

C-Based Fuzzy Logic Implemented on the MC68HC12

By: Jeff Kornblith and Jim Harrow

ECSE-4790-01
Microprocessor Systems

December 11th, 2001

Table of Contents

List of Figures	1
Abstract	2
Introduction	3
Materials	6
Methods	8
Results	22
Discussion	24
References	26
Appendices	
A: Main Header File	28
B: User Interface Header File	35
C: Demonstration Program	50
D: Demonstration Circuit Diagram	53

List of Figures

Input Member Function Example	4
Rule Evaluation Example	5
Initialization of the Data Array	10
Offset Values Connected to Data Array	11
Rules in Memory	12
Inputs Loaded into Data Array	14
Input Membership Function Values Loaded into Data Array	15
Rule Structure	17
Input/Output Membership Function Structure	18

Abstract

The HC12 comes with built in fuzzy logic routines. However, these routines are coded only in assembly and do not come with a C interface. In order to make this power feature of the HC12 available for C programmers, an interface must be created. This interface does not perform the fuzzy logic functionality, it merely hands the proper data to the assembly routines. This is powerful because the assembly routines are much faster than any code that could be written in C. The interface must allow for the programmer to use any number of inputs with any number of fuzzy input states. The output must also be available for the programmer to easily access. This report describes the steps that were taken over the past three months to create this interface.

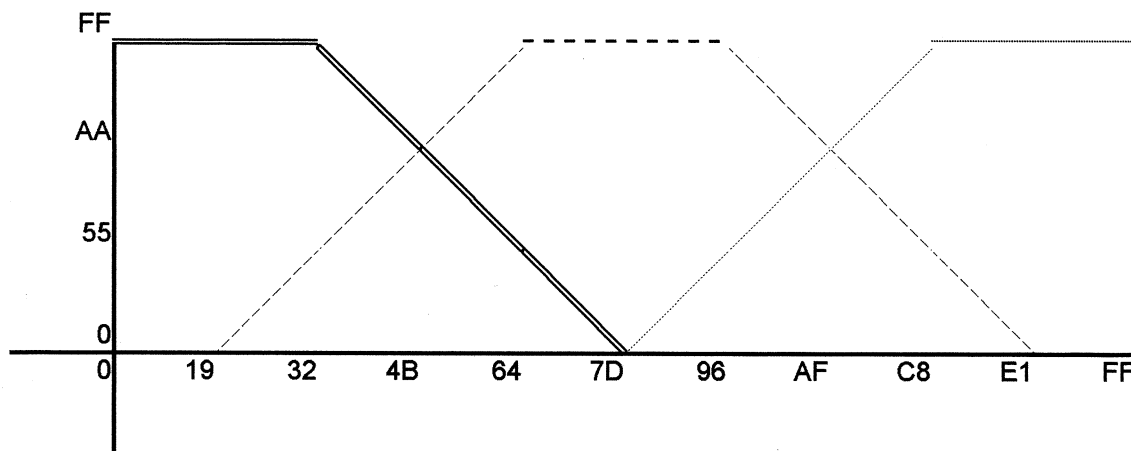
INTRODUCTION

Introduction

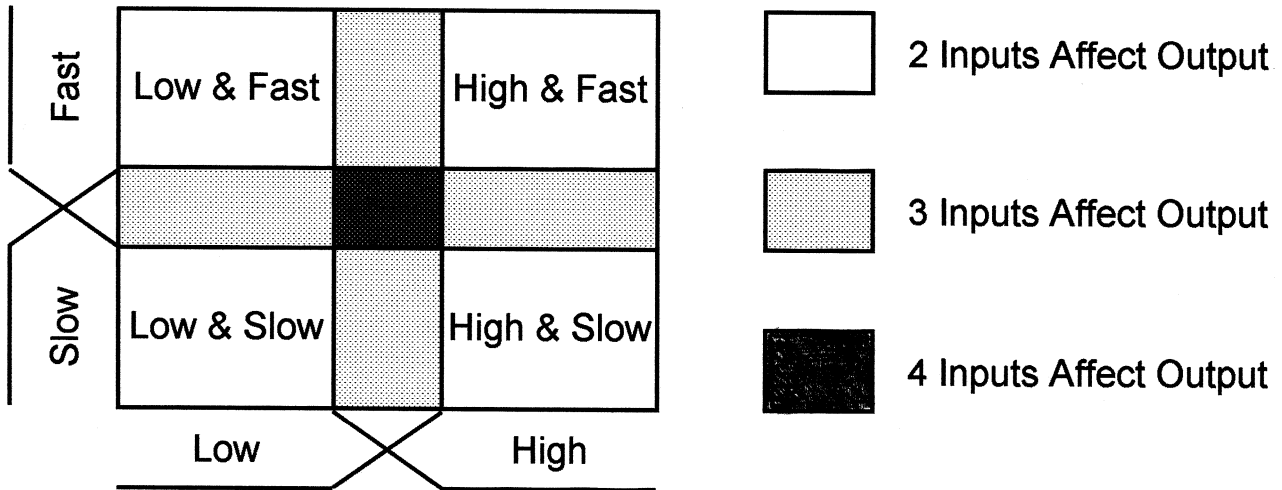
We set out to implement one of the features on the MC68HC12 that was barely touched during class, the hardware driven Fuzzy Logic controller. Our goal was to successfully manipulate the fuzzy logic controls and create a header file that could be used in anyone's program for the HC12.

We needed to understand how fuzzy logic worked in the first place. After doing some research we determined that Fuzzy Logic can be broken down into three steps: Fuzzification; Rule Evaluation and, De-Fuzzification.

Fuzzification involves taking one or more analog inputs and determine values for input membership function. A membership function is basically a graph of that describes based upon an input, when it has a value between 0x00 and 0xFF. The solid line below has a value of 0xFF while the input is below 0x32, and then slowly decreases to 0x00 when the input equals 0x7D. The following applies for the other two lines below.



Once fuzzification determines defined hexadecimal values for each membership function for all the inputs, rule evaluation occurs. Rule evaluation takes the values from the input membership functions and adds up each one to determine the value for each rule. These values become a percentage and are multiplied toward the output membership function's values to create a new value to be used in the last step of fuzzy logic.



De-Fuzzification takes the values of each output membership function and adds them to the value of the output that the output membership functions are a part of. This will create a solid analog value that can be used elsewhere.

Once we understood how fuzzy logic operates, we took the assembly routines provided and translated them into a C based header file. We loaded a set of defined rules and membership functions to simulate a fuzzy logic system, and demonstrated them with a basic circuit involves potentiometers and light emitting diodes.

To further our project, we developed a C based interface that allowed users to define their own membership functions and rules to use in their own programs, followed by a plethora of translation functions to integrate this data into our existing header file instructions.

MATERIALS

List of Materials

Software:

- ProComm Plus – Terminal Program
- Introl C Compiler
- Microsoft Visual C++ Compiler
- Codewarrior C Compiler

Hardware:

- MC68HC12 Evaluation Board

Demonstration Hardware:

- 2 x 1000 Ohm Potentiometers
- 16 x Light Emitting Diodes
- 16 x 220 Ohm Resistors

METHODS

The Interfacing Code

The interfacing code has two main parts to it; the first is formatting the data to a format that the assembly routines understand, and the second is to actually call the assembly routines.

Occasionally in this section the example case will be referred to for clarification. The example case has two inputs, IN1 and IN2 each with two input states, (LOW HIGH) and (SLOW FAST) respectively. There is one output called SIZE, which has four output states SMALL, MEDUIM, LARGE, and HUGE.

The rules for the outputs are as followed:

IN1 LOW & IN2 SLOW	=>	SIZE SMALL
IN1 LOW & IN2 FAST	=>	SIZE MEDIUM
IN1 HIGH & IN2 SLOW	=>	SIZE LARGE
IN1 HIGH & IN2 FAST	=>	SIZE HUGE

The final element of the example case is the output member functions which are 0x00 for SMALL, 0x30 for MEDUIM, 0x80 for LARGE and 0xF0 for HUGE.

Formatting the Data:

The RunFuzzyLogic() function formats the data. The MC68HC12 expects most of the data to be placed in sequential bytes in memory. Single byte numbers but usually nothing separates the sections of data. The inputs tell the processor exactly when to stop. The C version of a sequential block of data is an array. So a global array is created (called _fuzzy_data_array) that is 512 bytes long. The choice of 512 was tricky. Fuzzy logic tables can grow very fast. Each membership function requires 4 bytes alone. This means 4 bytes for each input state. Then the rules require a byte for each input, a byte for each output and 2 separators. So the amount of space needed to hold and process the data grows geometrically as inputs are added!

On the other hand, the MC68HC12 has limited on board memory capabilities and since this interface will be distributed to programmers as merely a tool to use in their

design, it cannot be justified to use all of the MC68HC12's memory (neither can it be expected that additional memory will be supplied). So a value of 512 was selected.

The first block of data is a storage cell for values that will be computed during execution. These are the values for the fuzzy input states, the fuzzy output states and the final output values. For use during execution, markers are kept of the index of the array that specific data begins. For example the “_fuzzy_ins” variable holds the index in the array where the processor will keep the values of the input states (i.e. – LOW, HIGH, SLOW, and FAST). So after this first section of reserving data, the example array will look like this:

_fuzzy_data_array →	0	← IN1 value
	0	← IN2 value
_fuzzy_ins →	0	← LOW
	0	← HIGH
	0	← SLOW
	0	← FAST
_fuzzy_outs →	0	← SMALL
	0	← MEDIUM
	0	← LARGE
	0	← HUGE
_fuzzy_outs →	0	.
	0	.
	0	.

After room has been made to hold the input value, the fuzzy input state values, the fuzzy output state values, and the output value, the program begins writing the specifications of the fuzzy logic system. The first to write are the input membership functions. These define which rules will be activated and how much. Each membership function is defined by 4 byte sized values: two points and two slopes, in that order. The two points are X intercept points of the membership function. Any input that falls outside of these two points is evaluated from 0xFF to 0x00, and any input between these two points is 0xFF. The exact non-zero value is determined based on the slope values. The first slope describes how steep the values increase after the first point. The second slope describes how steep the values fall before the second point. Any area after the first slope has reached 0xFF and before the second slope begins to fall is evaluated as 0xFF (100% on).

The starting location of the input membership function bytes is saved for later use in the `_fuzzy_inputMFs` variable. The four-byte values are stored for each fuzzy input state without breaks (since the processor can count on them being four bytes apiece) in the `_fuzzy_data_array`.

The functions `GetNumInput()`, `GetNumIState()`, and `GetInput()` get the appropriate values from the programmer interface. The next data to be inserted into the array is the rule list. Again, the starting location of the rules is stored in a variable, this time called `_fuzzy_ruleList`. A rule is described in the memory based on the starting location of the fuzzy input state locations. These locations were reserved above and labels with `_fuzzy_ins`. So both the inputs and outputs of the rules are described based on this value. In the example, the data array, a `0x00` refers to LOW (since it is 0 bytes away from `_fuzzy_ins`) and `0x03` refers to HIGH (since it is 3 bytes away from `_fuzzy_ins`) and SMALL is `0x04` and LARGE is `0x06`. Notice that the outputs are indexed based on the start of the inputs as well!

<code>_fuzzy_data_array</code> →	0	← IN1 value	Offsets from <i>fuzzy_ins</i>
	0	← IN2 value	
<code>_fuzzy_ins</code> →	0	← LOW	← 00
	0	← HIGH	← 01
	0	← SLOW	← 02
	0	← FAST	← 03
<code>_fuzzy_outs</code> →	0	← SMALL	← 04
	0	← MEDIUM	← 05
	0	← LARGE	← 06
	0	← HUGE	← 07
<code>_fuzzy_outs</code> →	0	.	
	0	.	
	0	.	

So the first example rule would be `0x00` and `0x02` for the inputs equal `0x04` for the output. The inputs and the outputs are separated with a `0xFE`. The end of one rule and the beginning of another rule is also separated with a `0xFE`. The end of all the rules must be ended with a `0xFF`. So the rules above will go into the array like this:)

__fuzzy_ruleList →	00	low
	02	slow
	FE	←break
	04	small
	FE	← break
	00	low
	03	fast
	FE	← break
	05	medium
	FE	← break
	01	high
	02	slow
	FE	← break
	06	large
	FE	← break
	01	high
	03	fast
	FE	← break
	07	huge
	FF	<< end rules
	...	

The functions GetRule(), GetRuleIn(), GetRuleOut() and GetNumRule() provide the loops with loop information as well as the actual data that is placed in the array.

The last piece of data that needs to be placed into the array is the output functions. These are singleton values that are used in the weighted average calculation of the output. In the example, MEDIUM has a value of 0x30. This means that if the rules evaluate in such a way that MEDIUM is 100% on (has a value of 0xFF) and all the other fuzzy input states evaluate to 0%. Then the output will be 0x30. This is of course, the simplest of examples. The real power of fuzzy logic comes from when MEDIUM is 50% selected and low is 50% selected. The output would be:

$$(0x00*50\% + 0x30*50\%) / (50\% + 50\%)$$

general formula!

These output membership functions, which consist of one value, are just input straight, byte after byte into the array (with the ever important pointer to the first value stored in _fuzzy_outMFs).

So now all of the data that the HC12 uses during its calculations is stored sequentially in the array. By using the import command provided by Introl C-Compiler, all these values are available in the assembly coding section.

Executing the Fuzzy Logic

The RunFuzzy() function actually calls the MC68HC12 assembly routines that perform the fuzzy logic. The first few lines of this function import the needed variables from C to assembly using the 'import' command. Note that the not only are all of the location markers imported, but also the fuzzy_data_array itself. Now this does not import the conventional idea of an array that can use [] to get values off of it. This imports the value that C uses to store an array, the memory address of the first location.

The other two variables that are imported are used as temporary variables. Each is a different size (fuzzy_cnt is an integer, and thus is two bytes long, and fuzzy_temp is a char, one byte long).

This routine assumes that the input values from the analog inputs have been placed into their correct locations at the beginning of the array. So the first step is to fuzzify these inputs. Fuzzify means to evaluate each of the input membership functions and determine how much each fuzzy input state is active. This is the job of the MEM assembly instruction. MEM counts on three things to perform its duty. The first is that the X register holds the memory location of the input membership functions. Recall that the index of this value was stored in the _fuzzy_inputMFs variable. This is the number of bytes after the first byte of the array that the input membership functions begin. So to get the exact address the _fuzzy_inputMFs must be added to the _fuzzy_data_array variable. This is an important fact to realize, since it is used many times in the assembly code.

The next piece of information that MEM needs is the memory location of the first of the fuzzy inputs loaded into register Y. This is the _fuzzy_fuzzyIns variable. This is the location in memory that the 100% or the 50% of LOW, HIGH, etc will be stored.

The third piece of information that is needed is the input value itself in the A register. To do this, we use the input number _fuzzy_cnt1 as an offset for the memory array. The inputs are the very first pieces of data on the array so the offset will be from the beginning of the array.

<i>_fuzzy_data_array</i> →	0x80	← IN1 value	Offsets from <i>fuzzy_ins</i>
	0xFF	← IN2 value	
<i>_fuzzy_ins</i> →	0	← LOW	← 00
	0	← HIGH	← 01
	0	← SLOW	← 02
	0	← FAST	← 03
<i>_fuzzy_outs</i> →	0	← SMALL	← 04
	0	← MEDIUM	← 05
	0	← LARGE	← 06
	0	← HUGE	← 07
<i>_fuzzy_outs</i> →	0	.	
	0	.	
	0	.	

MEM is all ready to evaluate the first membership function. MEM evaluates one member function at a time in the following manner. It uses the membership points and the input value to find the Y value. It then places this Y value in the reserved memory location. This value is the 50% or 100% for the fuzzy input state. The last thing MEM is required to do is increment both the X register and the Y register. This means, that since all the membership functions are right next to each other, when MEM exits, X is already pointing to the next membership function. Incrementing Y means that it is pointing to the next fuzzy input membership function in memory. So by placing MEM between the FUZLOOP: label and the DBNE instruction, MEM will be executed one time for each fuzzy input state.

After each input and each input membership function are evaluated, the memory locations reserved for the fuzzy input states hold a value from 0x00 to 0xFF representing how much that state is active. In the example, if an IN1 value of 0x80 is given, the LOW fuzzy state would be 50% on and the HIGH fuzzy state would be 50% on. So the data array would look like this:

<i>_fuzzy_data_array</i> →	0x80	← IN1 value	Offsets from <i>fuzzy_ins</i>
	0xFF	← IN2 value	
<i>_fuzzy_ins</i> →	0x80	← LOW	← 00
	0x80	← HIGH	← 01
	0x00	← SLOW	← 02
	0xFF	← FAST	← 03
<i>_fuzzy_outs</i> →	0	← SMALL	← 04
	0	← MEDIUM	← 05
	0	← LARGE	← 06
	0	← HUGE	← 07
<i>_fuzzy_outs</i> →	0	.	
	0	.	
	0	.	

The next step in the execution is the evaluation of the rules. The *_fuzzy_temp* variable is loaded with the number of fuzzy output states. Then each of the output states is cleared to ensure that no previous data remains. The REV (Rule Evaluation) instruction begins operation here.

The REV instruction expects three things. The first is for the X register to hold the memory address of the first rule, so the *_fuzzy_ruleList* variable is used to find it. The second thing that REV expects is that the Y register holds the memory location of the fuzzy input states, which were calculated in the previous part). The last thing that REV needs is for the A register to have a 0xFF value. Calling REV will evaluate every rule.

What REV does is takes the fuzzy input state value pointed to by the first rule and stores it. It then moves to the next input and compares this to the previous one, holding onto lowest one. The REV instruction continues to move through the inputs, keep the lowest value until it reaches the 0xFE, and then stores the lowest value is found in the memory location after the 0xFE (this is the fuzzy output). When it reaches the next 0xFE, it begins looking for the lowest fuzzy input value again. It continues to do this until the final 0xFF is reached. This tells the REV instruction that there are no more rules and thus, it stops.

After the REV instruction has iterated through each rule and updated the fuzzy output values, the final step is called de-fuzzification. The key to this step is the WAV instruction. This instruction takes a weighted average of the output membership

functions, determined by the values in the fuzzy output states. Here is an example of what the WAV function will do:

These are some sample weights as calculated by the REV instruction

SMALL	0x80	=	50%
MEDIUM	0x40	=	25%
LARGE	0x00	=	0%
HUGE	0xF0	=	100%

Notice that the weights do not have to add up to 100%

These are the output membership function provided in the example:

SMALL	=	0x00
MEDIUM	=	0x30
LARGE	=	0x80
HUGE	=	0xF0

So a weighted average will multiply the corresponding values together and then divide by the total weight for an answer:

$$\frac{(50\% * 0x00) + (25\% * 0x40) + (0\% * 0x80) + (100\% * 0xF0)}{(50\% + 25\% + 100\%)}$$

The WAV instruction will calculate the numerator, and the EDIV function will find the denominator to calculate the final answer.

The WAV instruction, just like the other instructions, requires certain inputs. It needs the X register to hold the memory location of the first output membership function, and the Y register to hold the memory location of the first fuzzy output state value, which was calculated by the REV instruction. Finally, it needs the B register to hold the number of output states.

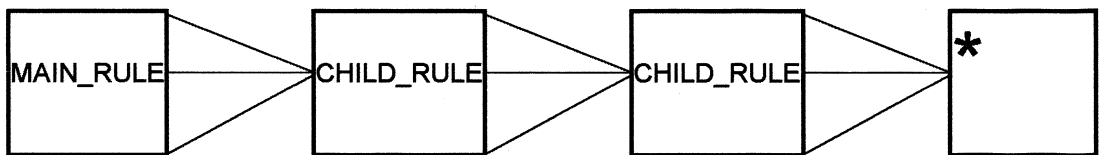
When the WAV and EDIV functions have completed, the Y register holds the final output. This is the output of the fuzzy logic system. In this code, this value is transferred to the B register so that it can be saved into a byte in the data array.

User Interface:

The purpose of the user interface is to create an easy to use method of entering the input membership functions, the output membership functions, and the rules governing them. The user interface header file also needed to provide a method to extract the entered information, and translate it into a form in which the main fuzzy logic header file can use.

The first goal is to create a structure for holding the necessary information required by the membership functions and rules. For the rules two structures were needed.

The first structure, `Main_Rule`, holds the number of total rules and a link to the second structure, `Child_Rule`. `Child_Rule` contains all the individual information for each rule including a link to an additional `Child_Rule`. This allows a linked list of all the rules, allowing any amount of rules for evaluation.



The information required for the child rule included the total number of inputs which determine the values of the output, the number of outputs effected by the inputs, and two strings which contain the input and output membership function names. The strings would be in the following format:

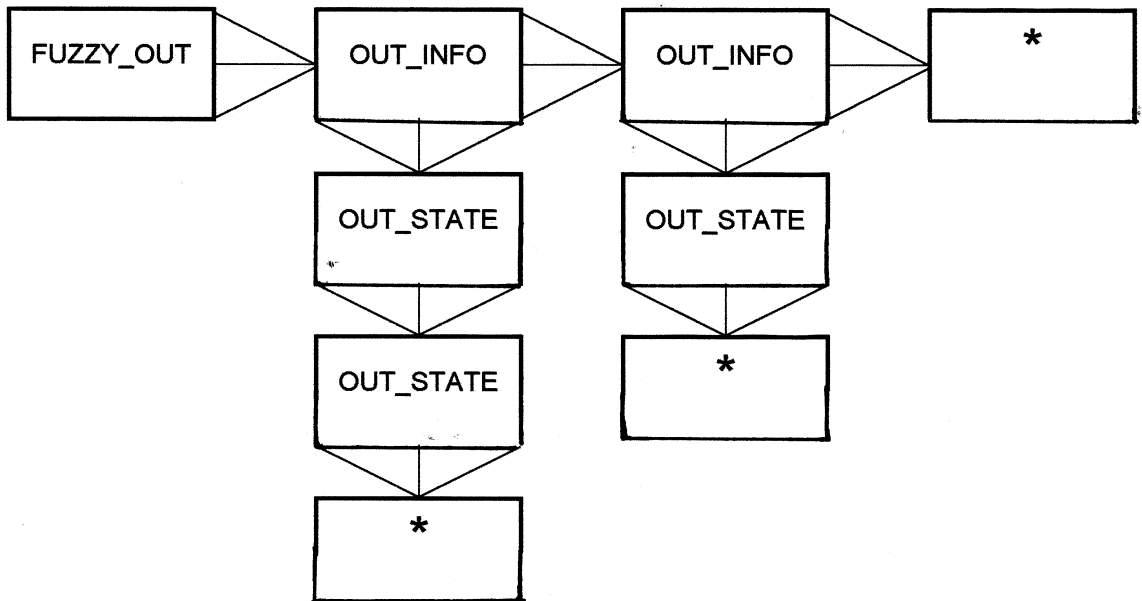
Name1_State1;Name2_State2;.....

Name1 and Name2 represent a name of an input or output, and State1 and State2 represent the name of a membership function within that input or output. The Name and State are connected with an underscore and a semicolon separates additional inputs or outputs. This also allowed for unlimited inputs or outputs required for each rule.

The second structure required held all the information required by the output membership functions. It was determined that three structures were necessary for holding this information.

The first structure, `Fuzzy_Out`, contained the number of outputs that are modified by the input and a link to the second structure, `Out_Info`. `Out_Info` contained the number

of output membership functions for that output, a name for the output, a link to another Out_Info structure, and a link to the third structure, Out_State. Out_State holds the name of the output membership function, the value that defined the output state, and a link to another Out_State structure. This structure for the output allowed for any amount of outputs as well as any amount of membership functions for each output.



The last structure needed is for the input information, and follows the same format as the output structure. The only differences are the name Fuzzy_Out, Out_Info, and Out_State are replaced with Fuzzy_In, In_Info, and In_State respectively, and the In_State structure holds different information. In_State still contains a name variable and a link to another In_State structure, but instead of holding a value, it contains two points and two slopes which determine the shape of the input membership function.

Now that we have the necessary structure for storing information, the next goal was to create function to enter the information into them. Five functions are what is needed and they are as followed:

- AddIState(char *a, char *b, char *c, char *d, char *e, char *f)

Adds an input membership function based on the passed parameters.

- "a" is the name of the input in which this membership function belongs.
- "b" is the name of the input membership function.

- “c” and “d” are two character long strings that represents the hexadecimal values for the two points.
- “e” and “f” are two character long strings that represent the hexadecimal values for the two slopes.
- **AddInput(char *a)**
Adds an input with the name “a” to the Fuzzy_In structure.
- **AddOState(char *a, char *b, char *c)**
Adds an output membership function based on the passed parameters.
 - “a” is the name of the output in which this membership function belongs.
 - “b” is the name of the output membership function.
 - “c” is a two character long string that represent the hexadecimal value for the output value.
- **AddOutput(char *a)**
Adds an output with the name “a” to the Fuzzy_Out structure
- **AddRule(int a, int b, char *c, char *d)**
Adds a rule based upon the passed parameters.
 - “a” is the number of input membership functions that drive the output membership functions.
 - “b” is the number of output membership functions that are driven by the input membership functions.
 - “c” and “d” are the strings containing the input and output information as described earlier.

With a combination of these five functions, the user can successfully enter all information required to run a fuzzy logic based program. Had this user interface been integrated into the main header file, the following code would have been used to simulate our demonstration:

```
AddInput("IN1");
AddIState("IN1","LOW","00","60","00","0B");
AddIState("IN1","HIGH","40","FF","0B","00");
AddInput("IN2");
AddIState("IN2","SLOW","00","60","00","0B");
AddIState("IN2","FAST","40","FF","0B","00");
AddOutput("SIZE");
AddOState("SIZE","SMALL","00");
AddOState("SIZE","MEDIUM","30");
```

```

AddOState("SIZE","LARGE","80");
AddOState("SIZE","HUGE","F0");
AddRule(2,1,"IN1_LOW;IN2_SLOW","SIZE_SMALL");
AddRule(2,1,"IN1_LOW;IN2_FAST","SIZE_MEDIUM");
AddRule(2,1,"IN1_HIGH;IN2_SLOW","SIZE_LARGE");
AddRule(2,1,"IN1_HIGH;IN2_FAST","SIZE_HUGE");

```

The last part required for this structure system to work is a series of functions to extract the necessary information from the three information structures. As determined by the methods used for the C implementation of the fuzzy logic program, the following function were deemed necessary:

- **GetNumOutput()**
Returns the total number of inputs driving the user system.
- **GetNumInput()**
Returns the total number of outputs needed by the user system.
- **GetNumRules()**
Returns the total number of rules governing the inputs and outputs.
- **GetNumIState(int a)**
Returns the number of input membership functions for input “a”.
- **GetNumOState(int a)**
Returns the number of output membership functions for output “a”.
- **GetInput(int a, int b)**
Loads the input membership function information for the ath input and bth state into a globally defined variable FuzzInState.
- **GetOutput(int a, int b)**
Loads the output membership function information for the ath input and bth state into a globally defined variable FuzzOutState.
- **GetRule(int a)**
Loads the ath rule information into a globally defined variable FuzzRule.
- **GetRuleIn(int a)**
Reads from FuzzRule.inval the ath input membership function, where FuzzRule.inval is the string containing the inputs for a particular rule.
- **GetRuleOut(int a)**

Reads from FuzzRule.outval the ath output membership function, where FuzzRule.outval is the string containing the outputs for a particular rule.

- GetTotalInState()

Returns the total number of input membership functions for every input.

- GetTotalOutState()

Returns the total number of output membership functions for every output.

- IndexInputState(char *a, char *b)

For the purposes of memory mapping, this determines a number based upon the two strings passed as parameters. Had we used the five input functions described earlier, the following results would occur from these commands:

IndexInputState("IN1", "LOW")	yields a value of 0
IndexInputState("IN1", "HIGH")	yields a value of 1
IndexInputState("IN2", "SLOW")	yields a value of 2
IndexInputState("IN2", "FAST")	yields a value of 3

- IndexOutputState(char *a, char *b)

For the purposes of memory mapping, this determines a number based upon the two strings passed as parameters. Had we used the five input functions described earlier, the following results would occur from these commands:

IndexOutputState("SIZE", "SMALL");	yields a value of 0
IndexOutputState("SIZE", "MEDIUM");	yields a value of 1
IndexOutputState("SIZE", "LARGE");	yields a value of 2
IndexOutputState("SIZE", "HUGE");	yields a value of 3

The indexes received from IndexInputState() and IndexOutputState() would be different if the inputs/outputs or the input/output membership functions were entered in different orders.

With the combination of these structures, the input functions, and the information grabbing functions, all that is needed to create a universal fuzzy logic header file exists.

RESULTS

Results

We had a series of goals for this project.

The first was to understand, and be able to relay the concepts behind fuzzy logic. It took us a few lab periods to research other paper on fuzzy logic, and eventually was able to create our own fuzzy logic systems.

Our second goal was to translate the assembly programs using fuzzy logic provided in the textbook into C. We used embedded some membership functions and rules into this C/assembly program to mimic a simple system, and demonstrated it perfectly.

should.
go to
intro.

Our third goal was to create a series of C procedures that would allow a normal MC68HC12 programmer to input their own membership functions and rules. Furthermore, we wrote translation functions to take the information out of the user inputted data and relay it to our original C/Assembly program with hope to integrate the two at a later point. All of these functions were tested endlessly to make sure that no flaws existed, and none were found.

Our last goal was to combine the two programs and create a generic header file that could be used by any programmer to integrate fuzzy logic into their programs. This is where our project fell short, and led to a demonstration using the hard coded membership functions and rules.


DISCUSSION

Discussion

The initial goal of this lab was to interface the fuzzy logic functionality of the HC12 for C programmers. The final design had two separately working parts. One part that implements the fuzzy logic and the other gets data from the programmer. For some reason, the two would not work together. The part that holds the process up is figuring out exactly what the C compiler is doing. An example of this is realizing that an integer variable in C is two bytes of memory, while a 'char' is only one. This is a simple example of just how well one must know the C compiler to get the memory addressing and storage to work just right. Despite hours and hours of tedious debugging (and endless irritation), the two would just not merge.

With more time perhaps the functions could be integrated slowly. Perhaps the code could be converted line by line so that the exact problem could be determined.

Some thoughts for future expansions are to use the header file as a tool to teach future classes how fuzzy logic works. Perhaps by supplying this header file and a description of its functionality.



REFERENCES

References

Texts:

Software and Hardware Engineering – Motorola M68HC12

By: Frederick M. Cady

James M. Sibigtroth

Online Material:

Intelligent Systems with Mathematica: A Team Design Workshop

<http://www.asee.org/conferences/international/proceedings/Stachowicz.pdf>

By: Marian S. Stachowicz

Christopher R. Carroll

Fuzzy Logic

http://www.egr.msu.edu/vesl/teamprojects/ee482_s98/progress/fuzzy.pdf

By: Daniel Franklin

Niki Prince

Mel Tsai

Robert Yu

APPENDIX A

MAIN HEADER FILE

```

// Main Header File for Fuzzy Logic Demo
//
// By Jeff Kornblith and Jim Harrow
//
// Provides hard coded in membership functions and rules for
// demonstration purposes.
//
// Several dummy functions were created for the ease in combining the user
// interface when the time comes.
//
// Initialize fuzzy logic program by running RunFuzzyLogic();
// Send inputs using LoadInput();
// Calculate output using RunFuzzy();
// Receive Outputs using GetResult();
//
// Usage of these function is described below.
//

#include <hc812a4.h>
#include <introl.h>
#include <debug12.h>

// Global Variables

char in1;
char in2;
char testOut;
int whyme;

int      _fuzzy_cnt1,
         _fuzzy_cnt2;
char     _fuzzy_temp;
int      _fuzzy_index = 0;

int      _fuzzy_inputMFs,
         _fuzzy_fuzzyIns,
         _fuzzy_ruleList,
         _fuzzy_fuzzyOuts,
         _fuzzy_outputMFs,
         _fuzzy_outputs;

char     _fuzzy_data_array[512];

char     look;
int      lookAt;

////////////////////////////////////
// The following functons would be replaced by the funtions in
// the user interface header file when they are readyto be combined
////////////////////////////////////

// Dummy structures to mimic user interface structures
struct whoCares
{
    char start,end,sslope,eslope;
} FuzzInState;

struct whoCares2
{
    char value;
} FuzzOutState;

struct whoCares3
{
    char numin, numout;
} FuzzRule;

// Dummy information for testing
char input1[4] = { 0x00, 0xCD, 0x00, 0x02 };
char input2[4] = { 0x40, 0xFF, 0x02, 0x00 };

```

```

char input3[4] = { 0x00, 0xC0, 0x00, 0x02 };
char input4[4] = { 0x40, 0xFF, 0x02, 0x00 };

char GetNumInput()
{
    // Dummy information for testing
    return 2;
}

char GetNumIState(char num)
{
    // Dummy information for testing
    return 2;
}

void GetInput(char num, char num1)
{
    char * input;

    // Dummy information for testing
    if( num==0 && num1==0 ) input = input1;
    if( num==0 && num1==1 ) input = input2;
    if( num==1 && num1==0 ) input = input3;
    if( num==1 && num1==1 ) input = input4;

    FuzzInState.start = input[0];
    FuzzInState.end   = input[1];
    FuzzInState.sslope = input[2];
    FuzzInState.eslope = input[3];
}

char GetNumOutput()
{
    // Dummy information for testing
    return 1;
}

char GetNumOState(char num)
{
    // Dummy information for testing
    return 4;
}

void GetOutput(char num, char num1)
{
    // Dummy information for testing
    if( num==0 && num1==0 ) FuzzOutState.value = 0x00;
    if( num==0 && num1==1 ) FuzzOutState.value = 0x30;
    if( num==0 && num1==2 ) FuzzOutState.value = 0x80;
    if( num==0 && num1==3 ) FuzzOutState.value = 0xF0;
}

char GetNumRule()
{
    // Dummy information for testing
    return 4;
}

void GetRule( char num )
{
    // Dummy information for testing
    return;
}

char GetRuleIn( char rule, char input )
{
    // Dummy information for testing
    char array[4][2] = { 0x00, 0x02, 0x00, 0x03, 0x01, 0x02, 0x01, 0x03 };

    return array[rule][input];
}

```



```

char GetRuleOut( char rule, char output )
{
    // Dummy information for testing
    char array[4][1] = { 0x04, 0x05, 0x06, 0x07 };

    return array[rule][output];
}

char GetTotalNumIn()
{
    // Dummy information for testing
    return 4;
}

char GetTotalNumOut()
{
    // Dummy information for testing
    return 4;
}

////////////////////////////////////
////////////////////////////////////

// Prototypes
void RunFuzzyLogic();
void RunFuzzy();
void LoadInput( int , char );
char GetResult( int );

// Receives input from user program
// in - the input number that this information is for
// num - the hexadecimal value that this input should be set to
void LoadInput( int in, char num)
{
    _fuzzy_data_array[ in ] = num;
}

// Sends output desired to user program
// out - the output number that the user wants
char GetResult( int out )
{
    return _fuzzy_data_array[ _fuzzy_outputs + out ];
}

// Debug function to show memory locations
void showDataArray( int upTo , int pause)
{
    int i;

    for( i=0; i<upTo; i++)
    {
        DB12->printf( "%x\r %d --> %x\n\r", i, _fuzzy_data_array[i] );
        if( pause && ( i % 15 == 0 ) )
        {
            DB12->getchar();
        }
    }
}

// Initialization Function
void RunFuzzyLogic()
{
    FuzzInState.start = 0x40;
    FuzzInState.end = 0xC0;
    FuzzInState.sslope = 0x0B;
    FuzzInState.eslope = 0x0B;

    FuzzOutState.value = 0x80;

```

```

FuzzRule.numin = 2;
FuzzRule.numout = 1;

//*****
//* Make room for the real inputs
//*****
_fuzzy_index += GetNumInput();

//*****
//* Make room for the fuzzy inputs
//*****
// Store fuzzyins location
_fuzzy_fuzzyIns = _fuzzy_index;

_fuzzy_index += GetTotalNumIn();

// *****
// * Make room for the fuzzy outputs
// *****/
_fuzzy_fuzzyOuts = _fuzzy_index;
_fuzzy_index += GetTotalNumOut();

// *****
// * Make room for the real outputs
// *****
_fuzzy_outputs = _fuzzy_index;
_fuzzy_index += GetNumOutput();

// *****
// * Write the input membership functions *
// *****/
// the beginning of the MFS
_fuzzy_inputMFs = _fuzzy_index;

for( _fuzzy_cnt1=0; _fuzzy_cnt1<GetNumInput(); _fuzzy_cnt1++)
{
    for( _fuzzy_cnt2=0; _fuzzy_cnt2<GetNumIState(_fuzzy_cnt1); _fuzzy_cnt2++ )
    {
        GetInput( _fuzzy_cnt1, _fuzzy_cnt2 );

        _fuzzy_data_array[_fuzzy_index] = FuzzInState.start;
        _fuzzy_index++;
        _fuzzy_data_array[_fuzzy_index] = FuzzInState.end;
        _fuzzy_index++;
        _fuzzy_data_array[_fuzzy_index] = FuzzInState.sslope;
        _fuzzy_index++;
        _fuzzy_data_array[_fuzzy_index] = FuzzInState.eslope;
        _fuzzy_index++;
    }
}

// *****
// * Write the rule list
// *****/
_fuzzy_ruleList = _fuzzy_index;

for( _fuzzy_cnt1=0; _fuzzy_cnt1<GetNumRule(); _fuzzy_cnt1++)
{
    GetRule( _fuzzy_cnt1 );

    for( _fuzzy_cnt2=0; _fuzzy_cnt2<FuzzRule.numin; _fuzzy_cnt2++)
    {
        _fuzzy_data_array[_fuzzy_index] = GetRuleIn( _fuzzy_cnt1, _fuzzy_cnt2 );
        _fuzzy_index++;
    }

    _fuzzy_data_array[_fuzzy_index] = 0xFE;
    _fuzzy_index++;

    for( _fuzzy_cnt2=0; _fuzzy_cnt2<FuzzRule.numout; _fuzzy_cnt2++)

```

```

        {
            _fuzzy_data_array[_fuzzy_index] = GetRuleOut( _fuzzy_cnt1, _fuzzy_cnt2 );
            _fuzzy_index++;
        }

        if( _fuzzy_cnt1 != GetNumRule()-1 )
        {
            _fuzzy_data_array[_fuzzy_index] = 0xFE;
            _fuzzy_index++;
        }
    }

    _fuzzy_data_array[_fuzzy_index] = 0xFF;
    _fuzzy_index++;

// *****
// *   Write the output membership functions
// *****
    _fuzzy_outputMFs = _fuzzy_index;

    for( _fuzzy_cnt1=0; _fuzzy_cnt1<GetNumOutput(); _fuzzy_cnt1++)
    {
        for( _fuzzy_cnt2=0; _fuzzy_cnt2<GetNumOState(_fuzzy_cnt1); _fuzzy_cnt2++ )
        {
            GetOutput( _fuzzy_cnt1, _fuzzy_cnt2 );
            DB12->printf( "%x\r cnt1: %x cnt2: %x\n\r", _fuzzy_cnt1, _fuzzy_cnt2 );
            _fuzzy_data_array[_fuzzy_index] = FuzzOutState.value;
            _fuzzy_index++;
        }
    }
}

// Run this to calculate values based on inputs
void RunFuzzy()
{
// *****
// *   Import all the needed variables
// *****
    asm(" import look_testOut");
    asm(" import _fuzzy_cnt1, _fuzzy_temp");
    asm(" import _fuzzy_data_array");
    asm(" import _fuzzy_inputMFs, _fuzzy_fuzzyIns, _fuzzy_ruleList");
    asm(" import _fuzzy_fuzzyOuts, _fuzzy_outputMFs, _fuzzy_outputs");
// *****
// *   BEGIN WORK
// *****

    asm("START: EQU *");

// *****
// *   FUZZIFY
// *****

    asm("FUZZIFY: EQU *");

    asm(" LDD _fuzzy_inputMFs");
    asm(" LDX #_fuzzy_data_array");
    asm(" LEAX D,X");

    asm(" LDD _fuzzy_fuzzyIns");
    asm(" LDY #_fuzzy_data_array");
    asm(" LEAY D,Y");

    for( _fuzzy_cnt1=0; _fuzzy_cnt1<GetNumInput(); _fuzzy_cnt1++)
    {
        _fuzzy_temp = GetNumIState( _fuzzy_cnt1 );
        asm(" PSHX");
        asm(" LDD _fuzzy_cnt1");
        asm(" LDX #_fuzzy_data_array");
        asm(" LEAX D,X");
    }
}

```

```

asm(" LDAA D1X");
asm(" STAA look");
asm(" PULX");
asm(" LDAB _fuzzy_temp");
asm("FUZL00P: EQU *");
asm(" MEM");
asm(" DBNE B1FUZL00P");
}

```

```

_fuzzy_temp = GetTotalNumOut();

```

```

// ** Clear all the inputs
asm(" LDAB _fuzzy_temp"); // <- Number of output variables

```

```

asm("RULEEVAL:");
asm(" CLR I1Y+");
asm(" DBNE B1RULEEVAL");

asm(" LDD _fuzzy_ruleList");
asm(" LDY #_fuzzy_data_array");
asm(" LEAX D1X");

```

```

asm(" LDD _fuzzy_fuzzyIns");
asm(" LDY #_fuzzy_data_array");
asm(" LEAY D1Y");

```

```

asm(" LDAA #FF");
asm(" REV");

```

```

_fuzzy_temp = GetTotalNumOut();

```

```

asm("DEFUZ:");
asm(" LDD _fuzzy_outputMFs");
asm(" LDY #_fuzzy_data_array");
asm(" LEAX D1X");

```

```

asm(" LDD _fuzzy_fuzzyOuts");
asm(" LDY #_fuzzy_data_array");
asm(" LEAY D1Y");

```

```

asm(" LDAB _fuzzy_temp"); // <- All outputs need same number of fuzzyOuts
asm(" WAV");

```

```

asm(" EDIV");

```

```

asm(" LDD _fuzzy_outputs");
asm(" LDY #_fuzzy_data_array");
asm(" LEAX D1X");

```

```

asm(" TFR Y1D");
asm(" STAB D1X");
asm(" STAB testOut");

```

```

asm("END: EQU *");

```

```

}

```

APPENDIX B

USER INTERFACE HEADER FILE

```

// UserInterface.h
//
// Written by Jim Harrow and Jeff Kornblith
//
// Included functions for the universal fuzzy logic header file
// for the purpose of entering input/output/rules for fuuzzy logic,
// as well as the translation functions to integrate into the hardcoded
// header file.

#include <hc812a4.h>
#include <introl.h>
#include <debug12.h>

#define NULL '\0'

/////////////////////////////////////////////////////////////////
// Structures for the rules
struct child_rule
{
    int          numin;          // number of inputs deciding output
    int          numout;         // number of outputs effected by input
    char         *inval;        // Input string
    char         *outval;       // Output string
    struct child_rule *next;    // next rule
};

struct main_rule
{
    int          numrules;       // Number of rules
    struct child_rule rule;      // rules
};

/////////////////////////////////////////////////////////////////
// Structures for the fuzzy outputs
struct out_state
{
    char         *name;         // Name of State
    char         *value;        // state value
    struct out_state *next;     // next state associated with this output
};

struct out_info
{
    char         *name;         // name of output
    int          numstates;     // Number of of OUTPUTMFs for this output
    struct out_state state;     // states
    struct out_info *next;      // next output
};

struct fuzzy_out
{
    int          numout;        // number of outputs
    struct out_info *next;      // first output
};

/////////////////////////////////////////////////////////////////
// Structures for the fuzzy inputs
struct in_state
{
    char         *name;         // Name of State
    char         *start;        // Point 1
    char         *sslope;       // Slope 1
    char         *end;          // Point 2
    char         *eslope;       // Slope 2
    struct in_state *next;      // next state associated with this input
};

struct in_info
{
    char         *name;         // Name of Input
    int          numstates;     // Number of INPUTMFs for this Input
};

```

```

    struct in_state      state;           // states
    struct in_info      *next;           // next input
};

struct fuzzy_in
{
    int                  numin;           // number of inputs
    struct in_info      *next;           // first input
};

////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

// Various String Manipulation Functions
int  slen(char*);
char* scopy(char*);
int  scomp(char*,char*);

// Function (and helper) that translates a two character ascii string into
// the appropriate hex value.
int  helpctohex(char);
int  ctohex(char*);

// Initialization function to set up structures
void InitializeFuzzy();

// Helper functions to search if an input exists for adding states
int  SearchInput(char *name);
int  SearchOutput(char *name);

// The five fucntions the user must call to successfully enter the
// fuzzy logic information
int  AddIState(char*,char*,char*,char*,char*,char*);
int  AddInput(char*);
int  AddOState(char*,char*,char*);
int  AddOutput(char*);
void AddRule(int,int,char*,char*);

// Various data grabbing fucntions for integration with c-implementation
// of fuzzy logic
int  GetNumInput();
int  GetNumOutput();
int  GetNumRules();
int  GetNumIState(int);
int  GetNumOState(int);

// Loads an input member fucntion, an output member fucntion, or a rule
// into a global structure for data grabbing. This is for the integration
// with the other header file.
int  GetInput(int,int);
int  GetOutput(int,int);
int  GetRule(int);

// Grabs a separates a certain input member function name from the input/output
// information in the gloabl rule structure
char* GetRuleIn(int);
char* GetRuleOut(int);

// Data grabbing fucntion for the other header file
int  GetTotalInState();
int  GetTotalOutState();

// Indexing functions for the other header file
int  InputNum( char * );
int  InputStateNum( int, char * );
int  IndexInputState( char *, char * );
int  OutputNum( char * );
int  OutputStateNum( int, char * );
int  IndexOutputState( char *, char * );

////////////////////////////////////////////////////////////////

```

//

// Global Variables

```
struct fuzzy_in  fuzzyin;
struct fuzzy_out fuzzyout;
struct main_rule rulelist;
```

```
struct in_state  FuzzInState;
struct out_state FuzzOutState;
struct child_rule FuzzRule;
```

//
//

// Returns length of 'name'

int slen(char *name)

```
{
    int x=0;

    while ( name[x]!=NULL ) x++;

    return x;
}
```

// Returns a copy of 'name'

char* scopy(char *name)

```
{
    int x=0;
    char *buffer;

    buffer= malloc( slen(name) +1);

    for ( x=0 ; x<slen(name) ; x++ ) buffer[x]=name[x];

    buffer[x]=NULL;

    return buffer;
}
```

// Compares 'a' and 'b'... return 0 if not equal and 1 if equal

int scomp(char *a, char *b)

```
{
    int x;
    int lenA = 0, lenB = 0;
    char temp1, temp2;

    lenA = slen(a);
    lenB = slen(b);

    if ( lenA != lenB ) return 0;

    for ( x=0 ; x<lenA ; x++ )
    {
        temp1 = a[x]; temp2=b[x];

        if ( temp1 != temp2 ) return 0;
    }

    return 1;
}
```

// Helper function for 'int ctohex(char*)'

// Returns the appropriate number for each hexadecimal character 'a'.

// Returns a -1 if bad data was passed to 'a'.

int helpctohex(char a)

```
{
    switch (a)
    {
        case '0': return 0;
        case '1': return 1;
```



```

    case '2': return 2;
    case '3': return 3;
    case '4': return 4;
    case '5': return 5;
    case '6': return 6;
    case '7': return 7;
    case '8': return 8;
    case '9': return 9;
    case 'A': return 10;
    case 'B': return 11;
    case 'C': return 12;
    case 'D': return 13;
    case 'E': return 14;
    case 'F': return 15;
    default : return -1;
}

```

```

}

```

```

// Converts a 2 character hex value to an actual number ( "FF" to 255 )

```

```

// return -1 if error

```

```

int ctohex(char *val)

```

```

{
    int temp;
    char a=val[0];
    char b=val[1];

    if ( strlen(val)!=2 ) return -1;

```

```

    if ( helpctohex(a) ) temp=helpctohex(a)*16;
    else return -1;

```

```

    if ( helpctohex(b) ) return temp+helpctohex(b);
    else return -1;
}

```

```

// Initialization function for the user implementation

```

```

//   Initializes the three global structures for data insertion

```

```

void InitializeFuzzy()

```

```

{
    fuzzyin.numin=0;
    fuzzyin.next=NULL;

    fuzzyout.numout=0;
    fuzzyout.next=NULL;

    rulelist.numrules=0;
    rulelist.rule.numin=0;
    rulelist.rule.numout=0;
    rulelist.rule.inval=NULL;
    rulelist.rule.outval=NULL;
    rulelist.rule.next=NULL;
}

```

```

// Helper function for "AddInput()" and "AddIState()"

```

```

//   Returns 1 if 'name' exists on the input list.

```

```

//   Returns 0 if not.

```

```

int SearchInput(char *name)

```

```

{
    int x=0;
    struct in_info *a;

    if ( fuzzyin.numin==0 ) return 0;

```

```

    a=fuzzyin.next;

```

```

    while ( x<fuzzyin.numin )
    {

```

```

        if ( strcmp(a->name,name) ) return 1;
        a=a->next;
        x++;
    }
}

```

```

    return 0;
}

// Helper function for "AddOutput()" and "AddState()"
// Returns 1 if 'name' exists on the input list.
// Returns 0 if not.
SearchOutput(char *name)
{
    int x=0;
    struct out_info *a;

    if ( fuzzyout.numout==0 ) return 0;

    a=fuzzyout.next;

    while ( x<fuzzyout.numout )
    {
        if ( scomp(a->name,name) ) return 1;
        a=a->next;
        x++;
    }

    return 0;
}

// Add Input State (Membership Function)
// name = name of input that membership function is related to
// a1 = name of membership function
// a2 = point 1
// a3 = point 2
// a4 = slope 1
// a5 = slope 2
//
// This function adds a state (membership function) to the input structures.
// Returns -1 on error
int AddIState(char *name, char *a1, char *a2, char *a3, char *a4, char *a5)
{
    struct in_info *input=NULL;
    struct in_state *state=NULL;
    int x;
    int good=1;

    if ( !SearchInput(name) ) return -1;

    input = fuzzyin.next;

    while ( (input != NULL) && (!scomp(name,input->name)) )
    {
        input=input->next;
    }

    input->numstates = input->numstates+1;

    if ( input->numstates==1 )
    {
        input->state.name=scopy(a1);
        input->state.start=scopy(a2);
        input->state.end=scopy(a3);
        input->state.sslope=scopy(a4);
        input->state.eslope=scopy(a5);
        input->state.next=NULL;
        return 1;
    }

    if ( input->numstates==2 )
    {
        input->state.next= malloc (sizeof (struct in_state) );
        input->state.next->name=scopy(a1);
        input->state.next->start=scopy(a2);
        input->state.next->end=scopy(a3);
        input->state.next->sslope=scopy(a4);

```

```

        input->state->next->eslope=scopy(a5);
        input->state->next->next = NULL;
        return 1;
    }

    state=input->state->next;

    while ( state->next != NULL ) state=state->next;

    state->next=malloc (sizeof (struct in_state));
    state = state->next;

    state->name=scopy(a1);
    state->start=scopy(a2);
    state->end=scopy(a3);
    state->sslope=scopy(a4);
    state->eslope=scopy(a5);
    state->next = NULL;
    return 1;
}

// Add Input
//   name - name of input to be added
//
//   If 'name' does not already exist on the input list, it is added
//   and appropriate values are initialized
int AddInput(char *name)
{
    int x;
    struct in_info *a;
    int good=1;

    if ( SearchInput(name) ) return -1;

    fuzzyin.numin++;

    if ( fuzzyin.numin==1 )
    {
        fuzzyin.next= malloc (sizeof (struct in_info));
        fuzzyin.next->name=scopy(name);
        fuzzyin.next->numstates=0;
        fuzzyin.next->state.start=NULL;
        fuzzyin.next->state.end=NULL;
        fuzzyin.next->state.sslope=NULL;
        fuzzyin.next->state.eslope=NULL;
        fuzzyin.next->state.next=NULL;
        fuzzyin.next->next = NULL;
        return 1;
    }

    a=fuzzyin.next;
    x=fuzzyin.numin;

    while ( good )
    {
        x=x-1;
        if ( x==1 )
        {
            a->next= malloc (sizeof (struct in_info));
            good=0;
        }
        a=a->next;
    }

    a->name=scopy(name);
    a->numstates=0;
    a->state.start=NULL;
    a->state.end=NULL;
    a->state.sslope=NULL;
    a->state.eslope=NULL;
    a->state.next=NULL;

```

```

    return 1;
}

// Add Output State (Membership Function)
// name = name of output that membership function is related to
// a1 = name of membership function
// a2 = point 1
//
// This function adds a state (membership function) to the output structures.
// Returns -1 on error
int Add0State(char *name, char *a1, char *a2)
{
    struct out_info *output=NULL;
    struct out_state *state=NULL;
    int x;
    int good=1;

    if ( !Search0Output(name) ) return -1;

    output = fuzzyout.next;

    while ( !scomp(name,output->name) )
    {
        output=output->next;
    }

    output->numstates = output->numstates+1;

    if ( output->numstates==1 )
    {
        output->state.name=scopy(a1);
        output->state.value=scopy(a2);
        return 1;
    }

    if ( output->numstates==2 )
    {
        output->state.next=malloc (sizeof (struct out_state));
        output->state.next->name=scopy(a1);
        output->state.next->value=scopy(a2);
        return 1;
    }

    state=output->state.next;

    x=output->numstates;
    while ( good )
    {
        x=x-1;
        if ( x==2 )
        {
            state->next=malloc (sizeof (struct out_state));
            good=0;
        }
        state=state->next;
    }
    state->name=scopy(a1);
    state->value=scopy(a2);
    return 1;
}

// Add Output
// name - name of output to be added
//
// If 'name' does not already exist on the output list, it is added
// and appropriate values are initialized
int Add0Output(char *name)
{
    int x;
    struct out_info *a;
    int good=1;

```

```

if ( SearchOutput(name) ) return -1;

fuzzyout.numout++;

if ( fuzzyout.numout==1 )
{
    fuzzyout.next=malloc (sizeof (struct out_info));
    fuzzyout.next->name=scopy(name);
    fuzzyout.next->numstates=0;
    fuzzyout.next->state.value=NULL;
    fuzzyout.next->state.next=NULL;
    return 1;
}

a=fuzzyout.next;
x=fuzzyout.numout;

while ( good )
{
    x=x-1;
    if ( x==1 )
    {
        a->next= malloc (sizeof (struct out_info));
        good=0;
    }
    a=a->next;
}
a->name=scopy(name);
a->numstates=0;
a->state.value=NULL;
a->state.next=NULL;
return 1;
}

// Add Rule
// i1 = number of inputs that drive the output
// i2 = number of outputs driven by the input
// a1 = input string
// a2 = output string
//
// This function adds a rule to the rule list.
//
// The input (and output) strings are in the following format:
//
// INPUTNAME1_STATENAME1;INPUTNAME2_STATENAME
//
// where '_' link the input(or output) name and state name and ';'
// separate multiple inputs(or outputs)
void AddRule(int i1, int i2, char *a1, char *a2)
{
    struct child_rule *rule=NULL;
    int x;
    int good=1;

    rulelist.numrules+=1;

    if ( rulelist.numrules>1 )
    {
        if ( rulelist.numrules==2 )
        {
            rulelist.rule.next= malloc (sizeof (struct child_rule));
            good=0;
        }

        rule=rulelist.rule.next;

        x=rulelist.numrules;
        while ( good )
        {
            x=x-1;

```

```

        if ( x==2 )
        {
            rule->next= malloc (sizeof (struct child_rule));
            good=0;
        }
        rule=rule->next;
    }
    rule->numin=i1;
    rule->numout=i2;
    rule->inval=scopy(a1);
    rule->outval=scopy(a2);
}
else
{
    rulelist.rule.numin=i1;
    rulelist.rule.numout=i2;
    rulelist.rule.inval=scopy(a1);
    rulelist.rule.outval=scopy(a2);
}
}

// Returns the number of inputs
int GetNumInput()
{
    return fuzzyin.numin;
}

// Returns number of outputs
int GetNumOutput()
{
    return fuzzyout.numout;
}

// Returns number of rules
int GetNumRules()
{
    return rulelist.numrules;
}

// Returns number of states for input #'a'
int GetNumIState(int a)
{
    int x=1;
    struct in_info *b;

    if ( a>GetNumInput() ) return 0;

    b=fuzzyin.next;

    while ( x < a )
    {
        b=b->next;
        x++;
    }
    return b->numstates;
}

// Returns number of states for output #'a'
int GetNumOState(int a)
{
    int x=0;
    struct out_info *b;

    if ( a>GetNumOutput() ) return 0;

    b=fuzzyout.next;

    while ( x < a-1 )
    {
        b=b->next;

```

```

    x++;
}

return b->numstates;

```

```

}

```

```

// Loads input 'a', state 'b' into the global structure 'FuzzInState'
int GetInput(int a,int b)

```

```

{
    int x=1;
    struct in_info *big;
    struct in_state *lit;

    if ( a>GetNumInput() ) return 0;
    if ( b>GetNumIState(a) ) return 0;

    big=fuzzyin.next;

    while ( x < a )
    {
        big=big->next;
        x++;
    }

    x=1;

    *lit=big->state;
    while ( x < b )
    {
        lit=lit->next;
        x++;
    }
    FuzzInState.name=scopy(lit->name);
    FuzzInState.start=scopy(lit->start);
    FuzzInState.end=scopy(lit->end);
    FuzzInState.sslope=scopy(lit->sslope);
    FuzzInState.eslope=scopy(lit->eslope);

    return 1;
}

```

```

}

```

```

// Loads output 'a', state 'b' into the global structure "FuzzOutState"
int GetOutput(int a,int b)

```

```

{
    int x=1;
    struct out_info *big;
    struct out_state *lit;

    if ( a>GetNumOutput() ) return 0;
    if ( b>GetNumOState(a) ) return 0;

    big=fuzzyout.next;

    while ( x < a )
    {
        big=big->next;
        x++;
    }

    x=1;

    *lit=big->state;
    while ( x < b )
    {
        lit=lit->next;
        x++;
    }
    FuzzOutState.name=scopy(lit->name);
    FuzzOutState.value=scopy(lit->value);
}

```

```

    return 1;
}

// Loads rule 'a' into the global structure "FuzzRlue"
int GetRule(int a)
{
    int x=0;
    struct child_rule *rule;

    if ( a>GetNumRules() ) return 0;

    if ( a==1 )
    {
        FuzzRule.numin=rulelist.rule.numin;
        FuzzRule.numout=rulelist.rule.numout;
        FuzzRule.inval=scopy(rulelist.rule.inval);
        FuzzRule.outval=scopy(rulelist.rule.outval);
        return 0;
    }

    rule=rulelist.rule.next;
    if ( a!=2 ) while ( x < a-2)
    {
        rule=rule->next;
        x++;
    }

    FuzzRule.numin=rule->numin;
    FuzzRule.numout=rule->numout;
    FuzzRule.inval=scopy(rule->inval);
    FuzzRule.outval=scopy(rule->outval);
    return 0;
}

// Parses the string "FuzzRule.inval" and returns the 'a'th INPUTNAME_STATENAME
// string. GetRule(n) must be called prior to calling this function
char* GetRuleIn(int a)
{
    char *buffer;
    int x=1;
    int y=0;
    int d=0;
    if ( a>GetNumRules() ) return NULL;

    buffer=malloc( slen(a) + 1);

    while ( x!=a )
    {
        if ( FuzzRule.inval[y]==';' )
        {
            y++;
            x++;
        }
        else y++;
    }
    while ( FuzzRule.inval[y]!=';' && y<slen(FuzzRule.inval) )
    {
        buffer[d]=FuzzRule.inval[y];
        d++;
        y++;
    }
    buffer[d]=NULL;
    return buffer;
}

// Parses the string "FuzzRule.outval" and returns the 'a'th OUTPUTNAME_STATENAME
// string. GetRule(n) must be called prior to calling this function
char* GetRuleOut(int a)
{
    char *buffer;

```



```

int x=1;
int y=0;
int d=0;
if ( a>GetNumRules() ) return NULL;

buffer=malloc( slen(a) + 1);

while ( x!=a )
{
    if ( FuzzRule.outval[y]!=';' )
    {
        y++;
        x++;
    }
    else y++;
}
while ( FuzzRule.outval[y]!=';' && y<slen(FuzzRule.outval) )
{
    buffer[d]=FuzzRule.outval[y];
    d++;
    y++;
}
buffer[d]=NULL;
return buffer;
}

// Returns the total number of input states (membership functions)
int GetTotalInState()
{
    struct in_info *a;
    int x;
    int total=0;

    a=fuzzyin.next;

    for ( x=0 ; x<fuzzyin.numin ; x++ )
    {
        total=total+a->numstates;
        a=a->next;
    }
    return total;
}

// Returns the total number of output states (membership functions)
int GetTotalOutState()
{
    struct out_info *a;
    int x;
    int total=0;

    a=fuzzyout.next;

    for ( x=0 ; x<fuzzyout.numout ; x++ )
    {
        total=total+a->numstates;
        a=a->next;
    }
    return total;
}

// Returns the index of Input 'in', state 'state' for the purpose of
// memory mapping when integrated with the other header file.
int IndexInputState( char * in, char * state)
{
    int index = 0;
    int input = 1;
    int inputNum = InputNum( in );
    int stateNum = InputStateNum( inputNum, state );

    for( input=1; input < inputNum; input++ )

```

```

        index += GetNumIState( inputNum );

    index += stateNum-1;

    return index;
}

// Helper function for 'IndexInputState'
// Returns the index for input name 'input'
int InputNum( char * input )
{
    int x=1;
    struct in_info *big;

    big=fuzzyin.next;

    while ( big != NULL && !scomp(input, big->name) )
    {
        big=big->next;
        x++;
    }

    if( big == NULL ) return -1;

    return x;
}

// Helper function for "IndexInputState"
// Returns the index for input number 'input', state name 'state'
int InputStateNum( int input, char * state )
{
    int x=1,y=0;
    int numStates = GetNumIState( input );
    struct in_info *big;
    struct in_state *lit;
    char *name;

    if ( input > GetNumInput() ) return -2;

    big=fuzzyin.next;

    while ( x < input )
    {
        big=big->next;
        x++;
    }

    y=1;

    if( !scomp(big->state.name, state) )
    {
        y++;
        lit=big->state.next;
        while ( y<=numStates && !scomp( lit->name, state) )
        {
            lit=lit->next;
            y++;
        }
    }

    if( y>numStates ) return -1;

    return y;
}

// Returns the index of Output 'out', state 'state' for the purpose of
// memory mapping when integrated with the other header file.
int IndexOutputState( char * out, char * state)
{

```

```

int index = 0;
int output = 1;
int outputNum = OutputNum( out );
int stateNum = OutputStateNum( outputNum, state );

for( output=1; output < outputNum; output++ )
    index += GetNumOfState( outputNum );

index += stateNum-1;

return index;
}

// Helper function for 'IndexOfOutputState'
// Returns the index for output name 'output'
int OutputNum( char * output )
{
    int x=1;
    struct out_info *big;

    big=fuzzyout.next;

    while ( big != NULL && !scomp(output, big->name) )
    {
        big=big->next;
        x++;
    }

    if( big == NULL ) return -1;

    return x;
}

// Helper function for "IndexOfOutputState"
// Returns the index for output number 'output', state name 'state'
int OutputStateNum( int output, char * state )
{
    int x=1,y=0;
    int numStates = GetNumOfState( output );
    struct out_info *big;
    struct out_state *lit;
    char *name;

    if ( output > GetNumOfOutput() ) return -2;

    big=fuzzyout.next;

    while ( x < output )
    {
        big=big->next;
        x++;
    }

    y=1;

    if( !scomp(big->state.name, state) )
    {
        y++;

        lit=big->state.next;

        while ( y<=numStates && !scomp( lit->name, state) )
        {
            lit=lit->next;
            y++;
        }
    }

    if( y>numStates ) return -1;

    return y;
}

```

APPENDIX C

DEMONSTRATION PROGRAM

```

// James Jarrow & Jeff Kornblith
// Microprocessor Systems
//
// Test Program for Fuzzy Logic Demonstration
//
// Uses two Potentiometers connected to PAD1 & PAD2 for analog input
//
// Uses PORTS H & J for LED Output Display

#include <hc812a4.h>
#include <introl.h>
#include <debug12.h>

#include "fuzzylogic.h"

void __mod2__ ADInt();
void DisplayOutput(int);

unsigned int val2, val1;

void __main()
{
    int cnt1;

    // AtoD Initialization
    DB12->SetUserVector( AtoD, ADInt );
    _H12ADTCTL2 = (_H12ADTCTL2 | 0x82);
    _H12ADTCTL4 = 0x01;
    _H12ADTCTL5 = 0x10;

    // Port Initialization
    _H12DDRH = 0xFF;
    _H12DDRJ = 0xFF;

    DB12->printf("\033[2J");

    // Initializes Fuzzy Logic
    RunFuzzyLogic();
    while ( 1 )
    {
        // Load AtoD Input
        LoadInput( 0, val1 );
        LoadInput( 1, val2 );

        // Calculate Output
        RunFuzzy();

        // Display Output of LEDs
        DisplayOutput( GetResult( 0 ) );

        DB12->printf( "%xInputs: %x %x\n\r", val1, val2 );
        DB12->printf( "%xOutput: %x\n\r", GetResult( 0 ) );
    }
}

// AtoD Interrupt Code
void __mod2__ ADInt()
{
    if((_H12ADTSTAT & 0x06) == 0x06)
    {
        val2 = _H12ADR2H;
        val1 = _H12ADR1H;
        _H12TSCR = 0x80;
    }
    _H12ADTCTL5 = 0x10;
}

// Ugly LED output display code
void DisplayOutput(int val)
{
    _H12PORTJ=0xFF;

```

_H12P0RTH=0xFF;

```
if ( val > 8 ) { _H12P0RTJ=0xFE; _H12P0RTH=0xFF; }
if ( val > 24 ) { _H12P0RTJ=0xFC; _H12P0RTH=0xFF; }
if ( val > 40 ) { _H12P0RTJ=0xF8; _H12P0RTH=0xFF; }
if ( val > 56 ) { _H12P0RTJ=0xF0; _H12P0RTH=0xFF; }
if ( val > 72 ) { _H12P0RTJ=0xED; _H12P0RTH=0xFF; }
if ( val > 88 ) { _H12P0RTJ=0xC0; _H12P0RTH=0xFF; }
if ( val > 104 ) { _H12P0RTJ=0x80; _H12P0RTH=0xFF; }
if ( val > 120 ) { _H12P0RTJ=0x00; _H12P0RTH=0xFF; }
```

```
if ( val > 136 ) { _H12P0RTH=0xFE; _H12P0RTJ=0x00; }
if ( val > 152 ) { _H12P0RTH=0xFC; _H12P0RTJ=0x00; }
if ( val > 168 ) { _H12P0RTH=0xF8; _H12P0RTJ=0x00; }
if ( val > 184 ) { _H12P0RTH=0xF0; _H12P0RTJ=0x00; }
if ( val > 200 ) { _H12P0RTH=0xED; _H12P0RTJ=0x00; }
if ( val > 216 ) { _H12P0RTH=0xC0; _H12P0RTJ=0x00; }
if ( val > 232 ) { _H12P0RTH=0x80; _H12P0RTJ=0x00; }
if ( val > 248 ) { _H12P0RTH=0x00; _H12P0RTJ=0x00; }
```

APPENDIX D

DEMONSTRATION CIRCUIT DIAGRAM

Circuit Diagram for Demonstration

