85
FCB     $01
FCB     $09
FCB     $85
FCB     $02
FCB     $09
FCB     $84
FCB     $03
FCB     $09
FCB     $83
FCB     $04
FCB     $09
FCB     $82
FCB     $05
FCB     $09
FCB     $81
FCB     $06
FCB     $09
FCB     $81
FCB     $00
FCB     $0a
FCB     $86
FCB     $01
FCB     $0a
FCB     $86
FCB     $02
FCB     $0a
FCB     $85
FCB     $03
FCB     $0a
FCB     $85
FCB     $04
FCB     $0a
FCB     $84
FCB     $05
FCB     $0a
FCB     $82
FCB     $06
FCB     $0a
FCB     $81
FCB     $00
FCB     $0b
FCB     $86
FCB     $01
FCB     $0b
FCB     $86
FCB     $02
```

```
                        FCB     $0b
                        FCB     $86
                        FCB     $03
                        FCB     $0b
                        FCB     $85
                        FCB     $04
                        FCB     $0b
                        FCB     $85
                        FCB     $05
                        FCB     $0b
                        FCB     $84
                        FCB     $06
                        FCB     $0b
                        FCB     $82
END_OF_RULE             FCB     $ff
NUMINP  EQU             $2
NUMOUT                  EQU     $1
LPI             EQU     7
LPO             EQU     7
DEFVER  FCC             '1.02'
```

APPENDIX S
68HC05 CODE FOR FUZZY INFERENCE PROCESSOR

```
*=============================================================================
*
*    F U Z Z Y   L O G I C   I N F E R E N C E   E N G I N E
*    68HC05 Model
*    D G Weiss     Motorola     Semiconductor Sector     Microcontroller Technologies Group
*    Digital Signal Processor Division
*    Parallel Scalable Processors /
*      Center For Emerging Computer Technologies
*
* 0.0  4/20/92 DGW Original translation from 11 to 05
* 0.1  5/01/92 DGW Early stupid-bug fixes
* 1.0 11/11/93 DGW Corrected bhi to bne after label DeF05. Added
*              "MFSize equ 6" so it needn't be done outside this file
*
* This is a fuzzy engine for the 68HC05 architecture; it was adapted
* from the second major generation of fuzzy engine for the 68HC11
* architecture.
*
* A program is being written which allows the graphical, interactive
* definition of inputs, their membership functions and labels, and
* outputs (and their labels and centroids); this program generates
* the fixed tables as outputs which can be prepended or included with
* this program file, and the remainder of the user's application (I/O
* routines, at a minimum) to form an entire system.
*
*  Be sure that the Knowledge Base file is generating the right kind of
*  constant data structures for the engine being used!
*=============================================================================
*
*-----------------------------------------------------------------------------*
*     Definitions required from knowledge base file.
*     The numbers shown below are representative.
*INPUT_MFS equ  ?      Address of inputs
*NUMINP equ    4       Number of inputs
*LPI   equ    8       Labels per input
*SGLTN_POS equ  ?      Address of singleton position table
*NUMOUT equ    2       Number of outputs
*LPO   equ    8       Labels per Output
*RULE_START equ ?      Address of rule table
*-----------------------------------------------------------------------------*
     page
*-----------------------------------------------------------------------------*
*     Local symbol definitions and redefinitions
```

```
MFSize  equ    6
NumInps equ    NUMINP
NumOuts equ    NUMOUT
COGPos  equ    SGLTN_POS
Rules   equ    RULE_START
P1      equ    0
P2      equ    1
P3      equ    2
P4      equ    3
S1      equ    4
S2      equ    5
B7      equ    Bit7

*-------------------------------------------------------------------------*
*       RAM Data structures (fuzzy engine variables)
*-------------------------------------------------------------------------*
        BSCT
Inputs      rmb   NumInps       Crispy inputs
EndInputs   equ   *
InPtr       rmb   1
FuzyIns     rmb   NumInps*LPI   Fuzzified inputs
FuzyInPtr   rmb   1
MFPtr       rmb   1
MFCounter   rmb   1
RulePtr     rmb   1
MinRuleStren rmb  1
FuzyOuts    rmb   NumOuts*LPO   Fuzy outputs
FuzyOutPtr  rmb   1
Outputs     rmb   NumOuts       Crispified outputs
EndOutputs  equ   *
OutPtr      rmb   1
J           rmb   1
SumOfFuz    rmb   2             11-bit sum of fuzzy outs
SumOfProd   rmb   3             19-bit sum of products
Ctr         rmb   1
*Temps      rmb   9             Optimized allocation of misc. variables
*InPtr      equ   Temps+0       (from 15 -> 9 bytes of scalar temps)
*FuzyInPtr  equ   Temps+1
*MFPtr      equ   Temps+2
*MFCounter  equ   Temps+3
*RulePtr    equ   Temps+0
*MinRuleStren equ Temps+1
*FuzyOutPtr equ   Temps+0
*OutPtr     equ   Temps+1
*J          equ   Temps+2
*SumOfFuz   equ   Temps+3
*SumOfProd  equ   Temps+5
*Ctr        equ   Temps+8
        page
```

```
*-------------------------------------------------------------------------*
*          F U Z Z Y   I N F E R E N C E   E N G I N E
*-------------------------------------------------------------------------*
      PSCT
*     org    $200          debugging address simplification
Fuzzify
*     Initialize the MF, Fuzzy Input, and Input indices
      lda    #NumInps*LPI*MFSize
      sta    MFPtr
      lda    #NumInps*LPI
      sta    FuzyInPtr
      lda    #NumInps

*-------------------------------------------------------------------------*
*     F U Z Z I F Y   I N P U T S
*-------------------------------------------------------------------------*
*     Do one Fuzzy Input
*     Decrement the Input index
Fuz04 deca
      sta    InPtr
*           Do LPI Membership Grade Evaluations for the current input
*           --Initialize the MF counter ('LPI' MFs per input)
      lda    #LPI          MFCounter := LPI
      sta    MFCounter
*           Do one Membership Grade Evaluation
*           --Decrement the MF descriptor index
Fuz06 lda    MFPtr         MFPtr--
      sub    #MFSize
      sta    MFPtr
      ldx    InPtr         A := Inputs[InPtr]
      lda    Inputs,x
      ldx    MFPtr

*-------------------------------------------------------------------------*
* Grade - project a discrete input value onto the specified input membership
*     function (fuzzification process).
*
*     Assumes: A = Input value
*           X = offset of MF description in table 'MT'
*     Returns: A = Strength of membership
*           X = garbage
*-------------------------------------------------------------------------*
MT    equ    INPUT_MFS
Max   equ    $FF

Grade
      cmp    MT+P1,x 5    If I >= P1
      blo    G01    3
      cmp    MT+P4,x 5      Then If I <= P4
      bhi    G01    3
      cmp    MT+P2,x 5            Then If I < P2
      bhs    G02    3
      sub    MT+P1,x 5                  Then M := (I-P1)*S1
```

```
      ldx   MT+S1,x 5
      mul         11
      bra   G03   3
G02   cmp   MT+P3,x 5              Else If I <= P3
      bhi   G04   3
      lda   #Max  2                Then M := MaxM
      bra   G03   3
G04   nega        3                Else M := (P4-I)*S2
      add   MT+P4,x 5
      ldx   MT+S2,x 5
      mul         11
      bra   G03   3
G01   lda   #0    2
G03
*     Store the membership strength in the fuzzy input array
      dec   FuzyInPtr    FuzyInPtr--
      ldx   FuzyInPtr    FuzyIns[FuzyInPtr] := a
      sta   FuzyIns,x
*     Count down the MF counter; decide whether done with current input
      dec   MFCounter
      bne   Fuz06
*     Inspect the Input index; decide whether done with all inputs
      lda   InPtr
      bne   Fuz04
      page

*=============================================================================*
*     E V A L U A T E   R U L E S
*-----------------------------------------------------------------------------*
      lda   #0          Initialize the fuzzy output array
      ldx   #NumOuts*LPO-1
Ru02  sta   FuzyOuts,x
      decx
      bpl   Ru02

      ldx   #0          Index start of 1st rule
      stx   RulePtr
      ldx   Rules,x     Get If clause byte

*-----------------------------------------------------------------------------*
Ru10            ! Process an If-Part (also called a "When-Part")
*-----------------------------------------------------------------------------*
      lda   #$FF        Init Min Rule Strength variable to max value
      sta   MinRuleStren
*-----------------------------------------------------------------------------*
Ru12            ! Process an Antecedent
*-----------------------------------------------------------------------------*
      lda   FuzyIns,x   MinRuleStren :=
      cmp   MinRuleStren        Min(MinRuleStren,FuzyIns[X])
      bhs   Ru14
      sta   MinRuleStren
```

```
*-------------------------------------------------------------------------*
* Skip remainder of this Rule if its MinRuleStren has already hit zero
*-------------------------------------------------------------------------*
     bne    Ru14
     ldx    RulePtr        {return RulePtr to x}
Ru16  incx                 Repeat x++
     lda    Rules,x          Until Rules[x]:7 == 1 {found next Then-part}
     bpl    Ru16
Ru17  incx                 Repeat x++
     lda    Rules,x
     coma                  If Rules[x] == $FF
     beq    Ru99               Then GoTo Ru99 {found end of rules}
     bpl    Ru17            Until Rules[x]:7 == 0 {found next If-part}
     stx    RulePtr        {restore x to RulePtr}
     ldx    Rules,x        {get clean rule byte (1st byte of If-part)
     bra    Ru10           back into x, then go to process the new rule}

Ru14  inc    RulePtr
     ldx    RulePtr
     ldx    Rules,x        Get new byte
     bpl    Ru12           Loop if still an Antecedent byte
*-------------------------------------------------------------------------*
Ru20                ! Process a consequent
*-------------------------------------------------------------------------*
     lda    MinRuleStren   FuzyOuts[x] :=
     cmp    >FuzyOuts-B7,x      Max(MinRuleStren,FuzyOuts[x])
     bls    Ru22
     sta    >FuzyOuts-B7,x

Ru22  inc    RulePtr
     ldx    RulePtr
     ldx    Rules,x        Get new byte {index into FuzyOuts[]} into x
     bpl    Ru10           If an Antecedent byte, loop back to handle it
     cmpx   #$FF
     bne    Ru20           If Not rules terminator, loop back to Consequent
Ru99                ! Else fall thru; it's the end of the rules.
     page
```

```
*-------------------------------------------------------------------------*
*    D E F U Z Z I F Y   O U T P U T S
*-------------------------------------------------------------------------*
DeFuzzify
     ldx   #NumOuts       OutPtr := NumOuts
     stx    OutPtr
     ldx   #NumOuts*LPO    FuzOutPtr := NumOuts*LPO
     stx    FuzyOutPtr
*                    Repeat
DeF02  dec    OutPtr          OutPtr--
     lda   #0              SumOfFuz := SumOfProd := 0
     sta    <SumOfFuz+0
     sta    <SumOfFuz+1
     sta    <SumOfProd+0
     sta    <SumOfProd+1
     sta    <SumOfProd+2
     lda   #LPO            For J := LPO Downto 1
     sta    J
DeF04  dec    FuzyOutPtr         FuzyOutPtr--
     ldx    FuzyOutPtr
     lda    FuzyOuts,x         If FuzyOuts[FuzyOutPtr] <> 0 Then
     beq    DeF05
     add    <SumOfFuz+1          SumOfFuz += FuzyOuts[FuzyOutPtr]
     sta    <SumOfFuz+1
     lda   #0
     adc    <SumOfFuz+0
     sta    <SumOfFuz+0
     lda    FuzyOuts,x          SumOfProd += FuzyOuts[FuzyOutPtr]
     ldx    COGPos,x                    * COGPos[FuzyOutPtr]
     mul
     add    <SumOfProd+2
     sta    <SumOfProd+2
     txa
     adc    <SumOfProd+1
     sta    <SumOfProd+1
     lda   #0
     adc    <SumOfProd+0
     sta    <SumOfProd+0
DeF05  dec    J
     bne    DeF04
     page
```

```
*                       Outputs[OutPtr] := SumOfProd Div SumOfFuz
*-------------------------------------------------------------------------*
* Divide 24 bit unsigned integer in SumOfProd by 16 bit unsigned integer in
* SumOfFuzz; leave 8-bit quotient and 16-bit remainder in SumOfProd.
*
*   Accepts:
*     Dividend:     SumOfProd[0..2]
*     Divisor:      SumOfFuzz[0..1]
*   Yields:
*     Remainder:    SumOfProd[0..1]
*     Quotient:     SumOfProd[2]
*-------------------------------------------------------------------------*
Dvdnd2  equ    SumOfProd+0  Dividend high order byte
Dvdnd1  equ    SumOfProd+1
Dvdnd0  equ    SumOfProd+2  Dividend low order byte
Rmndr1  equ    SumOfProd+0  Remainder high order byte
Rmndr0  equ    SumOfProd+1  Remainder low order byte
Quot    equ    SumOfProd+2  Quotient
Dvsor1  equ    SumOfFuzz+0  Divisor high order byte offset
Dvsor0  equ    SumOfFuzz+1  Divisor low order byte offset

Div     equ    *
        lda    #8              For Ctr := 8 Downto 1
        sta    Ctr

DivA    lda    Dvdnd2          shift Dividend left one place
        rola
        rol    Dvdnd0
        rol    Dvdnd1
        rol    Dvdnd2

        lda    Dvdnd1          subtract Divisor from Remainder
        sub    Dvsor0
        sta    Dvdnd1
        lda    Dvdnd2
        sbc    Dvsor1
        sta    Dvdnd2
        lda    Dvdnd0          {Dividend low bit holds subtract carry}
        sbc    #0
        sta    Dvdnd0

        brclr  0,Dvdnd0,DivC   If subtract carry = 1
        lda    Dvdnd1          Then add Divisor back in
        add    Dvsor0
        sta    Dvdnd1
        lda    Dvdnd2
        adc    Dvsor1
        sta    Dvdnd2
        lda    Dvdnd0
        adc    #0
        sta    Dvdnd0
        bra    DivD
```

```
DivC    bset    0,Dvdnd0            Else set hi bit := 1

DivD    dec     Ctr
        bne     DivA

        ldx     OutPtr
        lda     Quot
        sta     Outputs,x
        lda     OutPtr          Until OutPtr == 0
        beq     *+5
        jmp     DeF02

*-------------------------------------------------------------------------*
*       Inference engine has completed one pass of all rules.
*-------------------------------------------------------------------------*
        rts
```

APPENDIX T
XFUDGE GUI INTERFACE
GLOBAL PROCEEDURES

```
Attribute VB_Name = "Module1"
Global InputFileName As String
Global KNBFileName As String
Global OpenFileName As String
Global FileSelected As Boolean

Sub FileExecProc()
   Dim MyAppID
   Dim FileString As String
   Dim FileNameStr As String
   Dim StatusStr As String

   On Error GoTo FileExecError:

   ' determine if the view knb creation box is checked
   If KNBx!StatusBox.Value = 1 Then
      StatusStr = "1"
   Else
      StatusStr = "0"
   End If

   ' check if file exist
   FileNameStr = Dir("xfudge_w.exe")

   ' if the file exist, execute the program
   If FileNameStr <> "" Then
      KNBx!StatusLine.Text = " Creating KNB file: " + KNBFileName
      FileString = "xfudge_w.exe " + InputFileName + " " + KNBFileName + " " + StatusStr
      KNBx!Command1(4).Enabled = True
      MyAppID = Shell(FileString, 1)
      KNBx!StatusLine.Text = " Done!!"

   ' else display error message
   Else
      GoTo FileExecError:
   End If

   Exit Sub

FileExecError:
   ' display a MsgBox with error msg
   Call MsgBox("Unable to locate XFUDGE_W.EXE   XFUDGE_W.EXE should be located in the current directory.", vbOkayOnly +
vbCritical, "Execution Error")
   KNBx!StatusLine.Text = " ERROR: Make sure that XFUDGE_W.EXE is in the current directory."

End Sub

Sub FileOpenProc()
   Dim RetVal

   On Error Resume Next

   ' setup common dialog box
   KNBx.CMD1.filename = ""
   KNBx.CMD1.ShowOpen
   OpenFileName = ""

   If Err <> 32755 Then    ' Err 32755 -> user chose cancel
```

```
      FileSelected = True
      OpenFileName = KNBx.CMD1.filename
      KNBx!StatusLine.Text = " Ready!!"
   Else
      FileSelected = False
      OpenFileName = ""
      KNBx!StatusLine.Text = " Waiting..."
   End If

End Sub

Sub FileKNBViewProc()
   Dim NotepadStr As String
   Dim KNBNameStr As String
   Dim MyAppID

   On Error GoTo FileError:

   KNBNameStr = LTrim(KNBFileName) 'remove any leading spaces
   ' check for notepad in the Windows subdirectory
   NotepadStr = Dir("C:\Windows\notepad.exe")
   KNBNameStr = Dir(KNBNameStr) 'check for the knb file to exist
   If KNBNameStr = "" Then
     KNBx!StatusLine.Text = " Unable to locate requested file:" + KNBFileName
     Call MsgBox("Unable to locate requested file.", vbOkayOnly + vbExclamation, "Launch Error")
   Elseif NotepadStr <> "" Then
      MyAppID = Shell("notepad.exe" + " " + KNBNameStr, 1)
   Else
      NotepadStr = Dir("D:\Windows\notepad.exe")
      If NotepadStr <> "" Then
         MyAppID = Shell("notepad.exe" + " " + KNBNameStr, 1)
      End If
   End If

   ' check for error
   If NotepadStr <> "" Then
      Exit Sub
   End If


FileError:
   ' notepad or KNB file not found, display error msg
   If NotepadStr = "" Then
      Call MsgBox("Unable to launch MS-Window's Notepad...", vbOkayOnly + vbCritical, "Launch Error")
      KNBx!Command1(4).Enabled = False
      KNBx!Command1(6).Enabled = False
      KNBx!StatusLine.Text = " Unable to locate NOTEPAD.EXE"
   ElseIf KNBNameStr = "" Then
      Call MsgBox("Unable to locate requested file.", vbOkayOnly + vbExclamation, "Launch Error")
      KNBx!StatusLine.Text = " Unable to locate requested file:" + KNBFileName
   End If

End Sub


Sub main()

   'only one permitted
   If App.PrevInstance <> 0 Then End

   ' change to the drive and directory of the APP
   ChDir App.Path
   ChDrive App.Path
```

```
    ' load the main form
    KNBx.Show

End Sub

Public Sub FileSourceViewProc()
    Dim NotepadStr As String
    Dim SourceNameStr As String
    Dim MyAppID

    On Error GoTo FileError:

    SourceNameStr = LTrim(InputFileName) 'remove any leading spaces
    ' check for notepad in the Windows subdirectory
    NotepadStr = Dir("C:\Windows\notepad.exe")
    SourceNameStr = Dir(SourceNameStr) 'check for the source file to exist
    If SourceNameStr = "" Then
      KNBx!StatusLine.Text = " Unable to locate requested file:" + InputFileName
      Call MsgBox("Unable to locate requested file.", vbOkayOnly + vbExclamation, "Launch Error")
    ElseIf NotepadStr <> "" Then
      MyAppID = Shell("notepad.exe" + " " + SourceNameStr, 1)
    Else
      NotepadStr = Dir("D:\Windows\notepad.exe")
      If NotepadStr <> "" Then
        MyAppID = Shell("notepad.exe" + " " + SourceNameStr, 1)
      End If
    End If

    ' check for error
    If NotepadStr <> "" Then
      Exit Sub
    End If


FileError:
    ' notepad or source file not found, display error msg
    If NotepadStr = "" Then
      Call MsgBox("Unable to launch MS-Window's Notepad...", vbOkayOnly + vbCritical, "Launch Error")
      KNBx!Command1(4).Enabled = False
      KNBx!Command1(6).Enabled = False
      KNBx!StatusLine.Text = " Unable to locate NOTEPAD.EXE"
    ElseIf SourceNameStr = "" Then
      KNBx!StatusLine.Text = " Unable to locate requested file:" + KNBFileName
      Call MsgBox("Unable to locate requested file.", vbOkayOnly + vbExclamation, "Launch Error")
    End If

End Sub
```

APPENDIX U
XFUDGE GUI INTERFACE FORM

```
VERSION 4.00
Begin VB.Form KNBx
  BackColor      =  &H00C0C0C0&
  BorderStyle    =  3  'Fixed Dialog
  Caption        =  "Form1"
  ClientHeight   =  3690
  ClientLeft     =  270
  ClientTop      =  1605
  ClientWidth    =  8865
  Height         =  4095
  Icon           =  "XKNB.frx":0000
  Left           =  210
  LinkTopic      =  "Form1"
  MaxButton      =  0  'False
  MinButton      =  0  'False
  ScaleHeight    =  5060.891
  ScaleMode      =  0  'User
  ScaleWidth     =  8865
  ShowInTaskbar  =  0  'False
  Top            =  1260
  Width          =  8985
  Begin VB.Timer Timer1
    Interval      =  1000
    Left          =  6480
    Top           =  0
  End
  Begin VB.TextBox StatusLine
    BackColor     =  &H00C0C0C0&
    BeginProperty Font
      name        =  "Arial"
      charset     =  0
      weight      =  400
      size        =  8.25
      underline   =  0  'False
      italic      =  0  'False
      strikethrough  =  0  'False
    EndProperty
    Height        =  315
    Left          =  960
    TabIndex      =  16
    TabStop       =  0  'False
    Text          =  "Text1"
    Top           =  3240
    Width         =  5190
  End
  Begin VB.CommandButton Command1
    Caption       =  "&Defaults"
    BeginProperty Font
      name        =  "MS Sans Serif"
      charset     =  1
      weight      =  700
      size        =  8.25
      underline   =  0  'False
      italic      =  0  'False
      strikethrough  =  0  'False
    EndProperty
    Height        =  375
    Index         =  5
    Left          =  3000
    TabIndex      =  9
```

```
  Top             =  2640
  Width           =  1095
End
Begin VB.CommandButton Command1
  Caption         =  "K&NB File"
  BeginProperty Font
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  Height          =  375
  Index           =  4
  Left            =  7200
  TabIndex        =  7
  Top             =  840
  Width           =  1335
End
Begin VB.TextBox KNBFile
  BackColor       =  &H00FFFFFF&
  BeginProperty Font
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  ForeColor       =  &H00FF0000&
  Height          =  285
  Left            =  480
  MousePointer    =  3   'I-Beam
  TabIndex        =  3
  Text            =  "Text1"
  Top             =  1920
  Width           =  4095
End
Begin VB.TextBox InputFile
  BackColor       =  &H00FFFFFF&
  BeginProperty Font
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  ForeColor       =  &H00FF0000&
  Height          =  285
  Left            =  480
  MousePointer    =  3   'I-Beam
  TabIndex        =  1
  Text            =  "Text1"
  Top             =  600
  Width           =  4095
End
Begin VB.CommandButton Command1
  Caption         =  "&Quit"
  BeginProperty Font
```

```
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  Height          =  375
  Index           =  3
  Left            =  5040
  TabIndex        =  5
  Top             =  2640
  Width           =  1095
End
Begin VB.CommandButton Command1
  Caption         =  "&Translate"
  BeginProperty Font
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  Height          =  375
  Index           =  2
  Left            =  960
  TabIndex        =  0
  Top             =  2640
  Width           =  1095
End
Begin VB.CommandButton Command1
  Caption         =  "Select &KNB File"
  BeginProperty Font
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  Height          =  495
  Index           =  1
  Left            =  4800
  TabIndex        =  4
  Top             =  1800
  Width           =  1695
End
Begin VB.CommandButton Command1
  Caption         =  "Select &Input File"
  BeginProperty Font
    name          =  "MS Sans Serif"
    charset       =  1
    weight        =  700
    size          =  8.25
    underline     =  0   'False
    italic        =  0   'False
    strikethrough =  0   'False
  EndProperty
  Height          =  495
  Index           =  0
```

```
  Left          =  4800
  TabIndex      =  2
  Top           =  480
  Width         =  1695
End
Begin VB.Frame Frame1
  Caption       =  " Input Name for  C Source  File: "
  BeginProperty Font
    name        =  "MS Sans Serif"
    charset     =  1
    weight      =  700
    size        =  8.25
    underline   =  0   'False
    italic      =  -1  'True
    strikethrough  =  0   'False
  EndProperty
  ForeColor     =  &H00FF0000&
  Height        =  855
  Index         =  0
  Left          =  240
  TabIndex      =  10
  Top           =  240
  Width         =  6495
End
Begin VB.Frame Frame1
  Caption       =  " Output Name for  C++ KNB  File: "
  BeginProperty Font
    name        =  "MS Sans Serif"
    charset     =  1
    weight      =  700
    size        =  8.25
    underline   =  0   'False
    italic      =  -1  'True
    strikethrough  =  0   'False
  EndProperty
  ForeColor     =  &H00FF0000&
  Height        =  855
  Index         =  1
  Left          =  240
  TabIndex      =  11
  Top           =  1560
  Width         =  6495
End
Begin VB.Frame Frame2
  Caption       =  " View "
  BeginProperty Font
    name        =  "MS Sans Serif"
    charset     =  1
    weight      =  700
    size        =  8.25
    underline   =  0   'False
    italic      =  -1  'True
    strikethrough  =  0   'False
  EndProperty
  ForeColor     =  &H000000FF&
  Height        =  1695
  Left          =  7080
  TabIndex      =  12
  Top           =  120
  Width         =  1575
  Begin VB.CheckBox StatusBox
    Caption     =  "KNB Creation"
    BeginProperty Font
      name      =  "MS Sans Serif"
```

```
    charset        = 0
    weight         = 700
    size           = 8.25
    underline      = 0  'False
    italic         = 0  'False
    strikethrough  = 0  'False
  EndProperty
  Height           = 255
  Left             = 240
  TabIndex         = 8
  Top              = 1200
  Width            = 255
End
Begin VB.CommandButton Command1
  Caption          = "&Source File"
  BeginProperty Font
    name           = "MS Sans Serif"
    charset        = 1
    weight         = 700
    size           = 8.25
    underline      = 0  'False
    italic         = 0  'False
    strikethrough  = 0  'False
  EndProperty
  Height           = 375
  Index            = 6
  Left             = 120
  TabIndex         = 6
  Top              = 240
  Width            = 1335
End
Begin VB.Label Label4
  Caption          = "KNB Creation"
  BeginProperty Font
    name           = "MS Sans Serif"
    charset        = 0
    weight         = 700
    size           = 8.25
    underline      = 0  'False
    italic         = 0  'False
    strikethrough  = 0  'False
  EndProperty
  Height           = 495
  Left             = 600
  TabIndex         = 18
  Top              = 1200
  Width            = 855
End
End
Begin VB.Label Label5
  Alignment        = 2  'Center
  Caption          = "Help Line"
  BeginProperty Font
    name           = "MS Sans Serif"
    charset        = 0
    weight         = 700
    size           = 8.25
    underline      = 0  'False
    italic         = 0  'False
    strikethrough  = 0  'False
  EndProperty
  ForeColor        = &H000000FF&
  Height           = 255
  Left             = 0
```

```
  TabIndex      =  17
  Top           =  3240
  Width         =  855
End
Begin VB.Label Label3
  Alignment     =  2  'Center
  Caption       =  $"XKNB.frx":030A
  Height        =  975
  Left          =  6480
  TabIndex      =  15
  Top           =  2640
  Width         =  2295
  WordWrap      =  -1  True
End
Begin VB.Label Label2
  Alignment     =  2  'Center
  Caption       =  "Ver. 1.0"
  Height        =  255
  Left          =  7080
  TabIndex      =  14
  Top           =  2280
  Width         =  975
End
Begin VB.Label Label1
  Caption       =  "xFudge"
  BeginProperty Font
    name        =  "MS Sans Serif"
    charset     =  1
    weight      =  700
    size        =  9.75
    underline   =  0  'False
    italic      =  0  'False
    strikethrough  =  0  'False
  EndProperty
  Height        =  255
  Left          =  7200
  TabIndex      =  13
  Top           =  2040
  Width         =  855
End
Begin VB.Image AboutImg
  Height        =  480
  Left          =  8160
  MousePointer  =  2  'Cross
  Picture       =  "XKNB.frx":038E
  Top           =  2040
  Width         =  480
End
Begin VB.Line Line1
  X1            =  120
  X2            =  6720
  Y1            =  1810.4
  Y2            =  1810.4
End
Begin MSComDlg.CommonDialog CMD1
  Left          =  6840
  Top           =  0
  _version      =  65536
  _extentx      =  847
  _extenty      =  847
  _stockprops   =  0
  cancelerror   =  -1  True
End
End
```

```
Attribute VB_Name = "KNBx"
Attribute VB_Creatable = False
Attribute VB_Exposed = False



Private Sub AboutImg_Click()
   Dim msg As String
   msg = msg + "          xFUDGE" + Chr(13) + Chr(10)
   msg = msg + "         Version 1.0 " + Chr(13) + Chr(10)
   msg = msg + "       Mark Workman " + Chr(13) + Chr(10)
   msg = msg + "Texas Tech University    " + Chr(13) + Chr(10)
   msg = msg + "       October 1996 "
   Call MsgBox(msg, vbOkayOnly + vbInformation, " About xFUDGE ")
End Sub

Private Sub AboutImg_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
        KNBx!StatusLine.Text = " About xFUDGE"

End Sub


Private Sub Command1_Click(Index As Integer)

   Select Case (Index)

     ' grab input file name
     Case 0: KNBx.CMD1.DialogTitle = "Select Input C Source Code File"
          KNBx.CMD1.Filter = "C Source Files (*.c)|*.c | All files (*.*)|*.*"
          Call FileOpenProc
          If FileSelected = True Then
             InputFileName = " " + OpenFileName
             InputFile.Text = InputFileName
          End If
          Command1(2).SetFocus

     ' grab KNB file name
     Case 1: KNBx.CMD1.DialogTitle = "Select Output KNB Translation File"
          KNBx.CMD1.Filter = "KNB Files (*.knb)|*.knb | All files (*.*) | *.*"
          Call FileOpenProc
          If FileSelected = True Then
             KNBFileName = " " + OpenFileName
             KNBFile.Text = KNBFileName
          End If
          Command1(2).SetFocus

     ' okay button selected
     Case 2: Command1(2).SetFocus
          Call FileExecProc

     ' quit button selected
     Case 3: Command1(2).SetFocus
          Unload KNBx

     ' view the output KNB file
     Case 4: Command1(2).SetFocus
          Call FileKNBViewProc

     ' load the default files
     Case 5: InputFile.Text = " FUZZY.C"
          KNBFile.Text = " FUZZY.KNB"

          InputFileName = "FUZZY.C"
          KNBFileName = "FUZZY.KNB"
```

```
            StatusBox.Value = 0
            Command1(4).Enabled = True
            Command1(2).SetFocus
            Command1(5).Enabled = False

        'view input source file
        Case 6: Command1(2).SetFocus
            Call FileSourceViewProc

        ' default case
        Case Else
            Command1(2).SetFocus
            Unload KNBx

    End Select
End Sub

Private Sub Command1_GotFocus(Index As Integer)

    Select Case (Index)

        ' grab input file name
        Case 0:
            KNBx!StatusLine.Text = " Select Input (C source) filename?"
        ' grab KNB file name
        Case 1:
            KNBx!StatusLine.Text = " Select Output KNB filename?"
        ' okay button selected
        Case 2:
            KNBx!StatusLine.Text = " Translate: Create a C++ KNB file from the Input file?"
        ' quit button selected
        Case 3:
            KNBx!StatusLine.Text = " Quit Program?"
        ' view the output KNB file
        Case 4:
            KNBx!StatusLine.Text = " View KNB files?"
        ' load the default files
        Case 5:
            KNBx!StatusLine.Text = " Return to default settings!"
        'view input source file
        Case 6:
            KNBx!StatusLine.Text = " View Source (C code) file?"

        ' default case
        Case Else

    End Select

End Sub


Private Sub Command1_MouseMove(Index As Integer, Button As Integer, Shift As Integer, X As Single, Y As Single)

    Select Case (Index)

        ' grab input file name
        Case 0:
            KNBx!StatusLine.Text = " Select Input (C source) filname?"
        ' grab KNB file name
        Case 1:
            KNBx!StatusLine.Text = " Select Output KNB filename?"
        ' okay button selected
        Case 2:
```

```
        KNBx!StatusLine.Text = " Translate: Create a C++ KNB file from the Input file?"
    ' quit button selected
    Case 3:
        KNBx!StatusLine.Text = " Quit Program?"
    ' view the output KNB file
    Case 4:
        KNBx!StatusLine.Text = " View KNB files?"
    ' load the default files
    Case 5:
        KNBx!StatusLine.Text = " Return to default settings!"
    'view input source file
    Case 6:
        KNBx!StatusLine.Text = " View Source (C code) file?"

    ' default case
    Case Else

  End Select
End Sub


Private Sub Form_Load()

  ' resize and positon form
  Me.Width = 8985
  Me.Height = 4100
  Me.Left = (Screen.Width - Me.Width) / 2
  Me.Top = (Screen.Height - Me.Height) / 3

  ' show the form
  Me.Show

  ' modify the caption
  Me.Caption = "xFUDGE Translation Interface"

  ' load default file names
  InputFile.Text = " FUZZY.C"
  KNBFile.Text = " FUZZY.KNB"

  InputFileName = "FUZZY.C"
  KNBFileName = "FUZZY.KNB"

  'setup status line
  StatusLine.Text = " Waiting"

  ' setup command buttons
  Command1(4).Enabled = True
  Command1(5).Enabled = False
  Command1(6).Enabled = True
  Command1(2).SetFocus
End Sub


Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
        KNBx!StatusLine.Text = " Ready..."
End Sub


Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
  Dim RetVal
  Dim MsgPrompt As String
  Dim MsgTitle As String

  MsgPrompt = "Exit  xFUDGE  Interface?"
```

```
    MsgTitle = "Quit Application?"

    RetVal = MsgBox(MsgPrompt, vbYesNo + vbQuestion, MsgTitle)
    If RetVal = vbYes Then
        Cancel = False
    Else
        Cancel = True
    End If
End Sub



Private Sub InputFile_Change()
    InputFileName = InputFile.Text
    Command1(5).Enabled = True
End Sub


Private Sub InputFile_GotFocus()
    KNBx!StatusLine.Text = " Type in the Input (C Source) Filename."
End Sub


Private Sub InputFile_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
  KNBx!StatusLine.Text = " Type in the Input (C Source) Filename."
End Sub



Private Sub KNBFile_Change()
    KNBFileName = KNBFile.Text
    Command1(4).Enabled = True
    Command1(5).Enabled = True
End Sub



Private Sub KNBFile_GotFocus()
    KNBx!StatusLine.Text = "  Type in the Output KNB Filename."
End Sub

Private Sub KNBFile_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    KNBx!StatusLine.Text = "  Type in the Output KNB Filename."
End Sub



Private Sub Label5_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
        KNBx!StatusLine.Text = " Help Line."
End Sub


Private Sub StatusBox_Click()
   Command1(2).SetFocus
   Command1(5).Enabled = True
End Sub


Private Sub StatusBox_GotFocus()
```

```
        KNBx!StatusLine.Text = " View the creation of the KNB file? (press space to select)"
End Sub

Private Sub StatusBox_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
        KNBx!StatusLine.Text = " View the creation of the KNB file?"
End Sub
```

APPENDIX V
XFUDGE GRAPHICAL USER INTERFACE

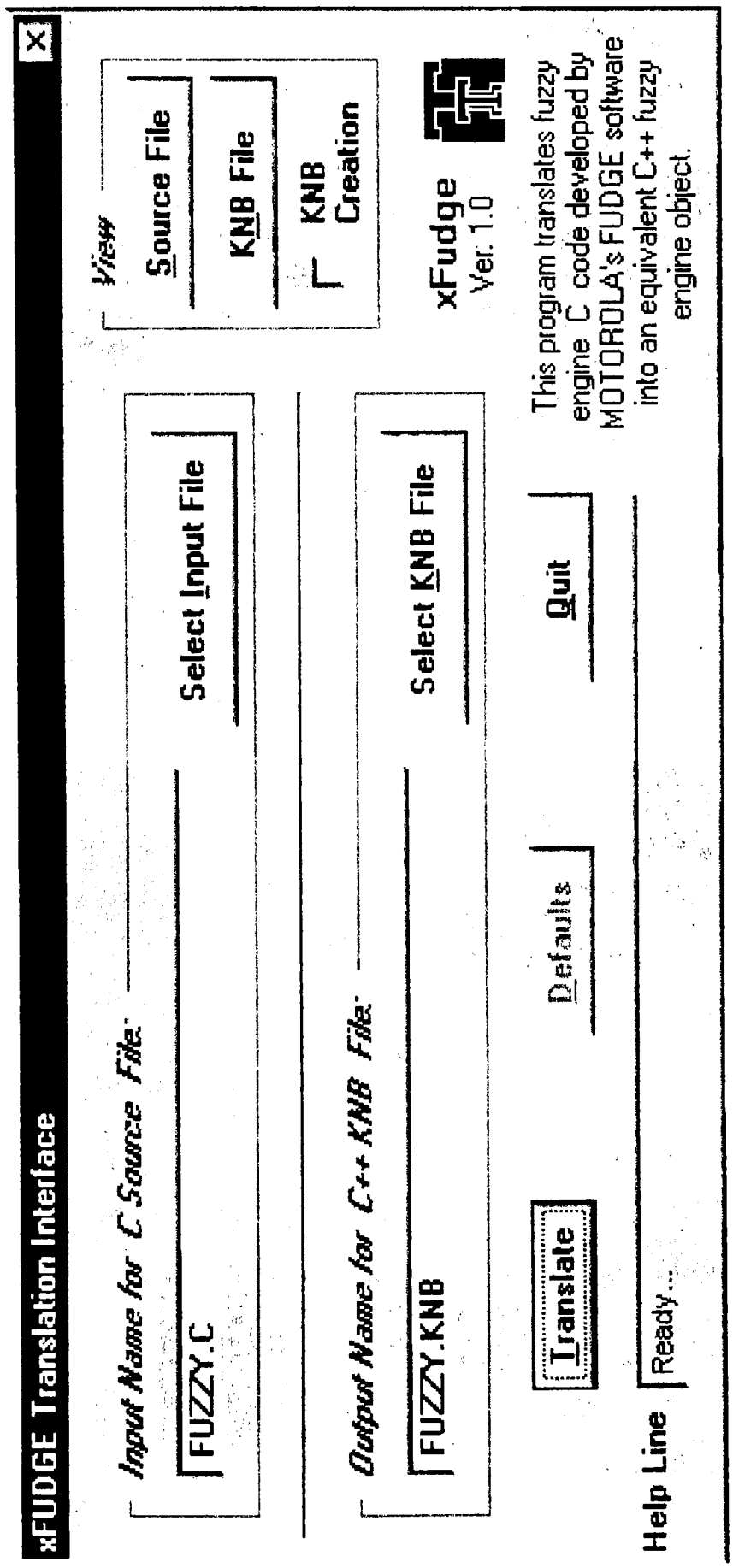


Figure 31. XFUDGE Graphical User Interface.