AN1225

Fuzzy Logic and the NEURON CHIP

INTRODUCTION

The world of embedded controls is currently experiencing a push into the realm of fuzzy logic. In the past, manufacturers have contended with performance vs. cost tradeoffs with no apparent fulfillment of both. However, the concept of fuzzy logic has repeatedly proven that for some applications a low cost, 8–bit microcontroller can equal or exceed the performance of a more expensive *number crunching* DSP (digital signal processor). Motorola has recognized the power of fuzzy logic and has created fuzzy kernels and support tools for a number of their microcontrollers including the MC143150/20 (NEURON CHIP).

The NEURON CHIP is a communications and control processor (designed by Echelon, manufactured by Motorola) with an embedded LONTALK protocol used for multimedia networking environments in which received network inputs (on the processor's communications port) are used to control processor outputs (on its I/O port). An important design concept relevant to NEURON CHIPS is **intelligent distributed control** — the distribution of control among several NEURON processors (called nodes) which share I/O data on a network. Specifically, in fuzzy applications input data can be sent via a network to a "fuzzy" node which will run the inputs through its fuzzy engine and control its outputs accordingly. This application note will give the reader a brief introduction to 8–bit fuzzy logic, present a fuzzy kernel for the NEURON CHIP practical for a 30 Hz controller, and demonstrate a fuzzy node in a fan controller application. Finally, refer to Motorola's data book (MC143150/D) for technical information on the NEURON CHIP and Echelon's NEURON C Programmer's Guide for details on NEURON C syntax.

FUZZY LOGIC PRIMER

This section gives a brief introduction to the simplest concepts of 8–bit fuzzy logic. Readers who are familiar with fuzzy logic should consider skipping this section. More details can be obtained through Motorola's Fuzzy Logic Education Program, a PC tutorial available through Motorola sales offices.

Introduction

The invention of fuzzy logic is usually attributed to Lotfi Zadeh, a professor at UC Berkeley, in the mid 1960s. He developed an approach to control solutions which require neither memory intensive lookup tables nor complicated mathematical formulae. In brief, the fuzzy logic methodology, called inference, assigns predefined degrees of truth to the entire range of inputs to a system and then processes real–time inputs through a set of rules to derive a weighted system output. Three basic steps of fuzzy inference are fuzzification, rule evaluation, and defuzzification (see Figure 1), described in the following sections. Though many inference methods exist, this document will detail only one, the **min–max inference** methodology.

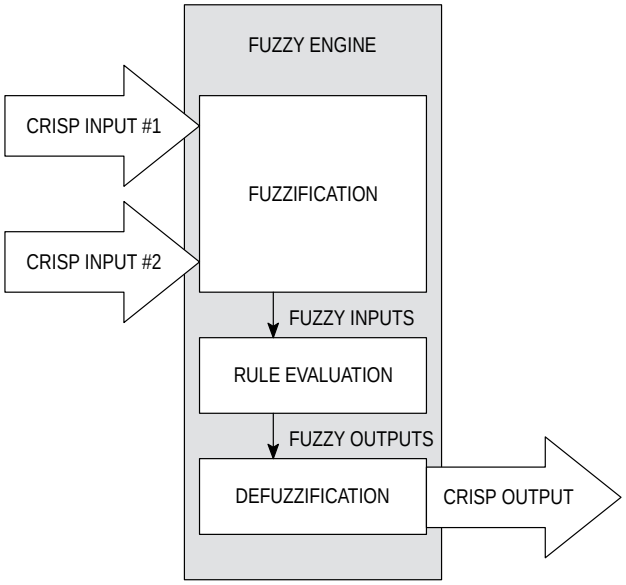

Figure 1. Block Diagram of Fuzzy System

Fuzzification

A fuzzy controller will receive crisp inputs (typically two or three) on its input or communications port and initially fuzzify them. Each system input is divided into overlapping sets of **membership functions**, typically 3 to 9 sets per input. The predefined membership functions cover the entire range of values (or universe of discourse) for an input and will define a degree of truth for every point in the universe of discourse. Figure 2 shows five trapezoidal membership functions for an input to a fuzzy controller; note that each membership function is typically labeled to *quantify* the input (i.e. very slow, fast, etc.) and that each function assigns a degree of truth (between 0 and 255) to an input. In other words, as you slide along the horizontal axis representing an input value, each point translates to **one** point on the border(s) of one or more trapezoids representing a degree of membership (pay attention only to points on the edges of the trapezoids when assigning degrees of truth to inputs). Thus fuzzy logic is unlike

NEURON is a registered trademark of Echelon Corporation.

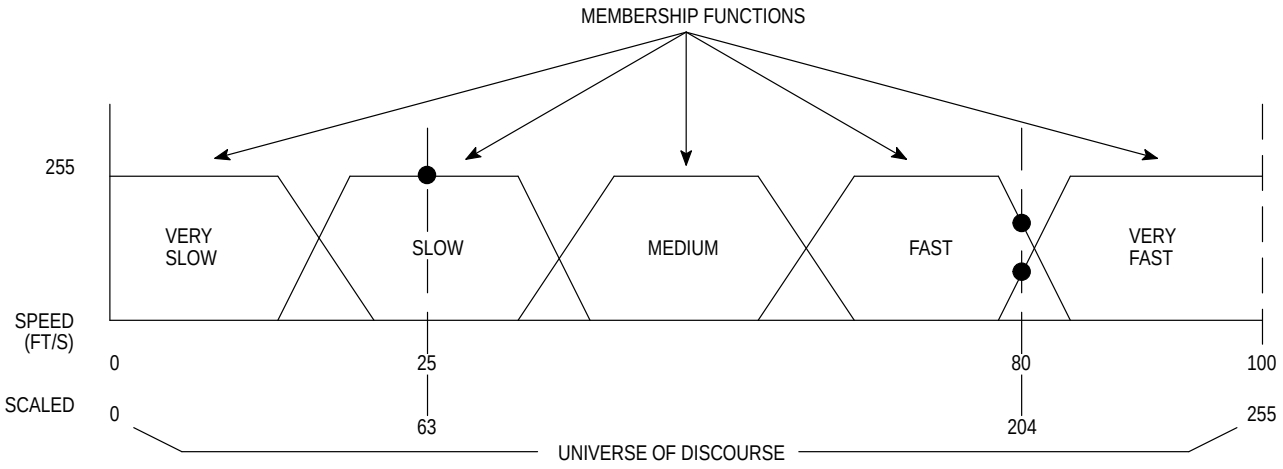

Figure 2. Input Membership Functions for Fuzzification

boolean logic in that system input values can **partially** belong to multiple sets (i.e., an input can be 30% slow and 70% medium); with boolean input values set membership is either 100% or 0%. In this sense, fuzzy logic will often help embedded controllers to respond in a smoother manner over the full range of inputs. Note that membership functions may be more complicated in shape (than the trapezoids in Figure 2) with a tradeoff of more complex arithmetic and memory requirements in the fuzzification step.

The fuzzification process uses two basic steps which are repeated for each system input. First, a crisp input must be read and scaled to a value between 0 and 255 (for an 8–bit fuzzy engine). Second, the input must be translated to a degree of membership (between 0 and 255) for each input membership function. For example, in Figure 2, if the read input indicates 25 ft/s, its value is scaled to 63 and 255 is assigned to the *slow* function — the other four functions (*very slow*, *medium*, *fast*, and *very fast*) are assigned to 0 (the input is 100% slow). In another example, the system input 80 ft/s is scaled to 204 and assigned to 179 for the *fast* function and to 76 for the *very fast* function — the other three functions are assigned to 0 (the input is 70% fast and 30% very fast). All of the assigned values to input membership functions in a system are called the fuzzified inputs of the system. In total, the number of fuzzified inputs will equal the number of inputs times the number of membership functions per input.

Rule Evaluation

Fuzzified inputs are processed through a predefined set of rules (typically 15 to 25 rules per system) using a **min–max** evaluation to form fuzzified outputs. In detail, rules are arranged in an *if–then* format — *if* two or more inputs (called antecedents) are all true *then* an output function (called a consequent) is executed to the degree of the **minimum** value antecedent (see Figure 3). Often times all the rules of a system are displayed in a matrix fashion as shown in Figure 4 where the consequents (outputs) are listed for all possible combination pairs of antecedents (inputs). For example, in Figure 4, if input #1 is 10% *medium* and input #2

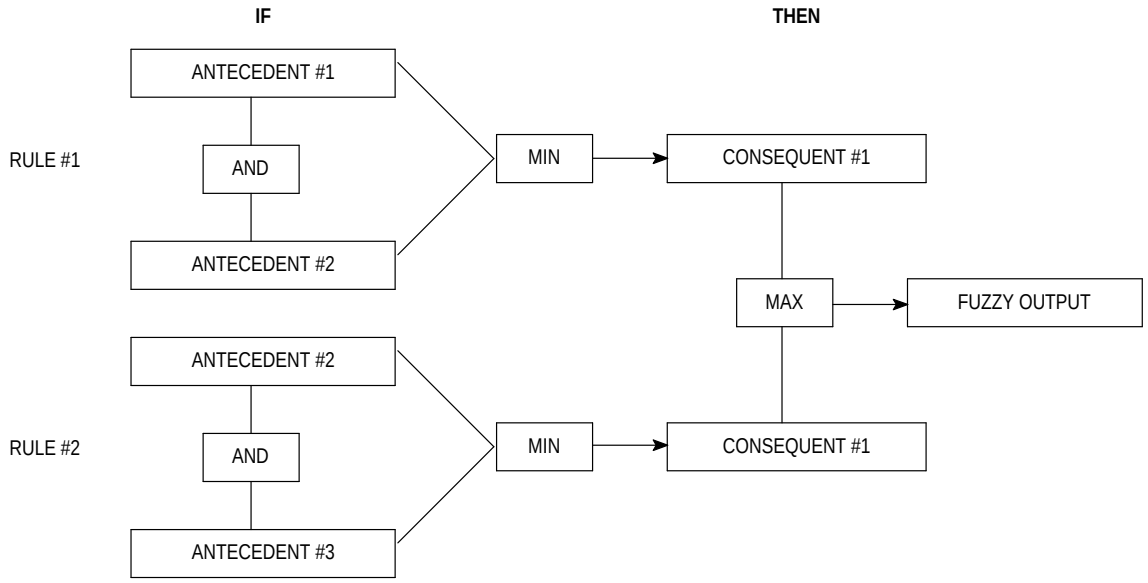

Figure 3. Rule Evaluation

INPUT #2 \ INPUT #1		SLOW	MEDIUM	FAST
INPUT #2	COLD	OFF	OFF	MEDIUM LOW
INPUT #2	WARM	MEDIUM LOW	MEDIUM	MEDIUM
INPUT #2	HOT	MEDIUM HIGH	MEDIUM HIGH	HIGH

Figure 4. Rule Matrix

is 50% *hot*, then the output *medium high* will be weighted at 10% as a result of the minimum value function. The remaining eight rules of the system would be evaluated in a similar manner to create a set of five fuzzified (weighted) outputs (described below). Additionally, for rules with the same consequent the fuzzy engine will choose the rule with the **maximum** value for the system's weighted output value. For example, in Figure 4, if *warm* and *medium* fuzzy inputs yield a 20% *medium* output, but *warm* and *fast* fuzzy inputs yield a 40% *medium* output, then the final output for *medium* will be 40% as a result of the **maximum** value function. The rule evaluation procedure just described is the primary step in min–max inference.

Fuzzified outputs are classified into membership sets similar to input membership functions. Though many types of output functions are valid, this document will only cover **singletons** in which the scaled outputs of a system (ranging from 0 to 255) are defined as 3 to 9 discrete values which are assigned weights (between 0 and 255) in the rule evaluation step described above. The example in Figure 5 illustrates five singletons representing possible output values for a single output. Note that the output singletons are often labeled to "quantify" the output (i.e., medium low, very high, etc.). The number of fuzzified outputs for a system will equal the number of outputs times the number of singletons per output. The final raw or crisp output value of the system is determined in the defuzzification step.

Defuzzification

The final task of a fuzzy engine is to defuzzify its fuzzy outputs into a single raw or crisp output for an external device (i.e., stepper motor, D/A converter, etc.). This document describes a center of gravity method. As described above, the fuzzified outputs are a set of weights for the discrete values called singletons. The final scaled output is the result of the following equation:

$$\text{scaled output} = \left(\Sigma\,(\text{fuzzy outputs} * \text{output singletons})\right) / \left(\Sigma\,\text{fuzzy outputs}\right)$$

The output is a value between 0 and 255 which might need to be scaled for non–8–bit output functions. For example, in Figure 5 if the rule evaluation process determines the system output is 30% medium low, 60% medium, and 10% medium high, then the center of gravity calculation will yield:

$$((.3*255)*\,90 + (.6*255)*128 + (.1*255)*170) / (.3*255 + .6*255 + .1*255) = 116$$

The value 116 can then be scaled for its output. Note that not all output singletons (i.e., the ones with a value of zero) will contribute to an output calculation for a set of inputs.

Conclusions

Min–max inference is quite simple to implement, yet it provides a powerful and rigorous solution for embedded controllers. Motorola's 8–bit kernels all use this type of inference because it is efficient in timing and code size. Many other types of fuzzy inference exist and may be required for complex or highly accurate solutions, but min–max inference is applicable to a majority of control applications.

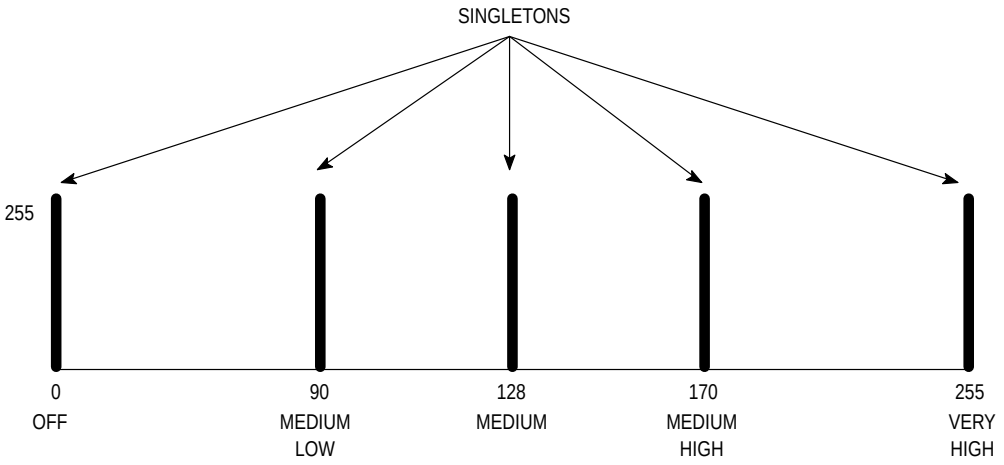

Figure 5. Output Singletons

FUZZY KERNEL FOR THE NEURON CHIP

Introduction

A fuzzy kernel, or engine, is simply a skeleton of programming code which will perform the three basic steps of fuzzy logic — fuzzification, rule evaluation, and defuzzification. The kernel user makes the programming code unique by defining and entering the input membership functions, rules, and singletons in tabular form and entering scaling equations for the crisp inputs and outputs. In addition to the kernel for the MC143150, Motorola offers, free of charge, fuzzy kernels for two of its 8–bit microcontrollers (the MC68HC11 and HC05) as well as its 16–bit HC16 microcontroller and the 32–bit 68000 family. The fuzzy kernel in this document, written in NEURON C, was designed using the HC11 kernel as a model. See Appendix A for a print out of the NEURON C fuzzy kernel.

Fuzzification

The fuzzification function produces a set of fuzzy inputs by reading a real–time crisp input, scaling it to 8 bits, and assigning a degree or grade to it for each input membership function defined by the user. First, the designer of the embedded controller must add equations to the kernel to scale crisp inputs to 8–bit values before fuzzification at which time the NEURON fuzzy engine allows up to 4 inputs (default of 4) with 8 membership functions per input. The number of inputs is set by redefining the constant called **NUM_INPUTS** (the elements per input membership function is always 8) and the input membership functions are defined by modifying the table called **input_function**. The membership functions (4 bytes each) are entered in tabular form and represent points and slopes which characterize the trapezoids (point 1, slope 1, point 2, slope 2 — see Figure 6). Note that negative slopes are entered as positive numbers since the kernel is aware that the second slope entered will be negative. Also, vertical slopes (typically on the minimum and maximum sides of the universe of discourse) are given values of 0. The minimum slope (default of 8) eliminates unnecessary slope calculations for larger input values and can be redefined by changing the constant called **MIN_SLOPE**. Unused membership functions must remain in the table and are entered as 0xff — the kernel is designed to ignore unused inputs.

Since the fuzzification process uses repetitive looping, the number of inputs and the number of membership functions per input will affect overall inference times. In other words, the basic function of the fuzzification process is to assign a degree or grade to each membership function, so the overall time of execution is directly related to the number in input membership functions used.

Rule Evaluation

The rule evaluation process produces a set of fuzzy outputs (one for each singleton) based on the min–max inference process described in the Primer section above. The NEURON fuzzy engine allows any number of rules each with any number of antecedents and consequents. The total number of antecedents is set by redefining the constant called **NUM_ANTECEDENTS**, and the total number of consequents is set with **NUM_CONSEQUENTS**. Each antecedent and consequent uses one byte of table space (in the table called **rules**) as shown in Figure 7. Also the number of outputs is limited to 2 (set by redefining **NUM_OUTPUTS**) and singletons per output is limited to 8 (controlled by redefining the constant called **SING_PER_OUTPUT**). An example of a rule table entry and its connections with membership functions and singletons is shown in Figure 8. Keep in mind that the rule evaluation step uses repetitive looping, thus as the number of rules and singletons increases so does the inference time (and the amount of memory required). The rule table is terminated by a 0xff.

Defuzzification

The defuzzification process performs a center of gravity calculation on the fuzzified outputs using the equation listed in the Primer section above. This process yields an 8–bit crisp output value which may need to be scaled for its output. Its execution time is dependent on the number of outputs and the number of singletons per output.

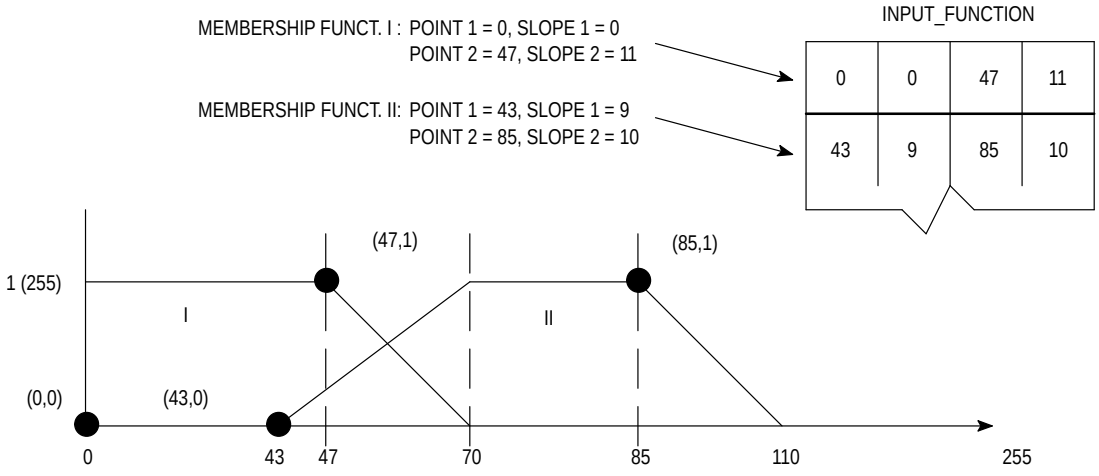

Figure 6. Table Entires for Input Membership Functions

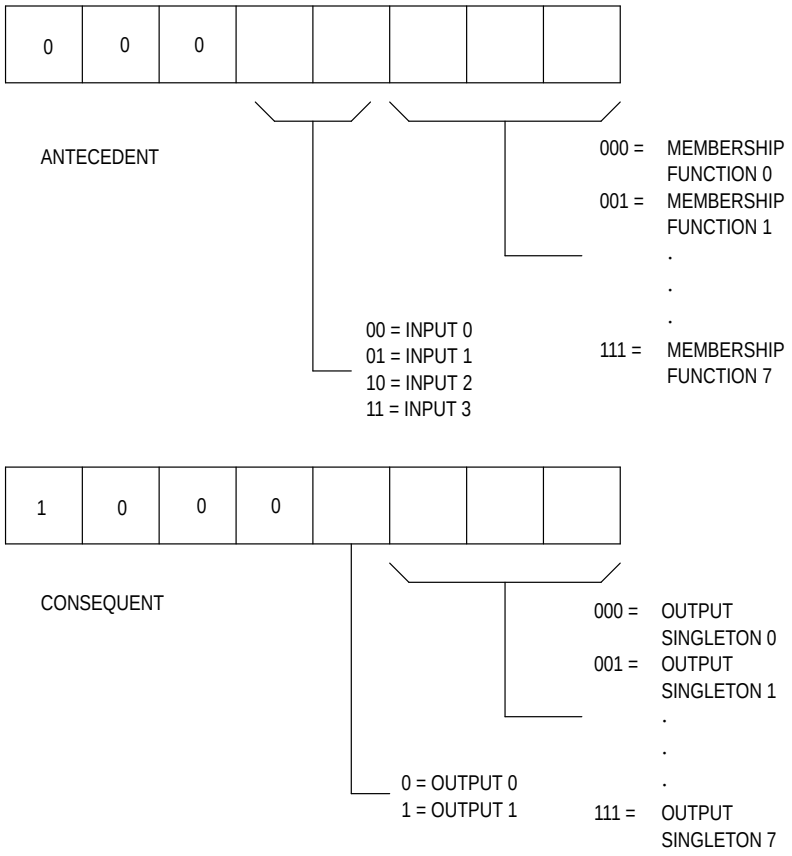

Figure 7. Table Format for Antecedents and Consequents

Results

Most fuzzy engines are analyzed for three basic parameters: performance, code size, and inference time. First, performance is a less tangible study and involves observing the *smoothness* of output performance particularly in transition areas (i.e., where the input membership functions overlap), and minimum and maximum input values. Second, the size of the kernel presented in this document is 983 bytes. Note that the size of the kernel in different applications will vary, depending upon the number of inputs, outputs, and rules. Third, inference times are measurable but are also dependent upon system parameters (number of inputs, membership functions per input, number of rules, and number of singletons). Our study used the following parameters for its benchmarking: two inputs, 5 membership functions per input, 20 rules, and 5 singletons for one output. The fuzzy inference times (not including input access or scaling) varied between 19 and 24 ms (larger values of *transition* inputs typically take longer to fuzzify); each section of code was timed in an optimization study (see Table 1) and parameters such as singletons and rules were varied in quantity when measuring overall inference time (note that the optimization study was performed with version 2.2 of the LONBUILDER; execution times were improved over version 2.1 by using *fastaccess* data types for all arrays). In conclusion the study shows that the NEURON fuzzy kernel can be used to implement a dedicated 30 Hz controller.

Table 1. Execution Times with Kernel Variations

Characteristic	Time (ms)
Fuzzification	3.6–7.8
Rule Evaluation	10–12.4
Defuzzification	3.6–4.4
Inference	19–24
Inference — with 10 rules	14–18
Inference — with 15 rules	16.5–21
Inference — with 3 singletons	17.5–22
Inference — with 7 singletons	20.5–26
Inference — with 3 membership functions	17.5–21.5
Inference — with 7 membership functions	19.5–25

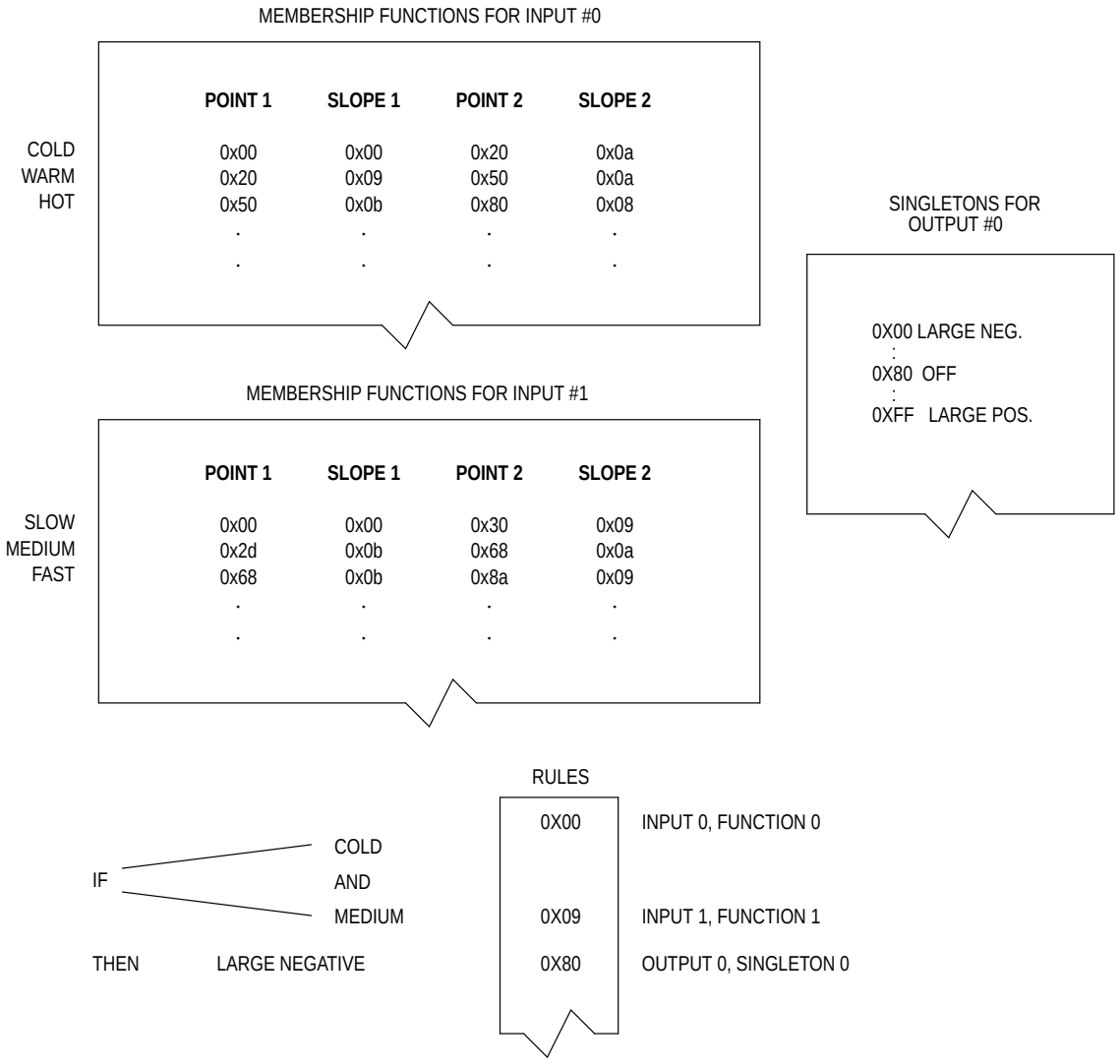

Figure 8. Association Between Rules, Membership Functions, and Singletons

FUZZY FAN APPLICATION USING THE NEURON CHIP

Introduction

The final goal of this document is to present the NEURON CHIP as a fuzzy controller in a network. The following example uses two network inputs, water temperature and water flow rate, to control the output (speed of a fan motor) in Motorola's LONWORKS Fluid Demo (see document on the Fluid Demo for more system details — Figure 9 of this document gives a system diagram). In brief, the fan node receives temperature and flow rate data from two nodes via its communication port on a 78 kbps twisted pair network, runs scaled values through its fuzzy controller, and scales crisp outputs for its PWM output control to a fan. Thus the fan speed will be controlled by network data. The software for the fan node is presented in Appendix B of this document.

Hardware

The hardware for the fan node is shown in Figure 10. A PWM signal is output from pin IO_1 of the NEURON IC to control a periodic pulse into the MOC2A40–10 triac device which will control the fan's motor speed. The schematic shows that Motorola's OEK–1 Evaluation Board was used; note that the input current amplifying circuit is not necessary with the NEURON IC since IO_1 is capable of driving 20 mA.

Software

Most of the required software was contained within the fuzzy kernel, however scaling equations had to be written and table values had to be entered to convert the kernel into a fan controller. First, the inputs were received as network variables, thus the temperature value (2–bytes ranging from 32 to 185) and flow rate (2–bytes ranging from 0 to 100) had to be scaled to 8–bit values. Second, input membership functions, rules, and output singletons were created for the fan controller. The input membership functions are shown in Figure 11; note that the temperature input has four membership functions and flow rate has five.

The rules for the fan controller are shown in Table 2. Note that all possible combinations of the two inputs were used to form 20 rules (40 antecedents and 20 consequents). Often times the rule process can be optimized to eliminate consequents, thus allowing the fuzzy engine to perform faster.

Finally, the 5 singletons for the fan speed output are shown in Figure 12. Be aware that if the values of singletons on the right side of the graph are too high, overflows can occur when using 16–bit arithmetic in the center of gravity calculation (this can be rectified by breaking the calculation down into several equations).

The third step in writing the software was scaling the crisp 8–bit output to a 16–bit PWM output. Once again, the fuzzy fan software is shown in Appendix B.

Results

The fan node was tested for performance characteristics and fuzzy execution time. The key areas of observation for performance were minimum and maximum input values and the transition areas of input membership functions. The limitations of 16–bit arithmetic were discovered in some of the transition areas as the center of gravity calculation overflowed with higher singleton values on the output. This problem can be avoided by adjusting the singletons of an output or breaking the calculations down into several blocks. After adjusting the singletons, the advantages of fuzzy logic were observed in the smooth transitions of the fan speed as the inputs varied. Finally the execution time of the fuzzy loop (including scaling) varied between 22.5 and 29 ms over the universe of discourse. Keep in mind that this NEURON CHIP was dedicated to fan control and that other functions could potentially slow down the operation of the fuzzy engine.

Conclusions

The NEURON CHIP can add value operating as a fuzzy engine for embedded controls on a distributed network. The only limiting factor of the NEURON controller is its slow inference time as a result of programming the kernel in NEURON C. However, many applications will operate to specification with a 30 Hz controller and if demand is high enough, Echelon may consider writing the fuzzy routines in object code. On the other hand, the added benefit of using the NEURON CHIP is its communications capabilities. Inputs received on the network take virtually no time for the application code to read, as they are handled by the device's network and MAC processors which place the data in RAM. Also, use of a network implies that inputs can easily be received from remote locations. Overall, the use of fuzzy controllers in a distributed network environment can result in considerable improvements in system performance.

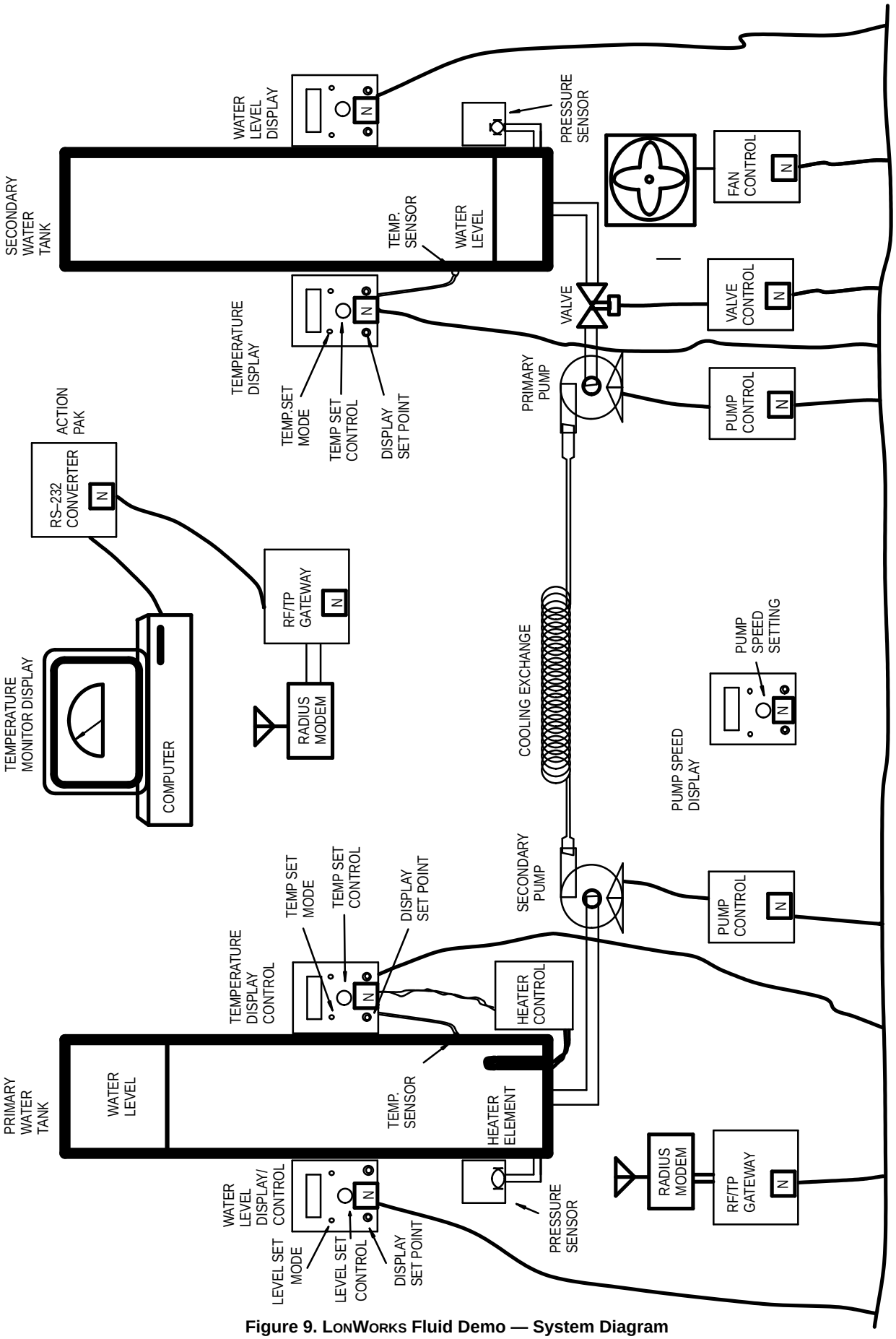

Figure 9. LonWorks Fluid Demo — System Diagram

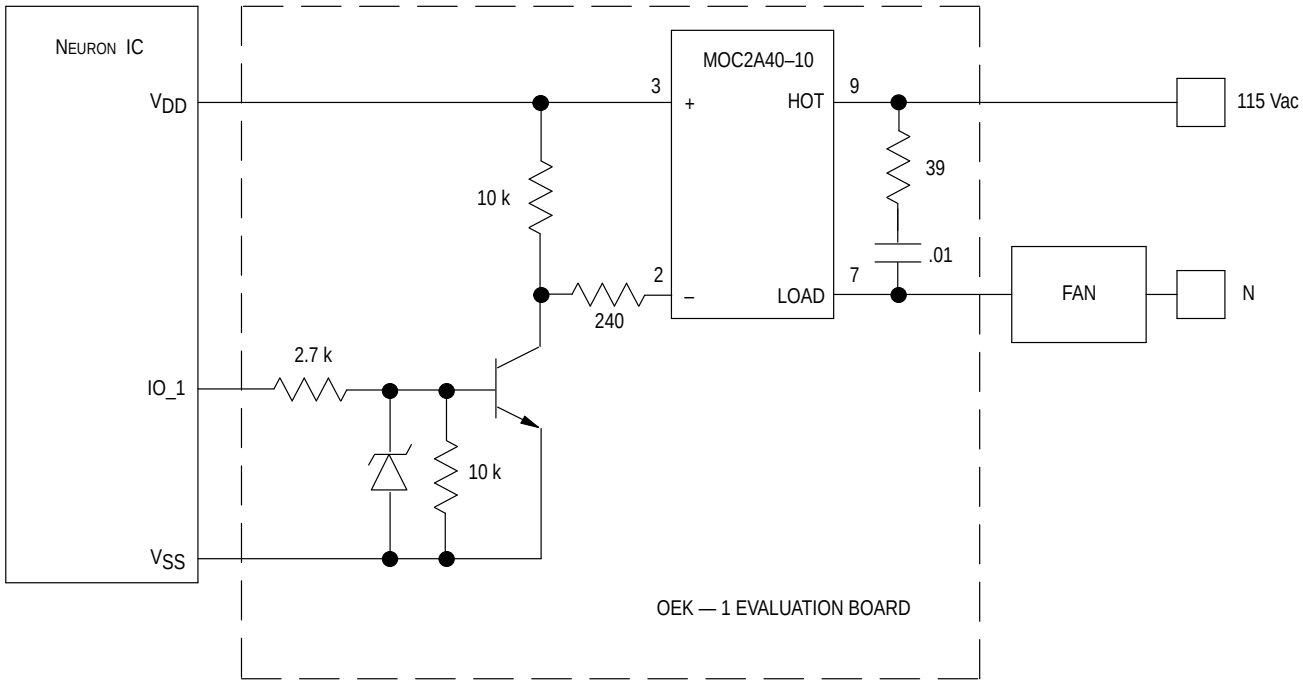

Figure 10. Schematic of Fan Mode

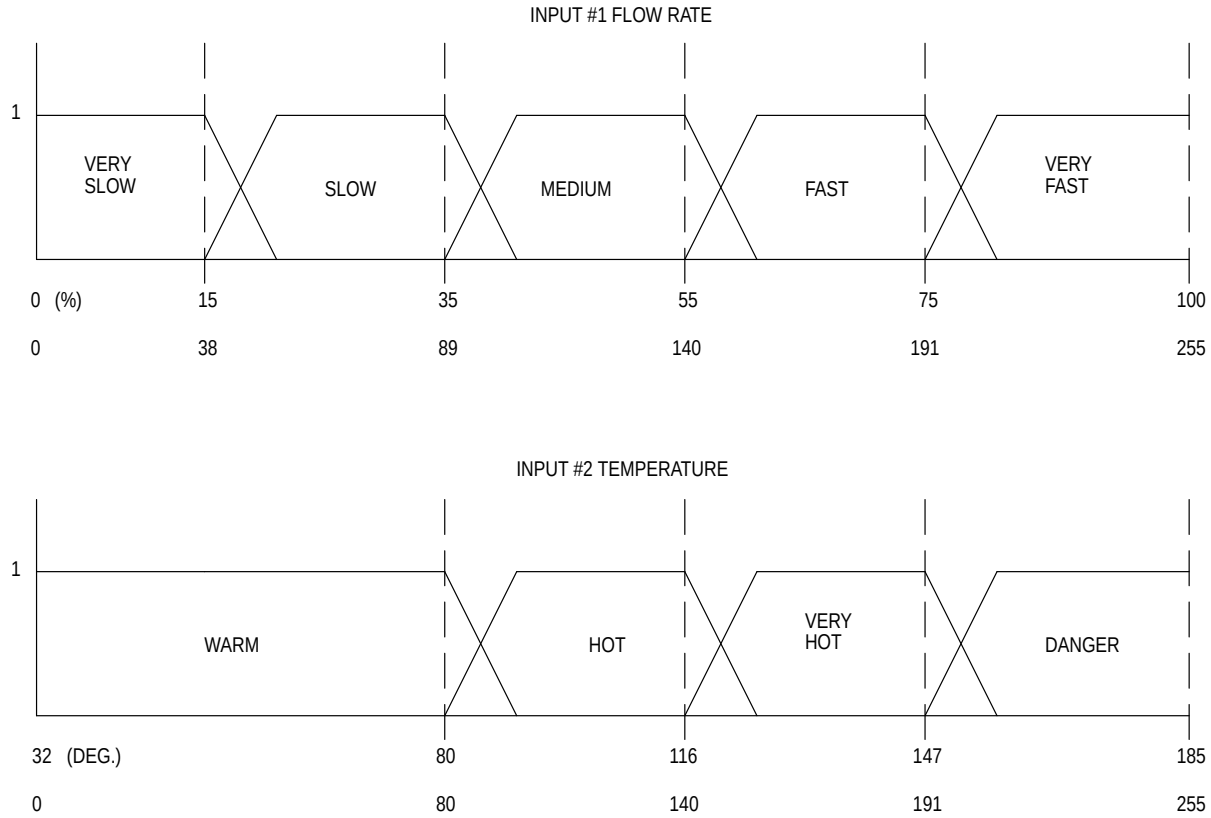

Figure 11. Input Membership Functions for Fuzzy Fan

Table 2. Fuzzy Fan Rules

TEMPERATURE \ FLOW RATE	VERY SLOW	SLOW	MEDIUM	FAST	VERY FAST
WARM	OFF	OFF	OFF	MEDIUM LOW	MEDIUM LOW
HOT	MEDIUM LOW	MEDIUM LOW	MEDIUM	MEDIUM	MEDIUM HIGH
VERY HOT	MEDIUM	MEDIUM HIGH	MEDIUM HIGH	HIGH	HIGH
DANGER	MEDIUM HIGH	MEDIUM HIGH	HIGH	HIGH	HIGH

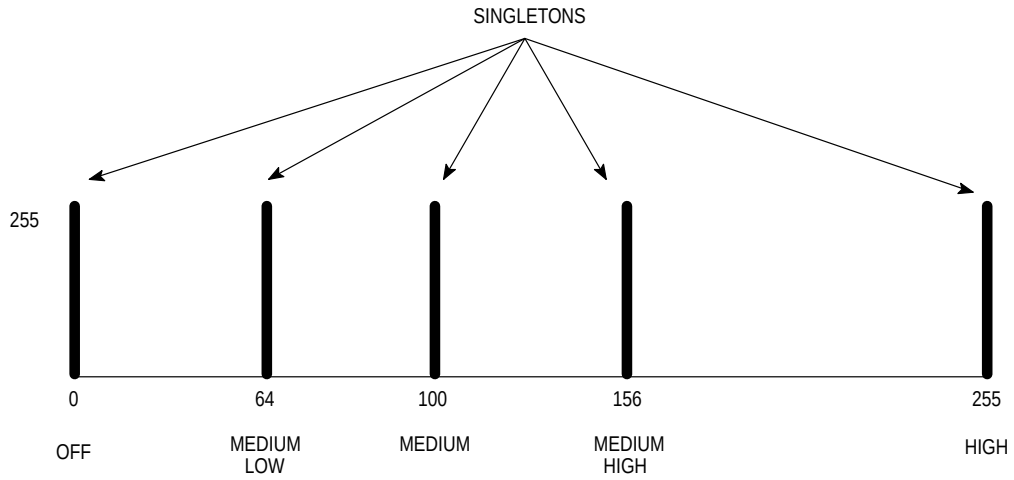

Figure 12. Singletons for Fan Node Output

APPENDIX A
NEURON C FUZZY KERNEL

```
//*****************************************************************************************
    This program is a Neuron IC fuzzy kernel written in Neuron C
    with the following features:

    1)  Up to 4 4-byte inputs formatted {pt1, slope1, pt2, slope2}.
    2)  Up to 8 membership elements per input function.
    3)  Up to 2 outputs.
    4)  Up to 8 1-byte singletons per output function formatted {pt}.
    5)  1 byte per antecedent ('if') formatted 000X XAAA
        where XX = input# and AAA = input function member#.
    6)  1 byte per consequent ('then') formatted 1000 XAAA
        where X = output# and AAA = output function singleton#.
    7)  Min-max inference.
    8)  Defuzzification using COG calculation.
*****************************************************************************************/

/********************************** Compiler directives **********************************/

#define NUM_OUTPUTS 2
#define SING_PER_OUTPUT 8
#define NUM_INPUTS 4
#define ELEMENTS_PER_INPUT 8
#define BYTES_PER_ELEMENT 4
#define NUM_ANTECEDENTS 0
#define NUM_CONSEQUENTS 0
#define MIN_SLOPE 8

/************************************** Globals ***************************************/

unsigned int input_function [NUM_INPUTS] [ELEMENTS_PER_INPUT]
    [BYTES_PER_ELEMENT] = {
    {                                       //Input #0
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff}
    },
    {                                       //Input #1
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff}
    },
    {                                       //Input #2
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff}
    },
```

```
    {                                       //Input #3
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff},
        {0xff, 0xff, 0xff, 0xff}
    }

};

unsigned int singletons [NUM_OUTPUTS] [SING_PER_OUTPUT] = {
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},    //Output #1
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}     //Output #2

};

unsigned int rules [NUM_ANTECEDENTS + NUM_CONSEQUENTS + 1] = {
    0xff

};

unsigned int *input_pt;
unsigned int *rule_pt;
signed long crisp_inputs [NUM_INPUTS];
unsigned int crisp_outputs [NUM_OUTPUTS];
unsigned int index;
unsigned int row_index;
unsigned int col_index;
signed long *fuzzy_pt;
signed long fuzzified_inputs [ELEMENTS_PER_INPUT * NUM_INPUTS];
unsigned int *fuzzy_pt2;
unsigned int fuzzified_outputs [NUM_OUTPUTS][SING_PER_OUTPUT];
unsigned int minimum;
unsigned long sum;
unsigned long sum_of_products;
unsigned int point1;
unsigned int point2;
unsigned int slope1;
unsigned int slope2;
unsigned long pwm_value;
unsigned int max_displacement;
unsigned int local [2];

when (reset) {
    max_displacement = 255 / MIN_SLOPE;
}       //end when
```

```
/*********************************** Fuzzy engine ***************************************/
when (1) {

/************* Scale inputs here (give values to crisp_inputs[] and local[]) *************/

/************************************ Fuzzification ************************************/
    input_pt = &input_function[0][0][0];
    fuzzy_pt = &fuzzified_inputs[0];

    /****************************** Look at each updated input ***************************/
    for (index = 0; index < NUM_INPUTS; index++)   {

        /************************ Assign a grade to each input function member ************/
        for (row_index = 0; row_index < ELEMENTS_PER_INPUT; row_index++)  {
            point1 = *input_pt++;

            /**check if crsip input is below defined input range ***/
            if (local[index] <= point1)  {
                if (point1) *fuzzy_pt++ = 0;                    //out of range
                else *fuzzy_pt++ = 0xff;
                input_pt += 3;                                  //point to next input
                goto jump;
            }       //end if

            slope1 = *input_pt++;
            point2 = *input_pt++;

            /* check if crisp input is within input range and to the left of pt2 */
            if (local[index] <= point 2)  {
                if (!slope1) *fuzzy_pt++ = 0xff;                //vertical slope
                else {
                   *fuzzy_pt = ((long)slope1) *
                      (crisp_inputs[index] – point1);
                   if (*fuzzy_pt > 0xff) *fuzzy_pt = 0xff;      //max value
                   *fuzzy_pt++;                                 //next grade
                }          //end else
                input_pt++;                                     //point to next input
                goto jump;
            }               //end if

            slope2 = *input_pt++;

            /* check if crisp input is to the right of pt2 within reasonable range */
            if (slope2 && (crisp_inputs[index] < (point2 + max_displacement)))  {
                *fuzzy_pt = 255 – ((long)slope2*
                (crisp_inputs[index] – point2));
                if (*fuzzy_pt < 0) *fuzzy_pt = 0;               //out of range
                *fuzzy_pt++;
            }             //end if

            else *fuzzy_pt++ = 0;                               //vertical slope
            jump: if (1);
        }          //end for
    }       //end for
```

```
/*********************************** Rule evaluation ************************************/
    fuzzy_pt2 = &fuzzified_outputs[0][0];
    for (index = 0; index < NUM_OUTPUTS * SING_PER_OUTPUT; index++)
        *fuzzy_pt2++ = 0;                   //clear output array
    rule_pt = &rules[0];
    while (1)  {
        minimum = 0xff;
        while (*rule_pt < 0x80)  {          //antecedent evaluation (min function)
            if (!minimum) rule_pt++;        //check for 0 minimum
            else {
               //point to fuzzified input location
               fuzzy_pt = &fuzzified_inputs [0] + *rule_pt++;
               //check for new minimum
               if (*fuzzy_pt < minimum) minimum = *fuzzy_pt;
            }       //end else
        }       //end while
        while (*rule_pt & 0x80) {           //consequent evaluation (max function)
            if (*rule_pt == 0xff) goto done;                //end of rules
            if (!minimum) rule_pt++;        //check for 0 maximum
            else {
               //point to fuzzified output location
               fuzzy_pt2 = &fuzzified_outputs [0][0] + (*rule_pt++ – 0x80);
               //check for new maximum
               if (minimum > *fuzzy_pt2) *fuzzy_pt2 = minimum;
            }       //end else
        }       //end while
    }       //end while
    done: if (1);

/*********************************** Defuzzification ************************************/

    /********************************* COG for all outputs ******************************/
    for (row_index = 0; row_index < NUM_OUTPUTS; row_index++)  {

        /***************************** Sum of products for each output *****************/
        sum = 0;
        sum_of_products = 0;
        for (col_index = 0; col_index < SING_PER_OUTPUT; col_index++)  {
            sum += fuzzified_outputs[row_index][col_index];
            sum_of_products += (unsigned long) singletons[row_index][col_index]
               * (unsigned long) fuzzified_outputs[row_index][col_index];
        }          //end for
        crisp_outputs[row_index] = sum_of_products / sum;
    }       //end for

/************************** Scale output(s) and call output function(s) ******************/
}       //end when
```

APPENDIX B
NEURON C FAN CONTROL NODE

```
/*****************************************************************************************

Function: fan.nc
Definition:
    This program is a NEURON IC fan node for Motorola's water demo using a fuzzy kernel with
    the following fuzzy features:

    1)  2 4-byte inputs - flow rate and temperature formatted {pt1, slope1, pt2, slope2}.
    2)  5 membership elements for flow rate, 4 for temperature.
    3)  1 output - fan speed.
    4)  5 1-byte singletons for fan speed.
    5)  1 byte per antecedent ('if') formatted 000X XAAAA
        where XX = input# and AAA = input function member#.
    6)  1 byte per consequent ('then') formatted 1000 XAAA
        where X = output# and AAA = output function singleton#.
    7)  Min-max inference.
    8)  Defuzzifiation using COG calculation.

I/O inputs: none
I/O output: PWM signal to ac fan motor via triac

net inputs: temperature and water flow rate
net outputs: none

application image (ROM): 1065 bytes

required header files: none

rev: 1.0    6/3/93      first revision
     1.1    6/10/93     optimization - used unsigned int comparison
                        in fuzzification instead of long; also
                        fastaccess arrays (rev. 2.2 of LONBUILDER).
*****************************************************************************************/


/********************************** Compiler directives *********************************/

#pragma enable_io_pullups
#define NUM_OUTPUTS 1
#define SING_PER_OUTPUT 5
#define NUM_INPUTS 2
#define ELEMENTS_PER_INPUT 8
#define BYTES_PER_ELEMENT 4
#define NUM_ANTECEDENTS 40
#define NUM_CONSEQUENTS 20
#define MIN_SLOPE 8


/************************************* I/O objects ************************************/

IO_1 output pulsewidth long clock(5) IO_pwm;
IO_4 output bit test;


/********************************** Network Variables ***********************************/

network input unsigned int NV_temp;
network input unsigned int NV_pump_spd;
```

```
/************************************** Globals ***************************************/

fastaccess unsigned int input_function [NUM_INPUTS] [ELEMENTS_PER_INPUT]
   [BYTES_PER_ELEMENT] = {
   {
     {0x00, 0x00, 0x26, 0x0a},          //very slow
     {0x26, 0x0a, 0x59, 0x0a},          //slow
     {0x59, 0x0a, 0x8c, 0x0a},          //medium
     {0x8c, 0x0a, 0xbf, 0x0a},          //fast
     {0xbf, 0x0a, 0xff, 0x00},          //very fast
     {0xff, 0xff, 0xff, 0xff},          //not used
     {0xff, 0xff, 0xff, 0xff},
     {0xff, 0xff, 0xff, 0xff}
   } ,
   {
     {0x00, 0x00, 0x50, 0x0a},          //warm
     {0x50, 0x0a, 0x8c, 0x0a},          //hot
     {0x8c, 0x0a, 0xbf, 0x0a},          //very hot
     {0xbf, 0x0a, 0xff, 0x00},          //danger
     {0xff, 0xff, 0xff, 0xff},          //not used
     {0xff, 0xff, 0xff, 0xff},
     {0xff, 0xff, 0xff, 0xff},
     {0xff, 0xff, 0xff, 0xff}
   }
};

fastaccess unsigned int singletons [NUM_OUTPUTS] [SING_PER_OUTPUT] = {
     {0x00, 0x40, 0x64, 0x9c, 0xff}
};

fastaccess unsigned int rules [NUM_ANTECEDENTS + NUM_CONSEQUENTS + 1] = {
     0x08, 0x00, 0x80,          //if warm and very slow, then off
     0x08, 0x01, 0x80,          //if warm and slow, then off
     0x08, 0x02, 0x80,          //if warm and medium, then off
     0x08, 0x03, 0x81,          //if warm and fast, then medium low
     0x08, 0x04, 0x81,          //if warm and very fast, then medium low
     0x09, 0x00, 0x81,          //if hot and very slow, then medium low
     0x09, 0x01, 0x81,          //if hot and slow, then medium low
     0x09, 0x02, 0x82,          //if hot and medium, then medium
     0x09, 0x03, 0x82,          //if hot and fast, then medium
     0x09, 0x04, 0x83,          //if hot and very fast, then medium high
     0x0a, 0x00, 0x82,          //if very hot and very slow, then medium
     0x0a, 0x01, 0x83,          //if very hot and slow, then medium high
     0x0a, 0x02, 0x83,          //if very hot and medium, then medium high
     0x0a, 0x03, 0x84,          //if very hot and fast, then high
     0x0a, 0x04, 0x84,          //if very hot and very fast, then high
     0x0b, 0x00, 0x83,          //if danger and very slow, then medium high
     0x0b, 0x01, 0x83,          //if danger and slow, then medium high
     0x0b, 0x02, 0x84,          //if danger and medium, then high
     0x0b, 0x03, 0x84,          //if danger and fast, then high
     0x0b, 0x04, 0x84,          //if danger and very fast, then high
     0xff                       //end of rules
};
```

```
unsigned int *input_pt;
unsigned int *rule_pt;
fastaccess signed long crisp_inputs [NUM_INPUTS];
fastaccess unsigned int crisp_outputs [NUM_OUTPUTS];
unsigned int index;
unsigned int row_index;
unsigned int col_index;
signed long *fuzzy_pt;
fastaccess signed long fuzzified_inputs [ELEMENTS_PER_INPUT * NUM_INPUTS];
unsigned int *fuzzy_pt2;
fastaccess unsigned int fuzzified_outputs [NUM_OUTPUTS] [SING_PER_OUTPUT];
unsigned int minimum;
unsigned long sum;
unsigned long sum_of_products;
unsigned int point1;
unsigned int point2;
unsigned int slope1;
unsigned int slope2;
unsigned long pwm_value;
unsigned int max_displacement;
fastaccess unsigned int local[2];


when (reset) {
    max_displacement = 255 / MIN_SLOPE;
}       //end when


/************************************ Fuzzy engine ************************************/

when (1) {


/************************************ Scale inputs ************************************/

    io_out (test,0);

    crisp_inputs[0] = ((signed long)NV_pump_spd) * 255 / 100;
    local[0] = crisp_inputs[0];
    crisp_inputs[1] = ((signed long)NV_temp – 32) * 5 / 3;
    local[1] = crisp_inputs[1];


/************************************ Fuzzification ************************************/

    input_pt = &input_function[0][0][0];
    fuzzy_pt = &fuzzified_inputs[0];


    /***************************** Look at each updated input *****************************/

    for (index = 0; index < NUM_INPUTS; index++) {
```

```
    /********************* Assign a grade to each input function member ******************/

    for (row_index = 0; row_index < ELEMENTS_PER_INPUT; row_index++) {
         point1 = *input_pt++;

         //check if crisp input is below defined input range
         if (local[index] <= point1) {
              if (point1) *fuzzy_pt++ = 0;        //out of range
              else *fuzzy_pt++ = 0xff;
              input_pt += 3;                      //point to next input
              goto jump;
         }         //end if

         slope1 = *input_pt++;
         point2 = *input_pt++;

         //check if crisp input is within input range and
         //to the left of pt2
         if (local[index] <= point2) {
              if (!slope1) *fuzzy_pt++ = 0xff;                 //vertical slope
              else {
                   *fuzzy_pt = ((long)slope1) *
                        (crisp_inputs[index] – point1) ;
                   if (*fuzzy_pt > 0xff) *fuzzy_pt = 0xff;     //max value
                   *fuzzy_pt++;                                //next grade
              }          //end else
              input_pt++;                                      //point to next input
              goto jump;
         }         //end if

         slope2 = *input_pt++;

         //check if crisp input is to the right of pt2 and
         //within reasonable range
         if (slope2 && (crisp_inputs[index] < (point 2 + max_displacement)))   {
              *fuzzy_pt = 255 – ((long) slope2 *
                   (crisp_inputs[index] – point2));
              if (*fuzzy_pt < 0) *fuzzy_pt = 0;                //out of range
              *fuzzy_pt++;
         }         //end if

         else *fuzzy_pt++ = 0                                   //vertical slope
         jump: if (1);
    }          //end for
}         //end for
```

```
/************************************ Rule evaluation ********************************/

fuzzy_pt2 = &fuzzified_outputs[0][0];
for (index = 0; index < NUM_OUTPUTS * SING_PER_OUTPUT; index++)
    *fuzzy_pt2++ = 0;                  //clear output array
rule_pt = &rules[0];
while (1)  {
    minimum = 0xff;
    while (*rule_pt < 0x80)  {         //antecedent evaluation (min function)
        if (!minimum) rule_pt++;       //check for 0 minimum
        else  {
            //point to fuzzified input location
            fuzzy_pt = &fuzzified_inputs[0] + *rule_pt++;
            //check for new minimum
            if (*fuzzy_pt < minimum) minimum = *fuzzy_pt;
        }          //end else
    }        //end while
    while (*rule_pt & 0x80)  {         //consequent evaluation (max function)
        if (*rule_pt == 0xff) goto done;                //end of rules
        if (!minimum) rule_pt++;                        //check for 0 maximum
        else {
            //point to fuzzified output location
            fuzzy_pt2 = &fuzzified_outputs[0][0] + (*rule_pt++ – 0x80);
            //check for new maximum
            if (minimum > *fuzzy_pt2) *fuzzy_pt2 = minimum;
        }        //end else
    }        //end while
}        //end while
done: if (1);


/************************************ Defuzzification ********************************/


    /****************************** COG for all outputs ********************************/

    for (row_index = 0; row_index < NUM_OUTPUTS; row_index++)  {


        /************************** Sum of products for each output *********************/

        sum = 0;
        sum_of_products = 0;
        for (col_index = 0; col_index < SING_PER_OUTPUT; col_index++)  {
            sum += fuzzified_outputs[row_index][col_index];
            sum_of_products += (unsigned long) singletons[row_index][col_index]
                *(unsigned long)fuzzified_outputs[row_index][col_index];
        }         //end for
        crisp_outputs[row_index] = sum_of_products / sum;
    }        //end for


    pwm_value = (unsigned long)crisp_outputs[0] * 257;
    io_out (IO_pwm, pwm_value);


    io_out (test,1);
}   //end when
```