AL/AO-TP-1994-0015

CHARACTER RECOGNITION USING NOVEL OPTOELECTRONIC NEURAL NETWORK

William McGregor Robinson

The University of Texas at San Antonio (UTSA)
Department of Electrical Engineering
6900 N. Loop 1604 W.
San Antonio, TX 78249

AEROSPACE MEDICINE DIRECTORATE
CLINICAL SCIENCES DIVISION
2507 Kennedy Circle
Brooks Air Force Base, TX 78235-5117

July 1994

Final Technical Paper for Period April 1993

AIR FORCE MATERIEL COMMAND
BROOKS AIR FORCE BASE, TEXAS

NOTICES

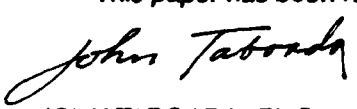
This technical paper is published as received and has not been edited by the technical editing staff of the Armstrong Laboratory.

The interpretations, conclusions, recommendations, and opinions are those of the author and are not necessarily endorsed by the U.S. Air Force or the Department of Defense.

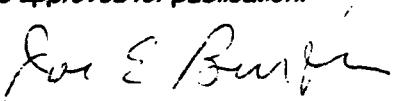
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Office of Public Affairs has reviewed this paper, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This paper has been reviewed and is approved for publication.



JOHN TABOADA, Ph.D.
Project Scientist



JOE EDWARD BURTON, Colonel, USAF, MC, CFS
Chief, Clinical Sciences Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1993	Final Paper, April 1993

4. TITLE AND SUBTITLE

Character Recognition Using Novel Optoelectronic Neural Network

5. FUNDING NUMBERS

PE - 62202F
PR - 7755
TA - 24
WU - 23

6. AUTHOR(S)

William McGregor Robinson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Texas at San Antonio (UTSA)
Dept. of Electrical Engineering

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Armstrong Laboratory (AFMC)
Aerospace Medicine Directorate
Clinical Sciences Division
2507 Kennedy Circle
Brooks Air Force Base, TX 78235-5117

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AL/AO-TP-1994-0015

11. SUPPLEMENTARY NOTES

Armstrong Laboratory Project Scientist: Dr. John Taboada, (210) 536-2857.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A novel optoelectronic neural network has been designed and constructed to recognize a set of characters from the alphabet. The network consists of a 15X1 binary input vector, two optoelectronic vector matrix multiplication layers, and a 15X1 binary output layer. The network utilizes a pair of custom fabricated Spatial Light Modulators (SLMs) with 120 levels of gray scale per pixel. The SLMs realize the matrix weights. Previous networks of this type were hampered by limited levels of gray scale and the need to use two separate weight masks (matrices) per layer. The weight masks are operated in unipolar mode. This allows both positive and negative weights to be realized from the same mask. A hard limiting function is used for the network's nonlinearity. A modification of Widrow's lesser known MR2 training algorithm is used to train the network. Furthermore, the network introduces a novel lens-free crossbar matrix-vector multiplier.

14. SUBJECT TERMS

Light modulators
Optical computing
Quadriprism
Range maps

15. NUMBER OF PAGES

190

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Dedication

I dedicate this thesis to my Lord Jesus Christ for giving all the wisdom and strength needed to complete this project. To my loving wife, Rebecca, whose needed support, encouragement, and motivation allowed me to overcome the many barriers I've encountered in my research. To my proud parents, Laurence and Elvira Robinson, for encouraging me to further my education, but never demanding or asking anything in return. To my thesis advisor, Dr. Harold Longbotham, for believing in me and supporting my research with patience, time, and funds. To my long friend, mentor, and teacher, Dr. John Taboada, for his unselfish advice and guidance through this difficult project. To Dr. Marvin Chatkoff for valued advice on the thesis document. And to Dr. John Schmalzel for insight on hardware and software issues.

CHARACTER RECOGNITION USING NOVEL OPTOELECTRONIC NEURAL NETWORK

by

William McGregor Robinson

B.S.E.E.

THESIS

Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
APRIL 1993

Acknowledgments

I wish to extend my gratitude to the following individuals who were instrumental in the development, support, and realization of the hardware and software. Daniel Shelton for programming advice and help. Dr. Ralph Hill of Southwest Research Institute, for the manufacturing of some of the printed circuit boards. To Technical Sergeant Floyd Messick for printed circuit board manufacturing and hardware assembly. And to Dr. Parimal Patel for the use of digital test equipment.

Abstract

A novel optoelectronic neural network has been designed and constructed to recognize a set of characters from the alphabet. The network consists of a 15X1 binary input vector, two optoelectronic vector matrix multiplication layers, and a 15x1 binary output layer. The network utilizes a pair of custom fabricated Spatial Light Modulators (SLM's) with 120 levels of gray scale per pixel. The SLM's realize the matrix weights. Previous networks of this type were hampered by limited levels of gray scale and the need to use two separate weight masks (matrices) per layer. The weight masks are operated in unipolar mode. This allows both positive and negative weights to be realized from the same mask. A hard limiting function is used for the network's nonlinearity . A modification of Widrow's lesser known MR2 training algorithm is used to train the network. Furthermore, the network introduces a novel lens-free crossbar matrix-vector multiplier.

TABLE OF CONTENTS

CHAPTER 1

Introduction

Artificial Neural Networks (ANN's) are computing networks that are adaptable or trainable, and naturally parallel. ANN's are also referred to as parallel distributed processors, adaptive systems, connectionist models, self-organizing systems, and neuromorphic systems [1,2]. ANN's are biologically inspired and are based on neurological models[1,2,3,5,8]. Neural networks offer significant potential benefits in the areas of pattern and speech recognition, information processing, nonlinear controls, and expert systems[2,3].

Currently, there is a demand for high speed, low cost, and small size ANN's to serve a variety of applications. Silicon large scale neuro chips have been designed and demonstrated [51,52,53,54,55,56]. However these are limited in interconnect density. Ozaktkas and Goodman [57] point out that low interconnect density arises from large area requirements of conductor guided interconnections.

To overcome the problems associated with electronic ANN's, various optical neural network configurations have been investigated [34 through 50]. Optics offer many advantages such as massive parallelism, free space interconnects, and immunity to EMI noise. However, most of these networks are impractical and only demonstrable in a laboratory environment. By combining optics and electronics, optoelectronic neuro-chips offer an alternative to previous networks [59]. A hybrid optoelectronic neural network for recognizing simple characters has been designed and developed.

This thesis provides a tutorial in the areas of neural networks and related optics and reports on the development of an optoelectronic artificial neural network. Chapter 2 covers some important foundations, and provides a general framework for neural networks. Chapter 3 discusses the adaptive linear combiner or ADALINE neuron and linear separability which provides a justification for multilayer networks. The MADALINE (many ADALINE) multi layer network is also discussed. Chapter 4 discusses optics for neural networks with special emphasis on Polarization and Spatial Light Modulators (SLM's). Chapter 5 describes development and implementation of an optoelectronic neural network and discusses input dependent thresholding. Driver electronics for the electro optical components are discussed. Software pseudocode is also presented. Chapter 6 describes the MADALINE RULE 2 (MR 2) training algorithm and the justification for its implementation. A training set is presented and results are shown. Chapter 7 concludes the thesis and presents recommendations for future work.

CHAPTER 2

Foundation

The human brain outperforms the fastest digital computers in areas of association, categorization, generalization, and feature extraction. It consists of about 10^{10}-10^{11} neurons, each making about 10^{3}-10^{4} synaptic interconnections with neighboring neurons for a total of 10^{13}-10^{15} interconnections. The brain operates at approximately 100 Hz. This means that the brain functions at about 10^{16} interconnections per second [2]. A cross sectional view of the cerebral cortex (main computing apparatus of the brain) reveals three to six layers of neurons similarly arranged. This corresponds roughly to about 500 neural network modules [3].

2.1 Biology

ANN's are biologically inspired and are made up of elements analogous to that of the biological neuron [3,5]. A biological neuron consists of *axons*, *dendrites* and *synapses* (Figure 2.1)[2]. The *soma*, or nerve cell, is the large central round body of the neuron. The *axon*, a thin extended fiber attached to the soma, produces and conducts electrical pulses. *Dendrites* extend from the cell body to other neurons where they passively receive electrical signals at a connection point called a *synapse*. At the synapse's receiving end, inputs are conducted to the cell body where they are summed. Inputs can either excite or inhibit the cell. When the cumulative excitation in the cell body exceeds a threshold, the cell fires, propagating a signal down the axon to other cells[5].

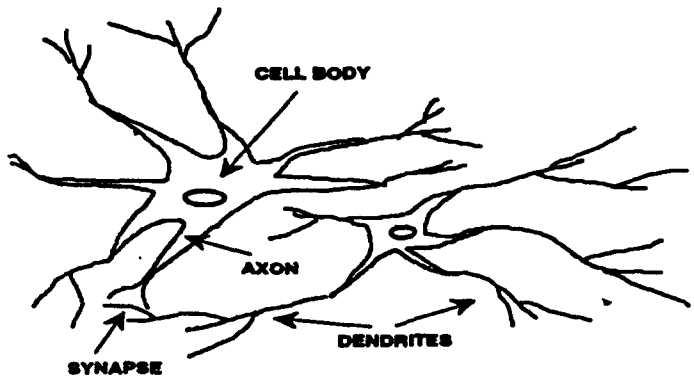


Figure 2.1. Biological Neuron

2.2 Artificial Neuron

An artificial neuron (Figure 2.2) is a simple processing element which emulates its biological counterpart. Processing elements are organized in a way that may or may not be related to the anatomy of the brain, but surprisingly exhibit a number of the brain's characteristics [3,5]. Processing elements sum weighted inputs. The weights can be either inhibitory (negative) or excitatory (positive) with each weight corresponding to the "strength" of a single biological synaptic connection. To better understand the artificial neuron model one must first be introduced to a general framework for ANN's.

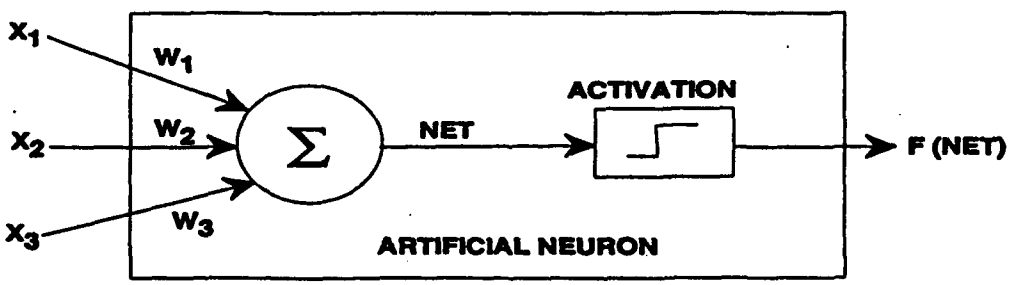


FIGURE 2.2. ARTIFICIAL NEURON

2.3 Framework

The general framework of ANN models is described by Rumelhart, Hinton, and Williams [11]. There are eight major aspects common to most ANN models. These are:

1. A *set of processing neurons*
2. A *state of activation*
3. An *output function* for each neuron
4. A *pattern of connectivity* for each neuron
5. A *propagation rule* for propagating patterns of activities through the network of connectivities
6. An *activation rule* for combining the inputs impinging on an neuron with the current state of that neuron to produce a new level of activation for the neuron
7. A *learning rule* whereby patterns of connectivity are modified by experience
8. An *environment* within which the system must operate

2.3.1 Set of processing neurons

ANN models must specify a set of processing neurons and what they represent. These neurons may represent conceptual objects such as words, features, or letters. They may also represent abstract elements which define meaningful patterns. There are three kinds of neurons, *input*, *hidden*, and *output* (Figure 2.3). Input neurons receive inputs from sources external to the system under study. Output neurons send signals out of the system. The inputs and outputs of hidden neurons are not directly accessible. Suppose there are N neurons. These neurons can be numbered arbitrarily since each

neuron is essentially a simple processing element carrying out a simple process. This process receives inputs, computes an output value, and sends this value to other neurons.

2.3.2 State of activation

A representation of the state of the system is needed at time k. This is normally specified by a vector of N real numbers, X_k, representing the pattern of activation over the set of processing neurons. Each element of X_k stands for the activation of the neuron at time k (Figure 2.2).

The pattern of activation over the set of neurons captures what the system is representing at any time. Activation values may be continuous or discrete. If continuous, values may be bounded or unbounded. If discrete, values may take binary or a small set of values. Binary values are often denoted by {-1,+1}, or in some cases {0,+1}. Non binary discrete values, for example, could be restricted to the set {-1,0+1}.

2.3.3 Output function

Neurons interact by transmitting signals to neighboring neurons. The strength of their signal and the degree by which neighboring neurons are affected, is determined by the neuron's degree of activation. Associated with each neuron (Figure 2.2), there is an output function which maps the current state of activation to an output signal, OUT= F (NET), where $NET = X_kW$. This function is usually a threshold function, or a sigmoidal function.

2.3.4 Pattern of connectivity

Since neurons are connected to one another, it is the pattern of connectivity that constitutes what the system knows and determines how it will respond to arbitrary inputs. Normally, we assume that each neuron additively contributes to the input of the neurons to which it is connected. Each neuron's output is simply a weighted sum of the separate inputs from each of the connected neurons. The pattern of connectivity is normally represented by a weight matrix, W, with individual connection weights denoted by w_{ij} (Figure 2.3). Weight w_{ij} is positive if the input associated with it excites the neuron, negative if it inhibits the neuron, and zero if it has no direct connection to the neuron.

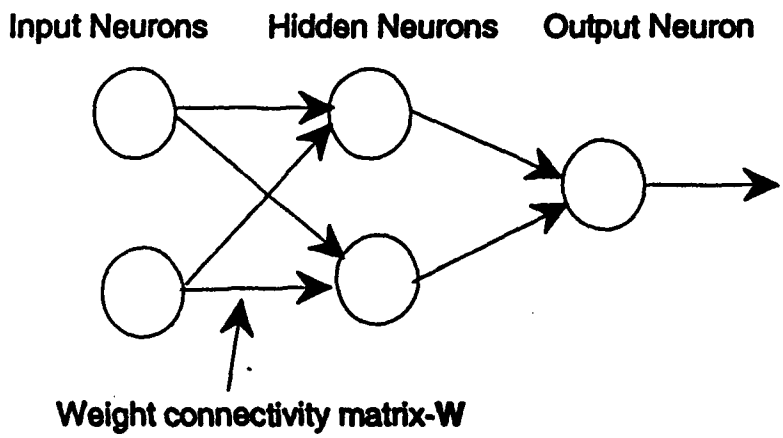


Figure 2.3. Neurons connected by weight matrices W.

The pattern of connectivity is very important because it determines what each neuron represents. How much information can be stored and how much serial

processing the network must perform is the fan-in and fan-out of each neuron[11]. Fan-in is the number of elements that either excite or inhibit a given neuron. Fan-out refers to the number of neurons affected by a single neuron.

2.3.5 Propagation rule

The rule of propagation takes the output vector representing the output values of the neurons and combines it using connectivity weight matrix W to produce a net input, NET, for each type of input (inhibitory or excitatory) into the next neuron. This is expressed as $NET = WX_k$.

2.3.6 Activation rule

This rule takes the combined net inputs of each type impinging on a particular neuron and combines them with the current state of the neuron to produce a new state of activation. In other words, function F takes X_k and vectors NET for each different type of connection and produces a new state of activation, ie $X_{k+1} = F(NET)$, where F itself is the activation rule. Activation functions may be a simple linear function, OUT = K(NET) where K is a constant; a threshold function O = 1 if NET > T and OUT = 0 otherwise, where T is a constant threshold value; or a sigmoid (squashing) function $OUT = 1/(1 + e^{-NET})$. These are by no means the only activation functions, but are used in many models described in the literature[1 through 12].

2.3.7 Learning rule

Learning rules involve modifying the patterns of connectivity as a function of experience. In principle this can involve three kinds of modifications:

1. Development of new connections
2. Loss of existing connections
3. Modifications of strengths of existing connections.

All rules of this type are variants of the Hebbian learning rule [5,11]. If a neuron i receives an input from another neuron j, and if both are highly active, weight w_{ij} from neuron i to neuron j should be strengthened. This is shown in equation (2.1).

$$w_{ij}(k+1) = w_{ij}(k) + \alpha\, OUT_i OUT_j \qquad (2.1)$$

where $w_{ij}(k)$ = the value of connection weight from neuron i to neuron j prior to adjustment, $w_{ij}(k+1)$ = the value of connection weight from neuron i to neuron j after adjustment, α = the learning-rate coefficient, OUT_i = output of neuron i and input to neuron j, and OUT_j = output of neuron j.

CHAPTER 3

ADALINE

The adaptive linear combiner, or non recursive adaptive filter, was developed by Widrow and Stearns [66] and is the basic building element used in many neural networks (Figure 3.1). The ADALINE functions as an adaptive threshold logic element. In digital implementation, an input vector $X_k = [x_0\ x_{1k}\ x_{2k}\ x_{3k} \ldots . x_{nk}]$ is applied at time k and is weighted by a set of coefficients which form a vector $W_k = [w_{0k}\ w_{1k}\ w_{2k}\ w_{3k} \ldots . w_{nk}]$. The ADALINE produces a scalar analog output or inner product $y_k = X_k^T W_k = W_k^T X_k$. Weight vector element w_{0k} is usually selected as the "bias" weight and is normally connected to a constant input $x_0 = +1$. A binary output, q_k, is then computed by the ADALINE based on scalar output y_k. Decisions are made by a two level quantizer, where $q_k = \text{SGN}(y_k)$ for the case of symmetric (+1,-1) inputs. For binary (0 or 1) inputs, $q_k = 1$ if $y_k => 0$, otherwise $q_k = 0$.

The desired response signal is used to train the neuron. During training, input patterns and desired responses are presented to the ADALINE. An adaptation algorithm adjusts the weights by minimizing error between the output and the desired output. For practicality, the adaptation process is oriented toward minimizing the mean-square value (MSE) or average power of the signal. The least squares adaptation algorithm or LMS [16,66] (Appendix 1) *or the Widrow-Hoff Delta Rule minimizes the sum of the* squares of the errors over the training set. The desired response and the components of X_k could be analog or binary. Once the weights are adjusted and the neuron is trained,

responses can be tested by applying a set of input patterns which contains elements not included in the training set. If the neuron responds correctly (with high probability), then it is said that generalization has taken place.

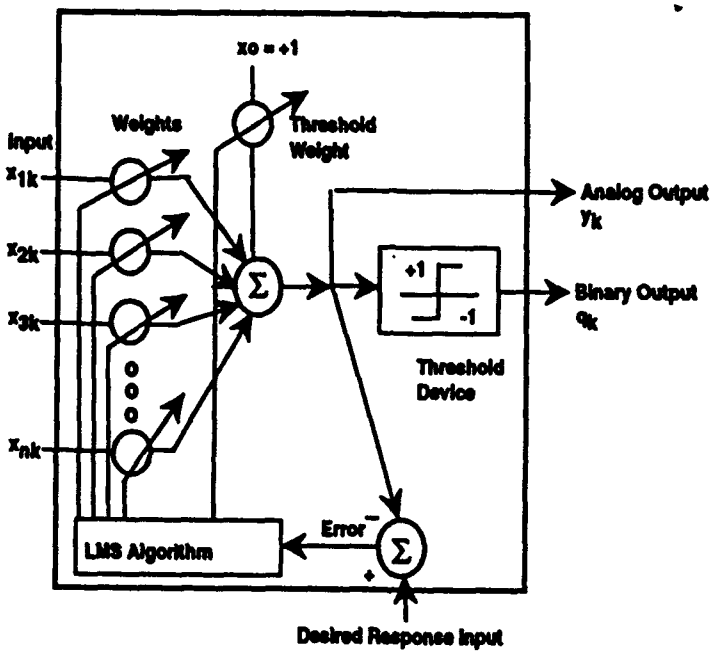


Figure 3.1. Adaptive Linear Combiner or ADALINE

3.1 Linear separability

Linear separability is useful for explaining how ADALINE neurons make decisions. It also illustrates the limitations of single layer networks. Consider a neuron with n binary inputs and one binary output. With n inputs, there are 2^n possible input patterns and 2^{2^n} boolean logic functions connecting n inputs to a single output. A single neuron with n binary inputs is capable of realizing only a subset of all possible boolean logic functions.

Figure 3.2 shows all possible binary inputs for a two-input neuron in pattern vector space. The coordinate axes are the components of the input pattern vector. Depending on the weight values, a critical thresholding condition occurs with the analog response , $y = 0$, that is:

$$y = x_1 w_1 + x_2 w_2 + w_0 = 0 \qquad (3.1)$$

which yields

$$x_2 = - (w_0 / w_2) - (w_1 / w_2)\, x_1 \qquad (3.2)$$

The three weights determine the slope $(-w_0 / w_2)$, the intercept $(-w_1 / w_2)$, and the side of the separating line that corresponds to a positive output.The opposite side of the line then corresponds to a negative output.

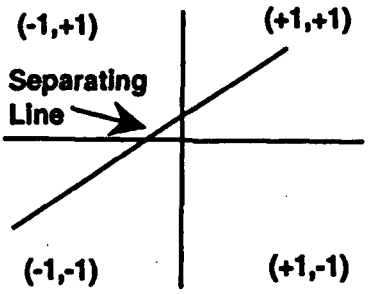


Figure 3.2

The binary inputs are tabulated in table 3.1 This is an example of a linearly separable function. For a nonlinearly separable function, no single line exists that can achieve the separation of the input patterns. An example is of a nonlinearly separable function is the EX-OR function (table 3.2).

TABLE 3.1

x1	x2	y
+1	+1	+1
+1	-1	+1
-1	-1	+1
-1	+1	- 1

TABLE 3.2

x1	x2	y
+1	+1	+1
+1	-1	- 1
-1	-1	+1
-1	+1	- 1

Table 3.3 summarizes the number of linearly separable functions for n inputs. It can be seen that the percentage of linearly separable functions decreases exponentially with increasing n and thereby limits the classification capability of single neurons (single layer networks).

n	2^{2^n}	Number of Linearly Separable Functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	4.3×10^9	94,572
6	1.8×10^{19}	5,028,134

Table 3.3.Number of Linearly Separable Functions

3.2 Networks

A single neuron can perform simple pattern detection functions. However, the power of neural computation comes from connecting neurons into networks (single or multi layer) [4,5,7,9,10,11] . In single and multilayer networks, W_k becomes a weight matrix rather than a weight vector. One can therefore connect many ADALINE's together to form MADALINE networks which have greater processing capability. Lippman explains that multilayer networks perform more general classifications by separating points contained in convex open or closed regions [4]. A convex region is described as having any two points which can be joined by a straight line that does not leave the region. A closed region is one in which all points are contained within a boundary (ie, circle). An open region has some points outside any defined boundary (ie, region between two parallel lines). To better understand convexity, consider a two layer (2-2-1) network with output neuron threshold set at 0.75. The output neuron receives signals with connection strengths of 0.5 from layer 2 neurons. It can be seen that the output neuron performs a logical "AND" function since both neurons in layer 2 need to be "1" for the output neuron to exceed its threshold. In Figure 3.3 it is assumed that each neuron in layer 1 subdivides the x-y plane, with one neuron producing an output of "1" for inputs below the upper line and the other producing an output of "1" for inputs above the lower line. A v-shaped region then results, in which the output neuron is one only over this v-shaped region. If enough neurons are included in the input layer, a convex polygon of any desired shape can be formed[5]. However, points not comprising a convex region cannot be separated from all other points in the plane by a two layer network. The

output of a neuron can also produce functions other than the logical "AND" by suitably choosing weights and thresholds. For continuous inputs, the network can subdivide the plane into continuous regions rather than separate discrete regions.

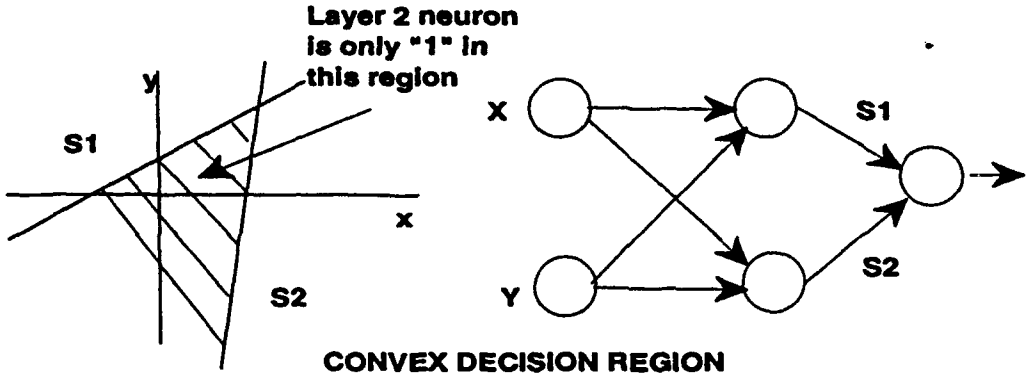


FIGURE 3.3. CONVEX DECISION REGION PRODUCED BY A 2-2-1 NETWORK

If another 2-2-1 network is used in conjunction with the first network, two separate convex decision regions are created. These two regions can then be joined by a third layer neuron as shown in Figure 3.4. Classification capacity is now limited only by the number of artificial neurons and by the weights. There are no convexity constraints[4,5].

It has been shown that a 3 layer network is capable of realizing any classification function [16]. A 3 layer network was therefore attempted classifying patterns. The first two layers were implemented optically since there were only two optical processors, and the output layer was done electronically in the computer memory.

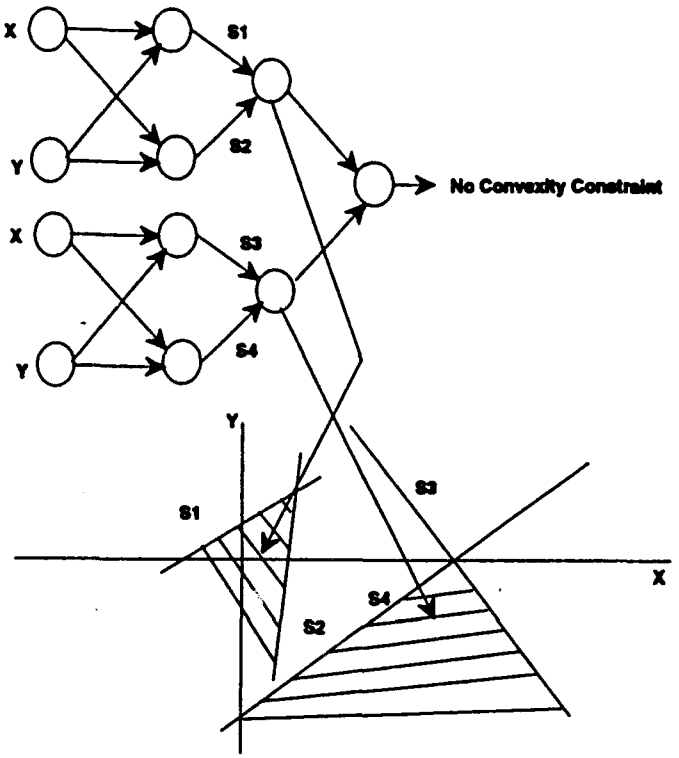


Figure 3.4. Three layer network shows no convexity constraints

CHAPTER 4

OPTICS FOR NEURAL NETWORKS

Optics offer many advantages which can be exploited for developing ANN's. One advantage is higher bandwidth over that of electronics because unlike electronics, there is no capacitance present in optical connections allowing for faster transmission of signals. Another advantage results from photons not interacting with one another unlike electrons (because of charge). Consequently, light beams may pass through one another without distorting the information carried [63].

However, a main disadvantage of optics is that there exist few devices which perform all optical signal processing functions, and these are still in development and not accessible. Until these devices are further developed (SEED)[63], electro optical devices provide a means by which signals can be applied electronically, converted to an optical equivalent for optical processing, and then converted back to an electronic signal which can be read or stored by a digital computer.

4.1 Polarization

Polarization is significantly important in optical computing because it enables some operations to be performed with very little energy loss, such as switching states or signal steering [63]. Polarization is also essential for using a Spatial Light Modulator (SLM) discussed in Section 4.3.

Electromagnetic waves such as light, have electrons oscillating transversely to the direction of wave propagation, z for example, in all directions of the x-y plane [63]. There are three forms of polarization: linear, circular, and elliptical [62,63].

A polarizer is an optical device whose input is natural or randomly polarized light and its output is some form of polarized light. Polarizers are categorized as linear,circular or elliptical and are all based on one of four fundamental physical mechanisms: dichroism or selective absorption, reflection,scattering, and birefringence or double refraction. When used as analyzers, polarizers allows light polarized in only one direction to pass. Polaroid HN 38 material is dichroic material which selectively absorbs one of the two orthogonal polarized components of an incident beam and is widely available [62].

4.2 Spatial Light Modulators

A spatial light modulator produces an output of light that is modulated over the plane perpendicular to the direction of propagation. The modulation is generally a change in the angle of polarization, which in turn can be translated into an amplitude modulation based on the angle of rotation and an output polarizer.

Modulation of the output beam is done by the beam's interaction with the active medium which is controlled by a field applied to it [33]. SLM's are constructed from either liquid crystal [35,39], ferro-electric liquid crystal [34], magneto-optic materials [37],or micro-channel plate [36]. Other types use deformable mirrors [40,41,42] and operate on the phase of the light and not its amplitude. Spatial Light Modulators can be

addressed optically or electronically. In electronic addressing , the optically active areas are defined as pixels and arranged as an n-by-n matrix. A computer stored image can then be written to the SLM via interface electronics by applying a voltage or current to the respective X-Y electrical connections.

SLM's perform optical signal steering, multiplication, and memory storage as illustrated in Figure 4.1. For multiplication, the SLM realizes a vector-matrix multiplication of vector **L** containing elements $\{L_1,L_2,L_3,L_4\}$ with matrix **A**, yielding vector **D** containing elements $\{D_1,D_2,D_3,D_4\}$, where $\mathbf{D}=\mathbf{L}^T\mathbf{A}$. If matrix coefficients are either "0" or "1" the SLM realizes an interconnection array routing signals from **L** to **D**.

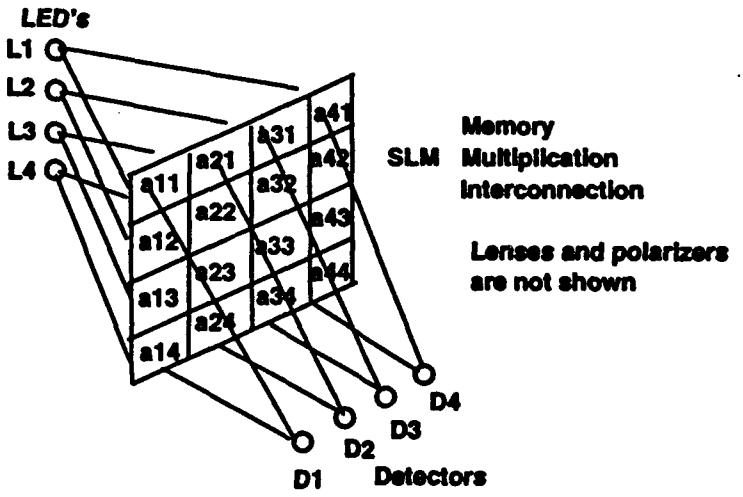


Figure 4.1. Spatial Light Modulator

The SLM's used in this thesis study were custom manufactured and constructed with liquid crystal material. Each SLM has a 16X16 weight matrix and is driven in the direct drive method with a IBM compatible computer controlling the pixel transmission level. Since the transmissive mode is used, input and an output polarizers are needed. The polarizing material is Polaroid HN 38.

4.3 Emitters

There are many forms of light emission but not all can be used in optical implementation of neural networks. A useful device is the light emitting diode or LED. Semiconductor lasers are of special interest for optical neural networks for the following reasons[63]:

- Devices are small, approximately 300 X 10 X 50 microns
- Voltages, currents, and power are low, typically 2V and 15-100 mA
- Efficiency is high, on the order of 20%
- High modulation rates, on the order of 10 GHz
- Devices interface easily with integrated optics and integrated electronics
- Low cost

Arrays of laser diode and light emitting diode sources provide an electronically addressable SLM with good intensity and equal spot illumination. Light is emitted only from diodes which have a voltage applied. Image information stored in a computer can be transferred to an optical network with such an array [63].

4.4 Detectors

Optical detectors convert optical signals into electronic signals. Detectors can be classified into four categories [61].

1. External photoelectric effect detectors. These include vacuum photodiodes and photomultipliers.
2. Internal photoelectric effect detectors. This covers a wide range of semiconductor devices in which photons are absorbed to produce charge carriers.
3. Thermal detectors. These use the direct open effect of absorbed radiation and include bolometers (resistance thermometers) and thermocouples.
4. Detectors in which a chemical change is initiated by optical radiation, such as in the human eye.

Semiconductor or junction photodiodes are primarily used in neural network optical detection [63]. A characteristic of these detectors is that if no bias is applied, they generate a photovoltaic voltage proportional to the amount of incident light.

CHAPTER 5

OPTICAL NEURAL NETWORK

Optical neural networks based on Vector-Matrix multipliers have been configured as associative memories [18, 20, 22, 23, 28, 29, 51], and pattern classifiers [22, 24, 25, 26, 27,30, 31]. The function of the weight mask is to produce a dynamically variable transmittance proportional to the desired connection weight. The use of cylindrical lenses for signal steering is not practical if this type of network is to be implemented at the micro-optical and micro-electronics level. Therefore crossbar card-readers which use waveguides for light distribution and collection were used.

The network consisted of 3 layers. The first two layers were realized optically with SLM's used as vector matrix multipliers. Each optical layer has a 16X1 input where one of the inputs served as a bias input,and a 16X1 output vector where one of the output elements is needed for input dependent thresholding (see Section 5.2). The third layer has a 16X1 input with one input serving as a bias input and a 3X1 output. Input patterns are presented as 5X3 2-D data mapped as a one dimensional 15X1 vector.

5.1 Unipolar vs Bipolar SLM arrangement

Most previously mentioned optoelectronic networks utilize one pair of SLM's per layer. One SLM stores positive weights and the other stores negative weights since one

cannot have "negative" light . Output detectors are then coupled to differential amplifiers which provide a weighted sum for each neuron. This technique increases network complexity and cost. In unipolar mode, one SLM is used per layer to realize both positive and negative weights. This was accomplished by means of a positive weight offset given to each connection.

5.2 Input Dependent Thresholding

For each output y_k, matrix W new weights are $W_{ik}^* = W_{ik} + p$, where p represents a positive offset. Because $y_k = \sum W_{ik}^* X_k = \sum (W_{ik} + p) X_k = \sum W_{ik} X_k + \sum p X_k$, the threshold function is dependent on the varying input. Positive offset p is determined from the spatial light modulator gray scale range. For example, if the gray scale range is 120 levels per pixel, the minimum negative weight value could be set at -59 and the maximum positive value could be set at 60 (because zero counts as a weight). Positive offset p is then required to have the value of 60 in order for the new weights to span a range of 1 to 120.

Hardware realization of input dependent thresholding is realized by having a threshold column in which each of the column's pixels are set to half of maximum contrast (Figure 5.1). The value of the photodiode detector which receives the weighted sum of the inputs through the threshold column now becomes the two level quantizer decision value.

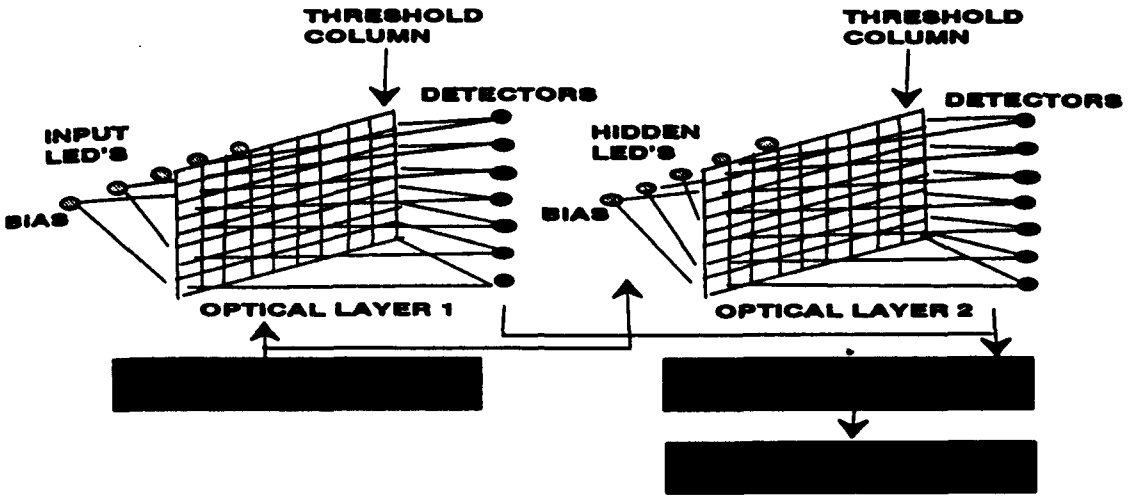


Figure 5.1. Input Dependent Thresholding Configuration

5.3 Driver and Interface electronics

The optical ANN system is functionally made up of seven sections. These are:

- Programmable liquid crystal display (LCD) drivers
- Spatial light modulators
- Light emitting diode or LED linear array stages
- Linear array photodiode receivers
- Analog signal conditioning
- Digital and analog interface
- Training software

A block diagram of the optoelectronic neural network (Figure 5.2) illustrates the signal flow of the system. The system's software controls an ADC board and a digital

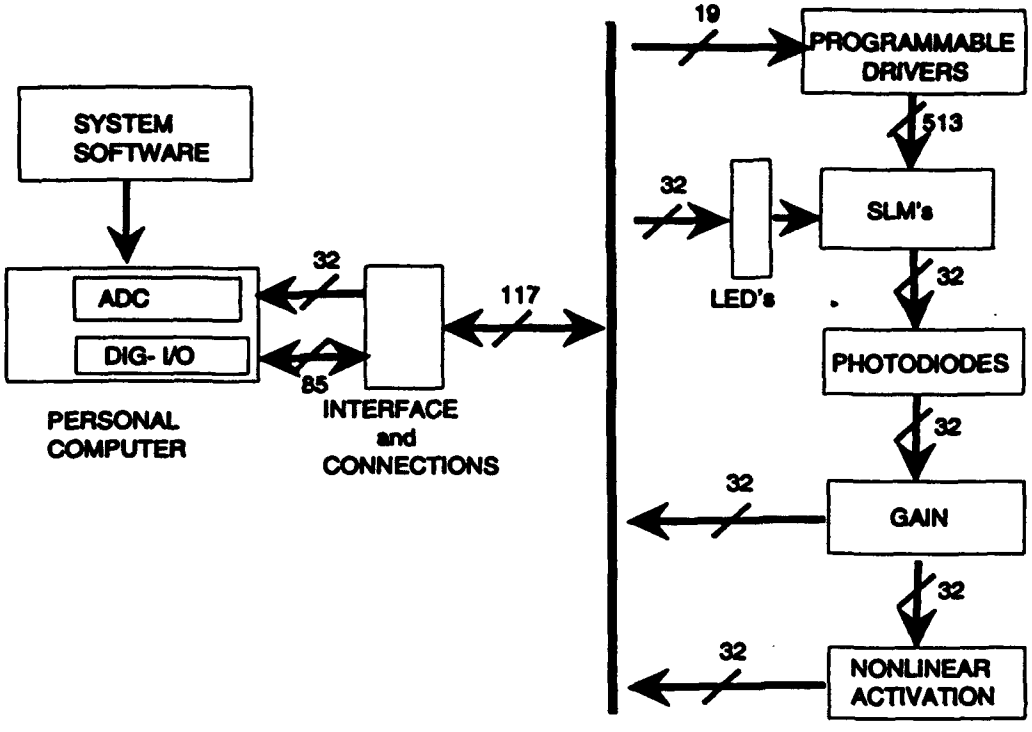


Figure 5.2. System overview

input/ output (I/O) board. The ADC board houses a multiplexed 48 channel 12 bit successive approximation analog to digital converter. The first 32 channels receive amplified photodiode signals which represent output vector analog elements. The digital I/O board consists of 96 channels of I/O capacity. This controls the two linear array LED boards which provide an input vector signal to the SLM's, the programmable SLM driver boards, and reads the outputs of the nonlinear activation function (comparators) boards. Except for the ADC and digital I/O boards, the system's hardware is housed in a 19 inch electronic cabinet.

5.3.1 Programmable drivers

Programmable drivers control the gray scale value of each pixel in the spatial light modulator. Each spatial light modulator contains 256 pixels. Since two modulators were used, a total of 512 independent programmable drivers were needed. The programmable drivers were designed with the goal of minimizing circuit complexity. This was realized with the design of a driver "cell" shown in Figure 5.3. The cell was then replicated 16 times to fit on a 12" x 12" printed circuit board (Appendix 2). The programmable driver printer circuit board provided 32 pixel driver signals. Sixteen driver printed circuit boards were manufactured for driving 512 individual pixels.

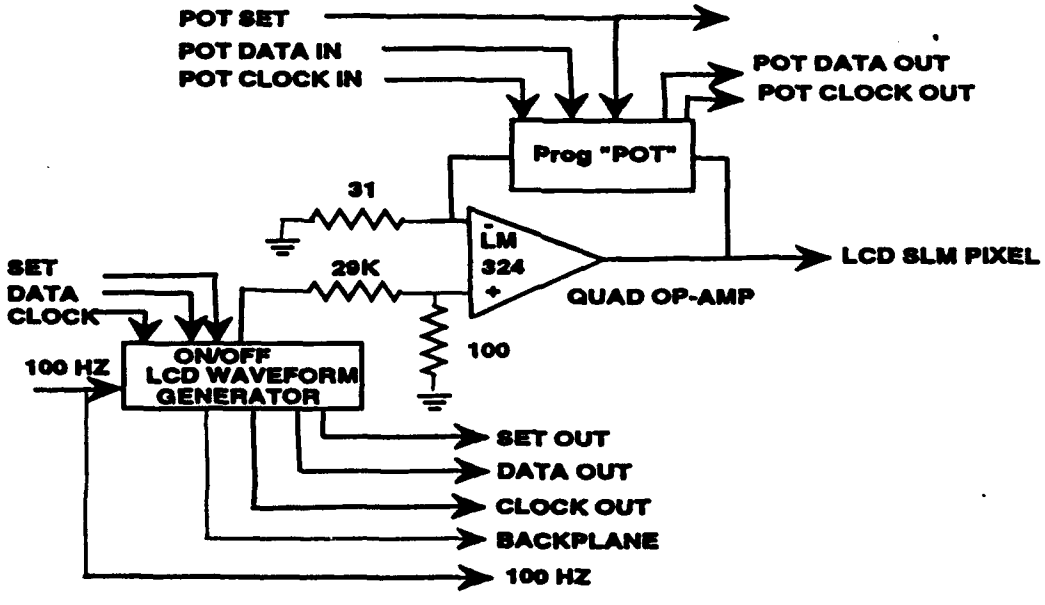


Figure 5.3. Driver Electronics (512 each)

Pixel transmission programming is accomplished in two steps. A total of 512

independent 0-5 volts DC LCD waveforms are generated through 16 cascadable LCD generator chips Hughes model H0438A. The H0438A is a CMOS/LSI circuit which drives field effect or dynamic scattering LCD displays.

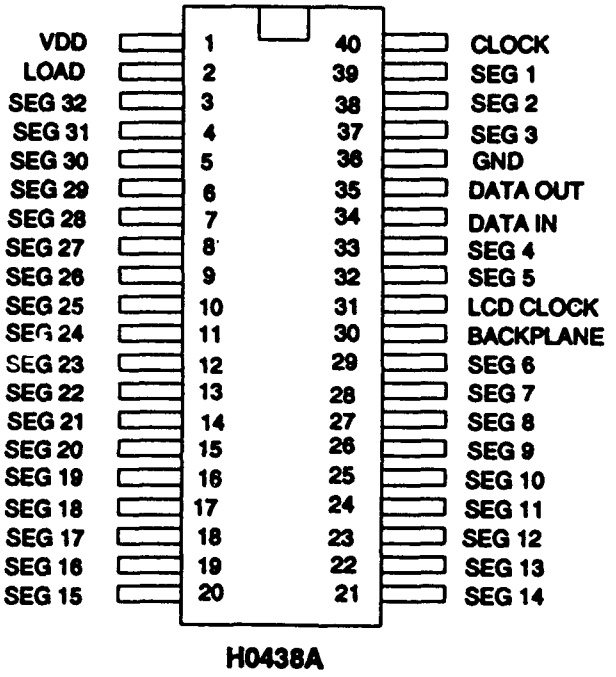


Figure 5.4. Hughes H0438A LCD Driver

Each H0438A (Figure 5.4) generates up to 32 pixel driving waveforms with correct phase and refresh rate for each LCD pixel. The H0438A contains a 32 bit static shift register, 32 latches, an LCD phase generator, and 32 segment (pixel) drivers (Figure 5.5). Information is written to the 32 bit shift register through a three wire serial port. The three wire serial port consist of Data In, Load, and Clock. Data is entered into the 32 bit static shift register only when Load is high. Data is entered while Load is high

through the Data In pin on the falling edge of Clock. Data is loaded starting with the value for segment 32 down to segment 1. A logic 1 on Data In causes a segment to be visible.The H0438A can be cascaded by connecting the Data Out of the first device to the Data In pin of the next device.The LCD clock signal input pins share a common driving signal.

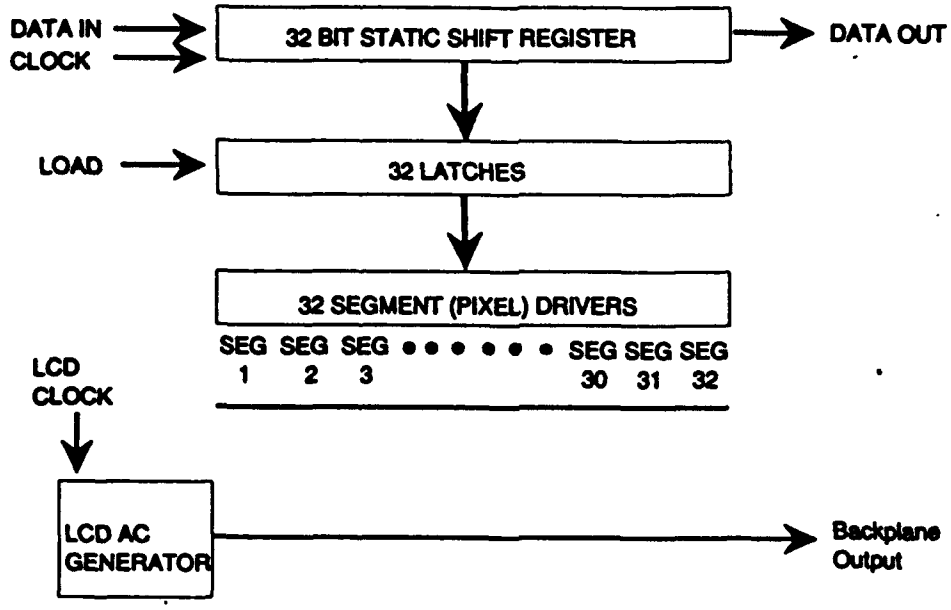


Figure 5.5. H0438A block diagram

Clock and Load signals are also shared by all cascaded devices. As data is clocked into the first device through Data In, previously stored information from the static shift register is clocked out bit by bit and into the next device.

If a DC voltage were applied to a pixel, it would destroy the pixel.The H0438A produces square waves with 5 volt amplitude. The frequency of the output square waves is equal to the LCD clock input frequency. A backplane output square wave is

also generated and serves as a phase reference. Figure 5.6 illustrates the purpose for having a backplane signal. If both backplane and segment waveforms are in phase, the pixel is essentially "off". If the segment signal is 90 degrees out of phase with the backplane signal, the pixel is "on".

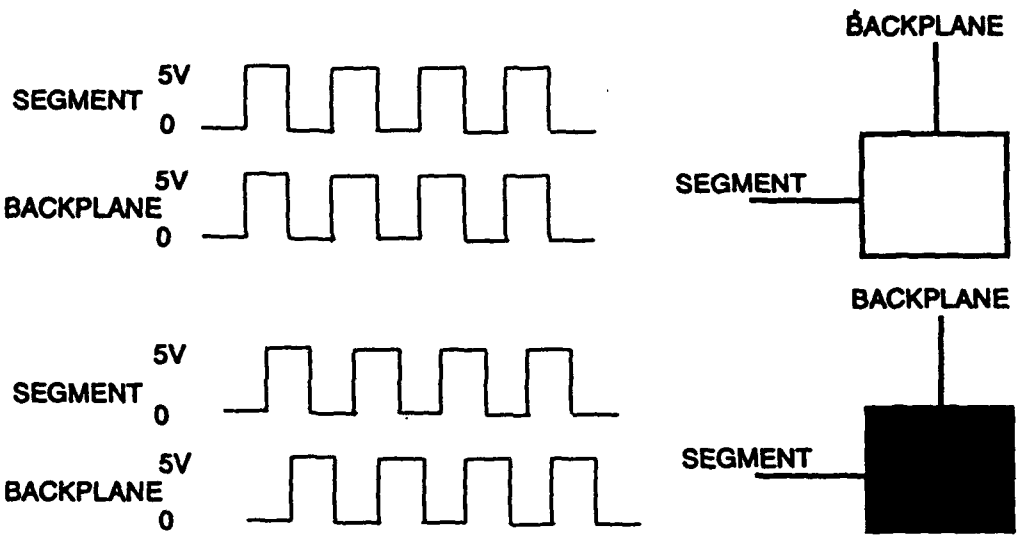


Figure 5.6. Segment (pixel) driving.

A LM555 timer chip is used to generate a 100 Hz LCD clock. Figure 5.7 describes the LCD clock generator. This circuit cannot realize a 50% duty cycle square wave, but a close approximation was adequate. Values for the circuit elements are: Ra = 1M, Rb= 220K, and Ct = 0.01 uF. These yield approximately a 100 Hz LCD clock.

The 0-5 volts DC waveforms are attenuated by a voltage divider resistor circuit. The non-inverting input receives a square wave with a 17 mV amplitude. Dallas Semiconductor DS1267 (Figure 5.8) dual programmable potentiometers are used in the

feedback of the non inverting gain stages and provide computer controlled gain for each waveform.

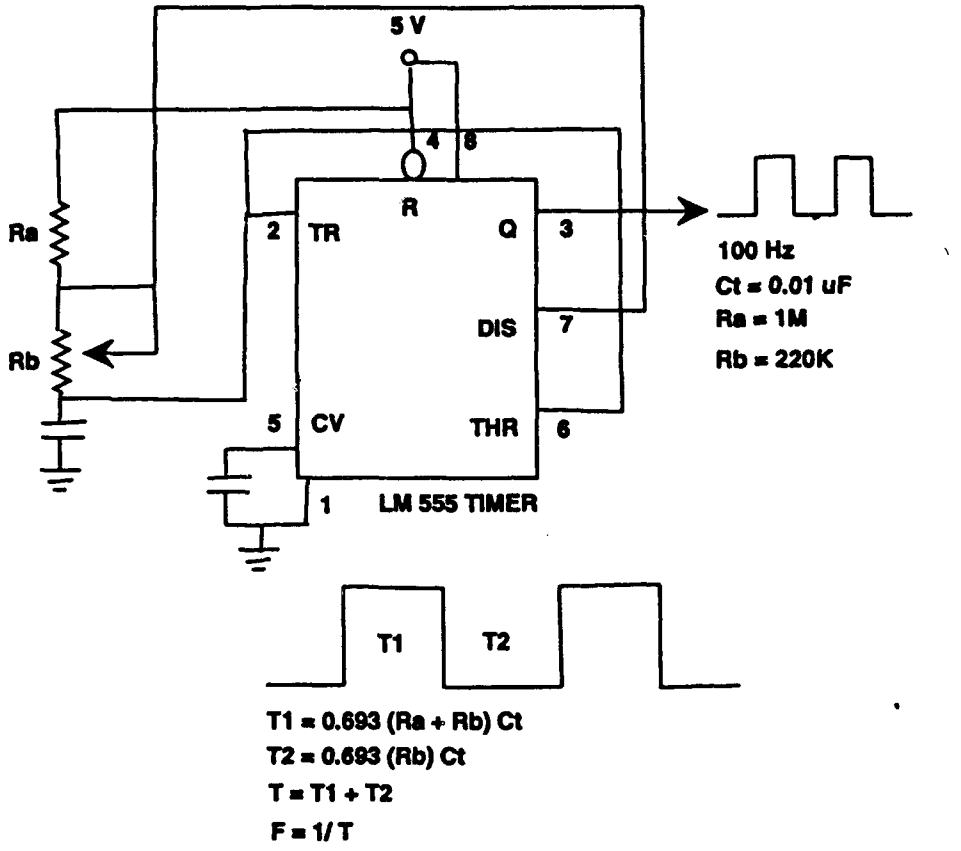


Figure 5.8. LCD clock generator

The DS1267 contains two potentiometers, each of which has its wiper set by a value contained in an 8 bit register (Figure 5.9). Each potentiometer consists of 256 resistors of equal value with tap points between each resistor and the low end. An 8 bit wiper register controls a 256-to-1 multiplexer (mux) that selects which tap point is connected to the wiper output.

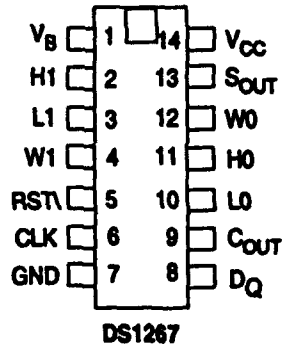


Figure 5.8. DS1267 Dual Potentiometer

Information is written and read from each wiper stack register via a 17 bit I/O shift register. The I/O shift register is serially loaded by a 3 wire serial port. The three wire serial port consist of RST\, DQ, and clock. Data is entered into the 17 bit shift register only when RST\ is high. Data is entered while RST\ is high through the DQ pin.

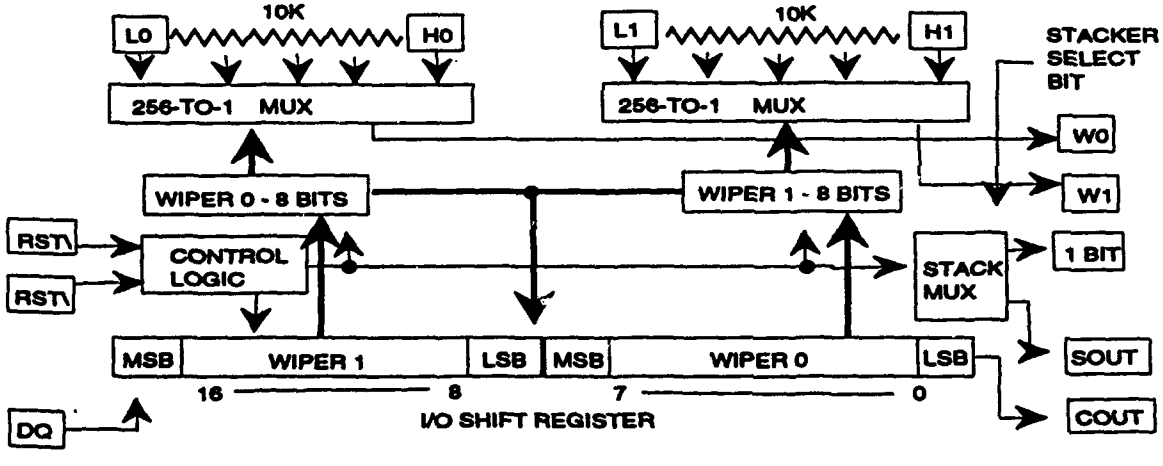


Figure 5.9. DS1267 Block diagram

The potentiometers always maintain their previous value until RST\ is set to a low level, which terminates data transfer. Valid data is clocked in on the low to high transition of the CLK input. Data is entered with the Stack Select bit starting the data stream, followed by the LSB of wiper 1, MSB of wiper 1, LSB of wiper 0 , and terminated by the MSB of wiper 0. A total of 17 bits constitutes valid data. As the data is clocked into the shift register, previous bits are shifted out bit by bit on the cascade serial port pin, C_{OUT}. By connecting the C_{OUT} to the DQ input of another DS1267, multiple devices can be daisy chained together.

The non-inverting gain stage has a maximum gain of (1 + 10K/31) = 257. This gives the programmable pixel driver an output amplitude range of 34 mV to 4.36 Volts.

The circuit was simulated with PSPICE and its output is shown in Figure 5.10. The circuit configuration allows for the potentiometer to be useful for almost its entire range of values. However, the operational amplifier saturates close to its positive supply rail value (5 volts) for the upper 1000 ohms (9,000-10,000 ohms).

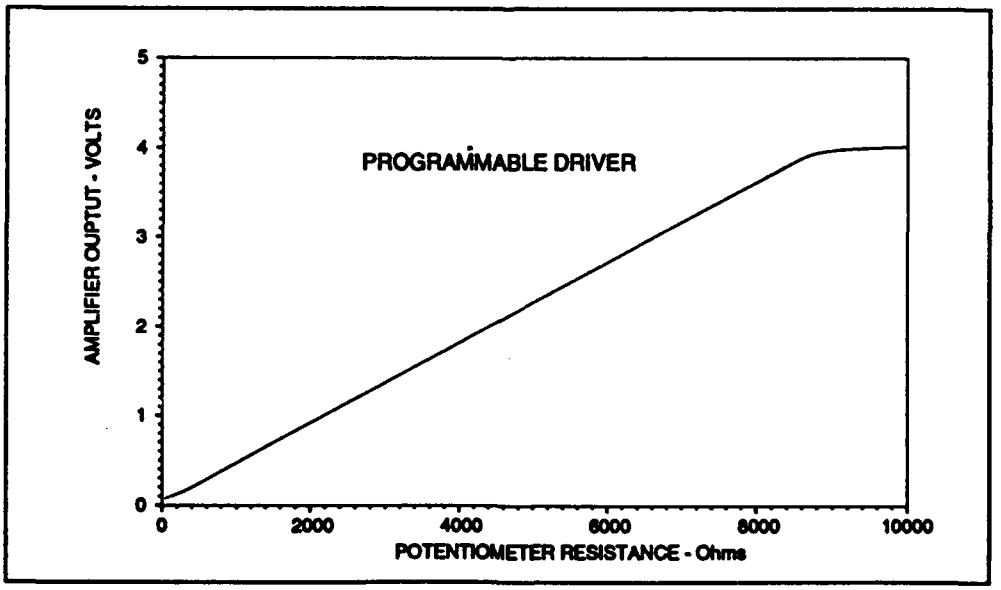


Figure 5.10. Driver output versus potentiometer value

The programmable drivers control each LCD pixel independently of its neighbors. Since the amount of polarization is dependent on the angle of rotation of the LCD material and the LCD material behaves nonlinearly with respect to the electric field applied to it, a linear range of approximately 120 gray levels (Figure 5.11) is achieved for each pixel.

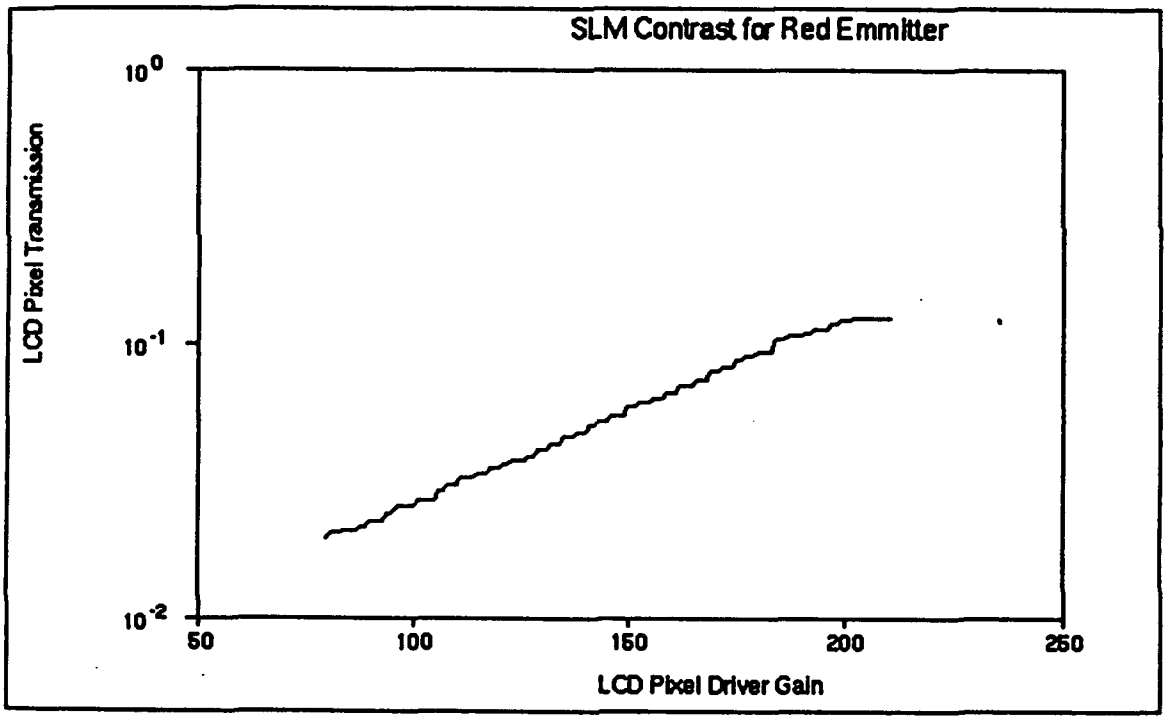


Figure 5.11. SLM pixel contrast

5.3.2 Spatial light modulator

The SLM 's are made of specially etched glass and were custom manufactured by

U.C.E. in Connecticut. Indium Tin Oxide conductors were etched on the glass and provide electrical contacts for each pixel. The arrangement of each pixel number is shown in Figure 5.12. The conductors are arranged in two set of 128 connections located on either end of the SLM. Figure 5.13 shows the correspondence of pixel number to weight matrix W element.

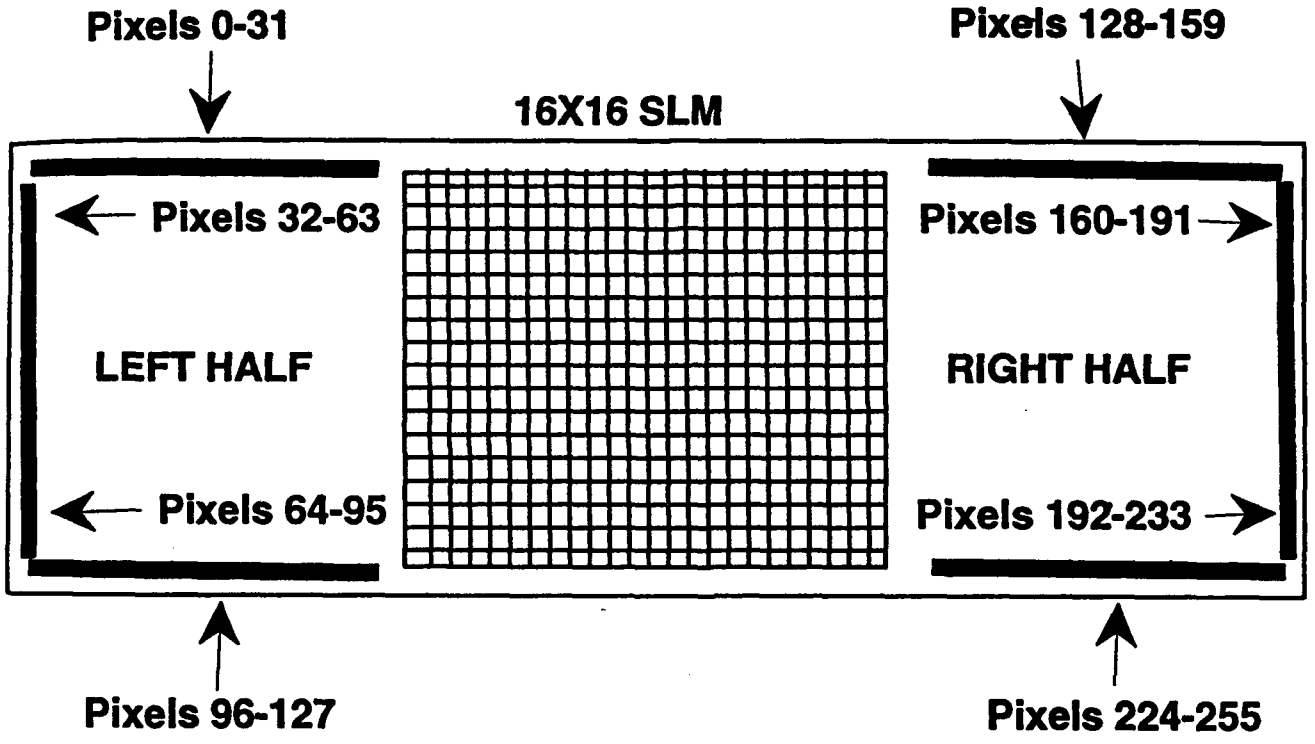


Figure 5.12. Indium Tin Oxide Conductor Placement

The Indium Tin Oxide conductors were bonded with .025" pitch Silicon Rubber heat seal connectors to two printed circuit boards for each SLM (Appendix 2). This was performed by ELFORM, a company located in Reno. Driver board to SLM board connections are made with 32 conductor ribbon cable assemblies.

W_{ij} i = 0 - 15 →

SLM LEFT HALF | SLM RIGHT HALF

j = 0 -15 ↓

1	2	3	4	5	6	7	8	136	135	134	133	132	131	130	129
9	10	11	12	13	14	15	16	144	143	142	141	140	139	138	137
17	18	19	20	21	22	23	24	152	151	150	149	148	147	146	145
25	26	27	28	29	30	31	32	160	159	158	157	156	155	154	153
33	34	35	36	37	38	39	40	168	167	166	165	164	163	162	161
41	42	43	44	45	46	47	48	176	175	174	173	172	171	170	169
49	50	51	52	53	54	55	56	184	183	182	181	180	179	178	177
57	58	59	60	61	62	63	64	192	191	190	189	188	187	186	185
65	66	67	68	69	70	71	72	200	199	198	197	196	195	194	193
73	74	75	76	77	78	79	80	208	207	206	205	204	203	202	201
81	82	83	84	85	86	87	88	216	215	214	213	212	211	210	209
89	90	91	92	93	94	95	96	224	223	222	221	220	219	218	217
97	98	99	100	101	102	103	104	232	231	230	229	228	227	226	225
105	106	107	108	109	110	111	112	240	239	238	237	236	235	234	233
113	114	115	116	117	118	119	120	248	247	246	245	244	243	242	241
121	122	123	124	125	126	127	128	256	255	254	253	252	251	250	249

WEIGHT MATRIX to PIXEL NUMBER MAP

Figure 5.13. Pixel number correspondence to SLM weights

5.3.3 LED linear array

Input and hidden vector light signals are provided by two linear arrays of 16 high output 680 nm peak light emitting diodes model A.N.D. Kilobright 180CRP. This wavelength was chosen because it provides the maximum contrast for HN 38 polarizing material. The LED's are driven with ULN 2001 LED driver IC's. Each ULN2001 IC consists of 7 driver transistors similar to 2N2222 transitors. Figure 5.14 illustrates the LED driving circuit used.

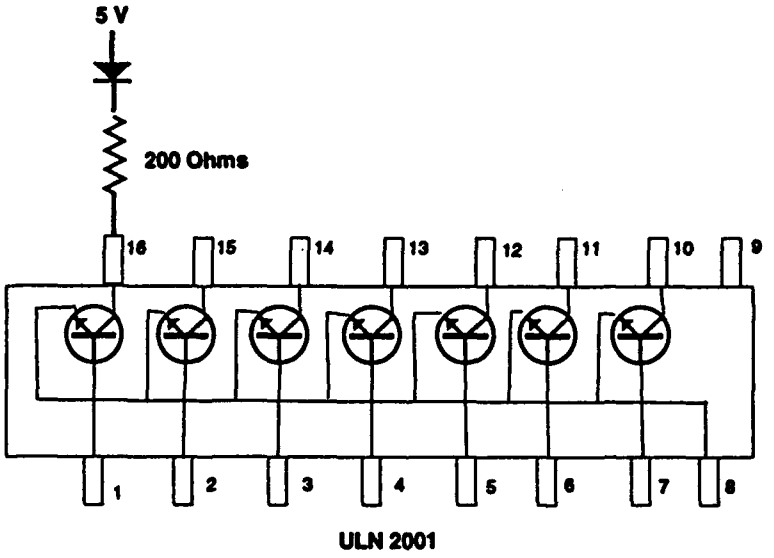


Figure 5.14. LED driver stage

The LED's were operated in binary fashion, i.e., they represented binary input vector elements. No gray levels were represented. The arrays were layed out on a printed circuit board with each LED spaced 5mm apart (Appendix 2). These are connected to a

digital interface board by 20 conductor ribbon cable.

5.3.4 Photodiode receivers

The output vector elements of layers 1 and 2 are realized electro optically with matched photovoltaic mode photodiodes, model UDT-10 from United Detector Technologies. These diodes are arranged in a linear array (Appendix 2) and provide a weighted sum of each input. Two printed circuit boards house the photodiode receivers, one for each layer. Each board is mounted next to the output image plane of the SLM matrix-vector multiplier. The photodiodes are spaced 5mm apart on the printed circuit boards. Output signals are then fed into corresponding analog conditioning boards with 20 conductor ribbon cables.

5.3.5 Analog signal conditioning

The photodiodes outputs cover a functional range of 40 mV to 340 mV. Each output vector is fed into a signal conditioning board which provides gain. The boards consists of LM324 operational amplifier non-inverting gain stages operating in photovoltaic mode (Figure 5.15).

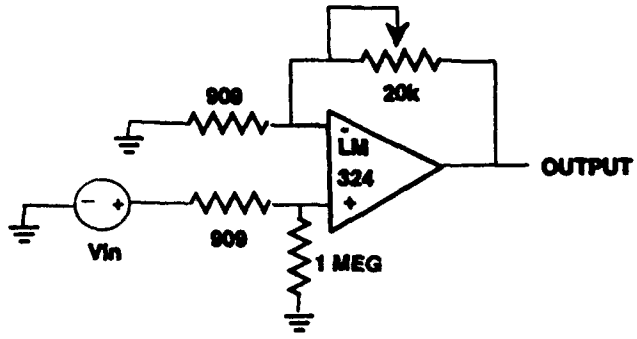


Figure 5.15. Photodiode amplifier

The non-inverting gain stage circuit was simulated with PSPICE. The simulation result is shown in Figure 5.16 below.

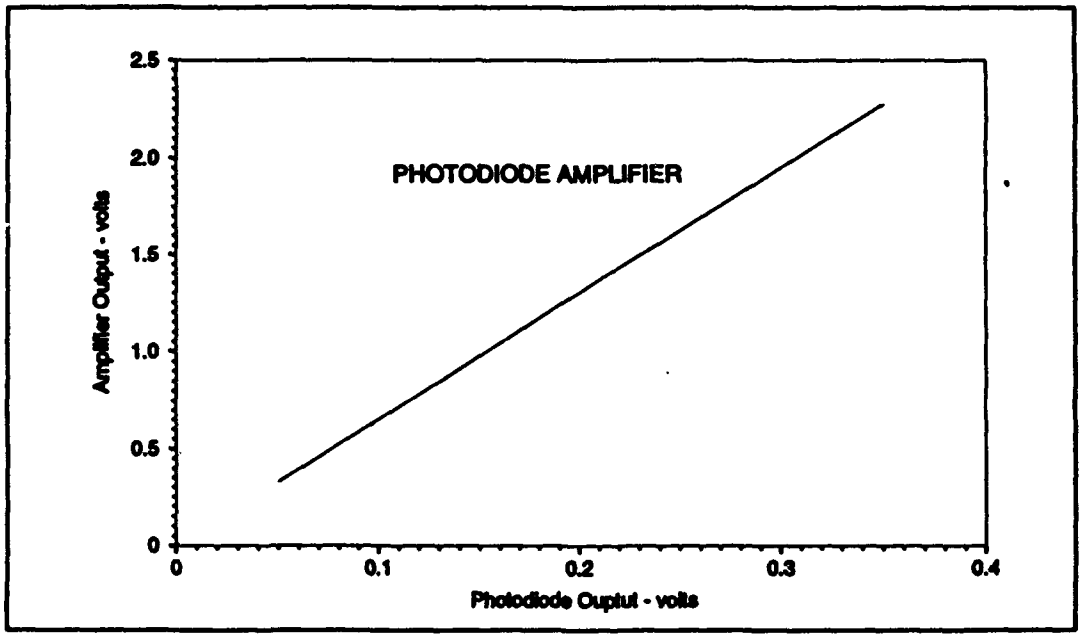


Figure 5.16. PSPICE simulation graph of photodiode gain stage

There are 16 amplifiers on each board. The output of the amplifiers are connected to analog comparator boards by 20 conductor ribbon cables, and

to a 50 pin ribbon cable connector from the ADC board.

The analog comparators (Figure 5.17) utilize LM311 precision comparators and realize the threshold activation function, i.e. the output is low when Vin <= Vref and high otherwise. A PSPICE simulation of comparator circuit illustrates a threshold activation function in Figure 5.18.

The output of the comparators are digital and are read by the computer through the digital interface board. The ADC is only used to read analog values which are needed during the training phase for rank ordering of the optical neurons. The ADC is not used for testing the network.

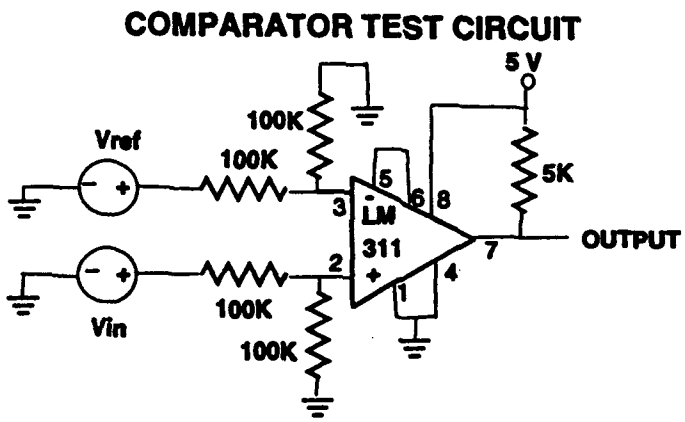


Figure 5.17. Nonlinear activation circuit

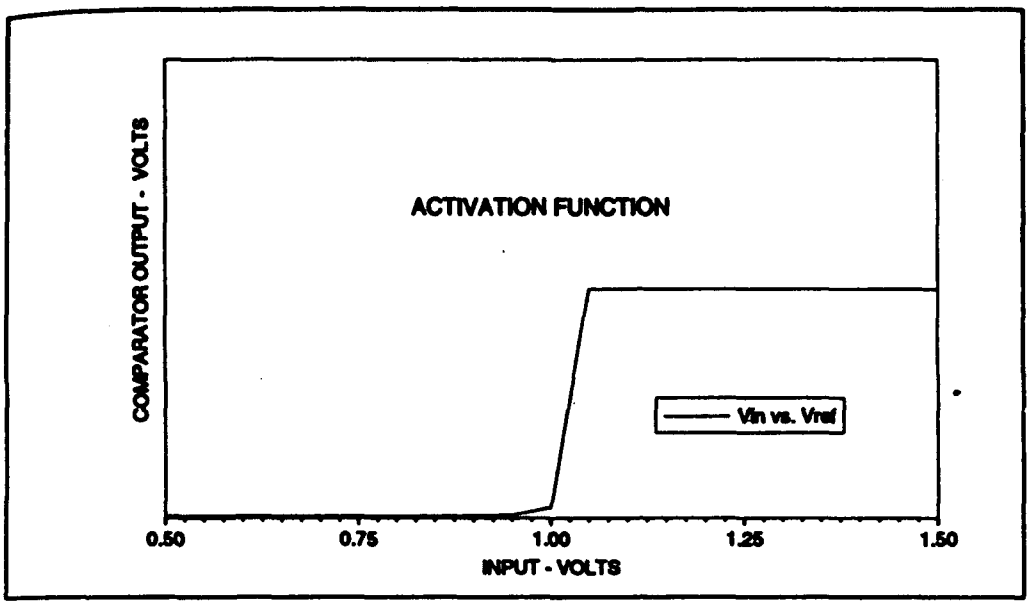


Figure 5.18. PSPICE simulation of circuit in Figure 5.9

5.3.6 Analog and digital interface

The analog interface is realized by a 48 channel ADC computer plug in board from Computer Boards, Inc. The board was located at base address 320 hex. It was used to read 32 channels of photodiode data. The values of the photodiodes represented the analog output of each neuron. These values were required for training the network.

The ADC board contains a 12 bit converter, a multiplexer , and 48 sample and hold amplifiers. The sample and hold amplifiers have a 15 us acquisition time to 0.01% of input signal value. Figure 5.19 provides connector pinout assignment for the ADC board and the photodiode amplifier outputs.

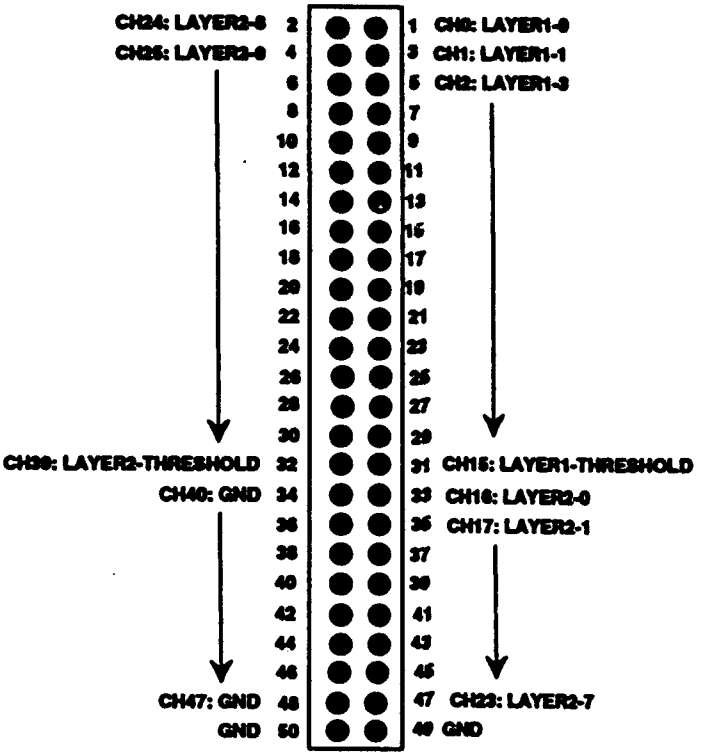


Figure 5.19. ADC connector pinout assignment.

The ADC board has 4 addresses which provide all necessary controls to the board's functions. The following code reads the first 16 channels of data corresponding to output layer 1 detectors.

```
/*******DEFINITIONS**************************************************************/
/******ADC  BOARD***************************************************************/
#define ADC_BASE        0x0320
#define TRIGGER         0
#define SET_GAIN        (ADC_BASE + 3)
#define GAIN             3
#define ADC_CONVERT   (ADC_BASE + 1)  /* 12 BIT CONVERSION */
#define SET_CHANNEL   (ADC_BASE + 2)
```

```
#define READ_LOW        ADC_BASE
#define READ_HIGH       (ADC_BASE + 1)
#define EOC             (ADC_BASE + 2)
 void ADC_LAYER1(void)
unsigned int channel;
int low_byte,high_byte,adc_counts;
unsigned char check_EOC;
double analog;
/*****Read layer 1 channels****/
for (channel = 0; channel < 16; channel++){
   outportb(SET_CHANNEL,channel);
   outportb(ADC_CONVERT,TRIGGER);
   do
    {
    check_EOC=inportb(EOC);
    printf("EOC = %u \n",check_EOC);
    }
   while (check_EOC > 127);
   low_byte=inportb(READ_LOW);
   high_byte=inportb(READ_HIGH);
   adc_counts=((high_byte*16) + (low_byte/16));  /*TOTAL ADC COUNTS */
   Hidden1_analog_value[channel] = adc_counts * .00122; /* 12 BIT CONVERSION 0 -5 VOLT RANGE */
  }
}
```

The digital interface is realized in two parts. A Computer boards Inc. CIO-DIO

96 digital I/O computer plug in board located at base address 300 hex provided digital lines for programming H0438A LCD waveform generators and DS1267 potentiometers, controlling individual LED's, and reading comparators. These signals were then distributed accordingly by a breakout board (Figure 5.19) which also provided a 100Hz clock for the H0438A waveform generators.

The CIO-DIO 96 is a 96 line digital I/O board controlled by 4 82C55 TTL level digital I/O chips. Each chip contains 3 data and one control register occupying 4 consecutive computer address locations (Figure 5.20). The CIO-DIO 96 needs 16 consecutive computer I/O address locations to function without conflicting with other devices. There are two connectors on the board denoted by P1 and P2 in Figure 5.20. These provide connections of external digital devices with the CIO-DIO 96.

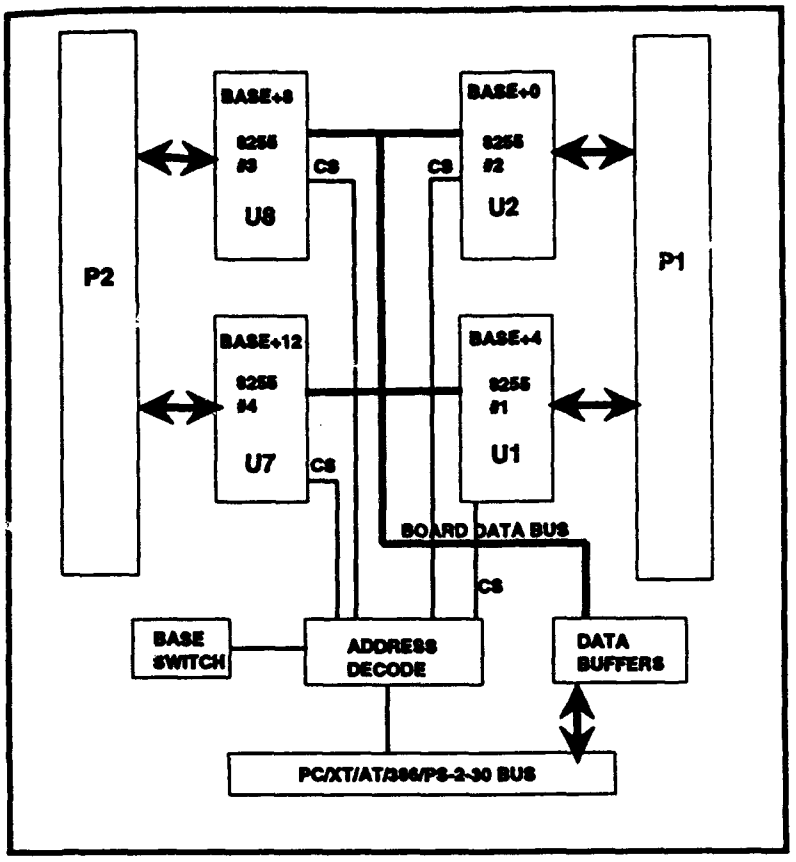


Figure 5.20. CIO-DIO 96 Block diagram

The CIO-DIO 96 has simple control and data register programming. Table 5.1 summarizes control and data register functions.

ADDRESS	READ FUNCTION	WRITE FUNCTION
BASE+0	Port A Input of 8255 #1	Port A Output
BASE+1	Port B Input	Port B Output
BASE+2	Port C Input	Port C Output
BASE+3	None. Read back on 8255	Configure 8255 #1
BASE+4	Port A Input of 8255#2	Port A Output of 8255 #2
BASE+5	And so on....	

Table 5.1. CIO-DIO 96 data and control register functions

The 8255 chips can be configured in 5 different modes of operation. Mode 0 provides simple I/O capability with no handshaking since the software controls all I/O events. The following C code illustrates Mode 0 I/O programming.

```
void MODE_0(void)
{
  outportb(BASE+3,128);     /* Mode 0 Output FOR U2*/
  outportb(BASE+7,128);     /* Mode 0 Output  FOR U1*/
  outportb(BASE+11,155);    /* Mode 0 Input  FOR U8*/
  outportb(BASE+15,137);    /* Mode 0 Input/Output FOR U7*/
}
```

Connectors P1 and P2 are connected to the interface board through two sets of 50 conductor ribbon cables. The pinout assignemt is shown in table 5.2. LED's are numbered 0 to 31 indicating that the first 16 drive the input vector and the remaining 16 drive the hidden layer 2 input vector. Likewise, comparator outputs are labeled PHOTO 0 to PHOTO 31 where PHOTO 0 -15 connect the first layers comparator output to the computer and the remaining 16 connect the comparators evaluating hidden layer 2's output. LCD clock 1, LCD Load 1, Pot Clock 1, Pot Data 1, and Pot Load 1, refer to weight layer 1 drivers. The others to weight layer 2 drivers. Programmable driver boards 1 through 16 are programmed with lines B0-B7, and A0-A7 of U7.

CONNECTOR P1				CONNECTOR P2			
U1		U2		U7		U8	
1		25	A7: LED23	1	A6:BOARD16DATA	25	A7:PHOTO23
2		26	A6: LED22	2	A5:BOARD15DATA	26	A6:PHOTO22
3		27	A5: LED21	3	A4:BOARD14DATA	27	A5:PHOTO21
4		28	A4: LED20	4	A3:BOARD13DATA	28	A4:PHOTO20
5	A3: POT2 CLOCK	29	A3: LED19	5	A2:BOARD12 DATA	29	A3:PHOTO19
6	A2: LCD2 CLOCK	30	A2: LED18	6	A1:BOARD11 DATA	30	A2:PHOTO18
7	A1: POT1 CLOCK	31	A1: LED17	7	A0:BOARD10 DATA	31	A1:PHOTO17
8	A0: LCD1 CLOCK	32	A0: LED16	8	B8: BOARD9 DATA	32	A0: PHOTO16
9	B7: POT2 LOAD	33	B7: LED15	9	B7: BOARD8 DATA	33	B7: PHOTO15
10	B6: POT2 DATA IN	34	B6: LED 14	10	B6: BOARD7 DATA	34	B6: PHOTO14
11	B5: POT1 LOAD	35	B5: LED 13	11	B5: BOARD6 DATA	35	B5: PHOTO13
12	B4: POT1 DATA IN	36	B4: LED12	12	B4: BOARD5 DATA	36	B4: PHOTO 12
13	B3: LCD2 LOAD	37	B3: LED 11	13	B3: BOARD4 DATA	37	B3: PHOTO11
14		38	B2: LED 10	14	B2: BOARD3 DATA	38	B2: PHOTO10
15	B1: LCD1 LOAD	39	B1: LED9	15	B1: BOARD2 DATA	39	B1: PHOTO9
16		40	B0: LED8	16	B0: BOARD1 DATA	40	B0: PHOTO8
17	C7: LED31	41	C7: LED7	17	C7:PHOTO31	41	C7: PHOTO7
18	C6: LED30	42	C6: LED6	18	C6:PHOTO30	42	C6: PHOTO6
19	C5: LED29	43	C5: LED5	19	C5:PHOTO29	43	C5: PHOTO5
20	C4: LED28	44	C4: LED4	20	C4:PHOTO28	44	C4: PHOTO4
21	C3: LED27	45	C3: LED3	21	C3:PHOTO27	45	C3: PHOTO3
22	C2: LED26	46	C2: LED2	22	C2:PHOTO26	46	C2: PHOTO2
23	C1: LED25	47	C1: LED1	23	C1:PHOTO25	47	C1: PHOTO1
24	C0: LED24	48	C0: LED0	24	C0:PHOTO24	48	C0: PHOTO0
		49	5V			49	5V
		50	GND			50	GND

Table 5.2. CIO-DIO 96 pin assignment

5.3.7 System software

The system software controls both analog and digital plug in boards, and trains the network. The system software consists of 12 separate hierarchically shown in Figure 5.12. **Menu3.c** provides the user with training options and parameters. **Include3.c** contains all headers, definitions, function prototypes, and global variables used or called by all other files. **Uniform.c** contains functions for

uniformly distributing network weights during weight initialization. **Rank.c** contains function for array indexing and ranking. **Network3.c** saves and recalls network weights for training and testing of patterns. **File.c** provides file input and output capability. **Train.c** provides a menu for training the network. **Test3.c** tests

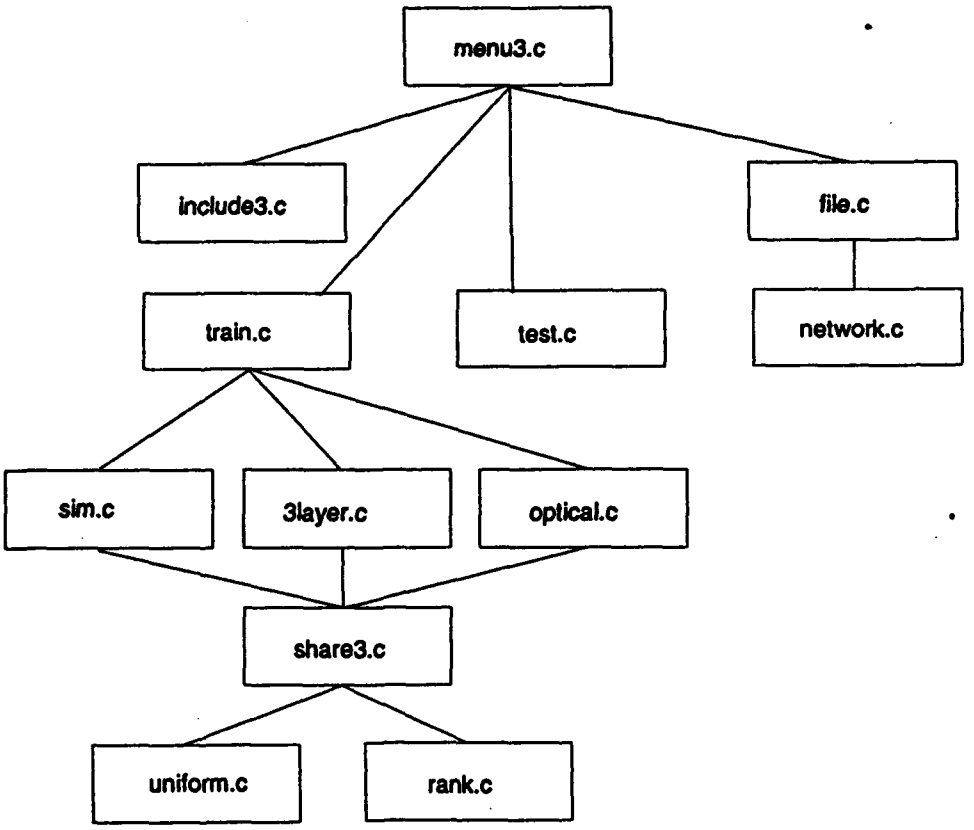


Figure 5.12. Software hierarchy

patterns after network has been trained. **Share3.c** provides MR2 training rule functions shared by optical system software, optical simulation software, and by traditional MR2 algorithm training. **Driver3.c** provides hardware digital and

analog input and output functions. **Optical3.c , Sim3_d.c** and **3layer.c** are the actual MR2 training software for the three systems. Detailed system software is found in Appendix 4. The files are listed in "loose" pseudocode style [65] below.

```
/*************************************************************************
PROGRAM : MENU3.C
DESCRIPTION: Provides simple menu interface for Optical ANN. Allows user
to select various modes of training.
*************************************************************************/
get include3 header and definition file
/*************************************************************************
SYSTEM PARAMETERS function asks user for choice of system (Optical or
Software only) and for weight distribution bounds.
*************************************************************************/
function system_parameters()
/*************************************************************************
Main Program
*************************************************************************/
integer main()
initialize variables
/*************************************************************************
Display the ANN Menu
*************************************************************************/
display Optical ANN Main Menu
    get choices from
```

1. System parameters

2. File menu

3. Train menu

4. Test menu

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

FILE: include3.c

DESCRIPTION: Contains headers, definitions, and global variables used in the

optical neural network system simulation. Refer to source code since it is

straighforward..

/*************************************************************************

FILE:share3.c

DESCRIPTION: Contains functions shared by simulation and optical

hardware during training phase.

/*************************************************************************

INITIALIZE WEIGHTS function uniformly distributes weight values in each

of the weight layers.

*************************************************************************/

function initialize_weights()

Weight_layers 1, 2 and 3 [i][[] = uniform_dist(Min,Max);

/*************************************************************************

INITIALIZE adapted function clears hidden neuron adapted value.

*************************************************************************/

function initialize_adapted()

Initialize Hidden vectors adaptation control variable at beginning

of training

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

BINARIZE_INPUTS function reads msb and lsb input vector values and assigns a corresponding +1 or 0. No symmetry is used. This is only used to calculate hidden neuron analog values.

**********/

function binarize_inputs(unsigned char lsb,unsigned char msb)

/**********

GENERATE OUTPUT NEURONS function calculates the weighted sum for each output neuron.

**********/

function generate_output_neurons()

Perform matrix-vector multiplications. Used only for simulation since hardware realizes the vector-matrix multipliers.

/**********

HAMMING DISTANCE function looks at the number of differences in the actual versus desired output vector.

**********/

unsigned integer hamming_distance(integer pattern_number)

{

Examine output vector and desired output vector element by element. For every nonmatching output, the Hamming number is increased by one.

}

/**********

TEST OUTPUT function looks for the test output neuron.

**********/

Like Hamming distance above, except uses test .

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

RANK HIDDEN1 NEURONS function ranks the value of the hidden neurons in

ascending order.

*****************************************************************************/

function rank_hidden1_neurons()

{

Ranks order neurons in Layer 1 using ranking algorithm found in numerical Recipes.

}

/*****************************************************************************

RANK HIDDEN2 NEURONS function ranks the value of the hidden neurons in

ascending order.

Same as above, but for layer 2 instead.

*****************************************************************************

SAVE CURRENT WEIGHTS1 function stores weight layer1 values.

*****************************************************************************/

function save_current_weights1()

Saves weights from layer 1 in temporary array

*****************************************************************************

RESTORE WEIGHTS1 function restores weight layer1 values.

*****************************************************************************/

function restore_prior_weights1()

Restore previous weight layer 1 values for this particular training case.

*****************************************************************************

SAVE CURRENT WEIGHTS2 function stores weight layer2 values.

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

```
function save_current_weights2()
Same as above, but for layer 2
/**********************************************************************
RESTORE WEIGHTS2 function restores weight layer2 values.
**********************************************************************/
function restore_prior_weights2()
Same as above, but for layer 2
/**********************************************************************
SAVE CURRENT WEIGHTS3 function stores weight layer3 values.
**********************************************************************/
function save_current_weights3()
same as above, but for layer 3
/**********************************************************************
RESTORE WEIGHTS3 function restores weight layer3 values.
**********************************************************************/
function restore_prior_weights3()
Same as above, but for layer 3
/**********************************************************************
ADAPT WEIGHTS1 function adapts one neuron at a time until its binary value
is inverted.
**********************************************************************/
function adapt_weights1(integer neuron_number)
Adapt neurons in layer 1 using MR 2 rule criteria
Uses singlets1, Doublets1, Triplets 1, and Quadlets 1 to minimize (train)
output error
```

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

ADAPT WEIGHTS2 function adapts one neuron at a time until its binary value

is inverted.

**********************************************************************/

function adapt_weights2(integer neuron_number)

Same as above, but for layer 2

/**********************************************************************

SINGLETS function adapts one neuron at a time. If change results in

reducing Hamming Distance then change is kept. Otherwise weights are

restored to original values.

**********************************************************************/

function singlets (integer pattern_number)

/*Rank analog neurons in order closest to zero*/

rank_hidden2_neurons();

/**********************************************************************

Adapt in direction to change corresponding binary output

of hidden layer

**********************************************************************/

Test for Hamming distance after each iteration

/**********************************************************************

DOUBLETS function adapts two neurons at a time. If change results in

reducing Hamming Distance then change is kept. Otherwise weights are

restored to original values.

**********************************************************************/

function doublets(integer pattern_number)
```

```
Same as above, but adapt two neurons at a time
/**************************************************************************
TRIPLETS function adapts three neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
**************************************************************************/
function triplets(integer pattern_number)
Same as above, but adapt three neurons at a time
/**************************************************************************
QUADLETS function adapts three neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
**************************************************************************/
function quadlets(integer pattern_number)
Same as above, but adapt four neurons at a time.
/**************************************************************************
ADAPT WEIGHTS 3 function adapts third layer neurons.
**************************************************************************/
function adapt_weights3(integer neuron_number)
Adapts output neurons that differ from target vector
Uses Adapt Output Layer below
/**************************************************************************
ADAPT OUTPUT LAYER function adapts one neuron at a time until its binary
value is inverted.
**************************************************************************/
function adapt_output_layer(integer pattern_number)
```

```
Uses Hamming distance to test for output error, then simply reads
output and tags the ones that need to be changed (inverted).
/**************************************************************************

FILE: NETWORK3.C
DESCRIPTION: Saves and recalls weight values from a user specified file.
**************************************************************************/

/**************************************************************************
SAVE WEIGHTS LAYERS saves Weight_layer1 and Weight_layer2 arrays to a file.
GET WEIGHT LAYERS retrieves weight layers 1 and 2 from a file and assigns
them to Weight_layer1 and Weight_layer2 arrays.
/**************************************************************************

FILE : SIM3_D.C
DESCRIPTION: Simulates Optical ANN hardware and trains with MR2 Rule.
Incorporates input dependent dynamic thresholding.
**************************************************************************/

/**************************************************************************
DYNAMIC THRESHOLD function calculates the threshold sum for each input
pattern through column sixteen.
**************************************************************************/

double dynamic_threshold(integer input[16])
Calculate weighted output sum of all input vector elements times a constant
value.
/**************************************************************************

WRITE SIM1 WEIGHTS function converts the weights and biases integero usable
forms for the  optical simulation weights
```

```
/*************************************************************************
WRITE SIM2 WEIGHTS function converts the weights and biases into usable
forms for the  optical simulation weights
/*************************************************************************
Simulate LAYER 1 NEURONS function calculates the weighted sum for each
neuron in layer 1.
/*************************************************************************
SIMULATE LAYER 2 NEURONS function calculates the weighted sum for each
neuron in layer 2.
/*************************************************************************
Madaline Rule Two Algorithm Training. The same for optical hardware
and  symmetric MR2 rule (3layer.c) so I wont repeat documentation.
*************************************************************************/
function simulate()
/* Uniform distribution of weights */
initialize_weights();
/*****Begin training*********************************************************/
for (outer_loop=0;outer_loop<iterations;outer_loop++){
for (pattern_counter=0;pattern_counter<24;pattern_counter++){
    initialize_adapted();
/*******Present input vector*************************************************/
    binarize_inputs(Letter_lsb[pattern_counter], Letter_msb[pattern_counter]);
/*******Check output vector and calculate Hamming Distance*****************/
    simulate_layer1_neurons();
    simulate_layer2_neurons();
```

```
    generate_output_neurons();
HammDist = hamming_distance(pattern_counter);
    if (HammDist > CRITERIA)
       singlets1(pattern_counter);
    if (HammDist > CRITERIA)
       doublets1(pattern_counter);
    if (HammDist > CRITERIA)
       triplets1(pattern_counter);
    if (HammDist > CRITERIA)
       quadlets1(pattern_counter);
    if (HammDist > CRITERIA)
      singlets2(pattern_counter);
    if (HammDist > CRITERIA)
       doublets2(pattern_counter);
    if (HammDist > CRITERIA)
       triplets2(pattern_counter);
    if (HammDist > CRITERIA)
      quadlets2(pattern_counter);
    if (HammDist > CRITERIA)
      adapt_output_layer(pattern_counter);
    if (HammDist > CRITERIA)
      pattern_learned[pattern_counter]=0;
    else
      pattern_learned[pattern_counter]=1;
/*************************************************************************
```

```
    Apply next pattern
    Store weights in network file and exit
save_weight_layers();
/*************************************************************************
FILE: OPTICAL3.C
DESCRIPTION: Performs Madaline Rule 2 training with optical hardware.
similar to above.
*************************************************************************/
PROGRAM : TEST3.C
DESCRIPTION: Provides user with a test menu for the optical neural network.
*************************************************************************/
TEST SET function looks for the test output neuron.
*************************************************************************/
unsigned integer test_set(integer test_pattern_number)
Look at  output differences
/*************************************************************************
TEST NETWORK function test network for given Hamming Distance criteria.
This is the actual test function. The two functions following this one
are for file I/o.
*************************************************************************/
function test_network()
/*****Begin testing*********************************************************/
for (test_pattern=0;test_pattern<10;test_pattern++){
/*******Present input vector*************************************************/
    binarize_inputs(Test_lsb[test_pattern],Test_msb[test_pattern]);
```

```
Propagate values through all three layers. Look at output.
/******Check output vector **********************************************/
    Apply next pattern
/*************************************************************************
ONE TEST function test network for a single Hamming Distance criteria.
MULTIPLE TESTS function tests network for a user prompted starting and
ending consecutive Hamming Distance criterias.
/*************************************************************************************
PROGRAM: DRIVER3.C
DESCRIPTION: The following routines provide hardware I/O interface to
optical ANN and host computer. I/O is performed through a 96 bit DIGIO-96
computer interface board located at base address HEX 300 utilizing an
in-house engineered break-out board, and by a 48 channel analog to digital
conversion board at base address HEX 320. This board reads 32 optical analog
outputs corresponding to 16 optical outputs for layer 1 and 16 optical
outputs for layer 2.
*************************************************************************************/
/*************************************************************************/
function ADC_LAYER1() and also layer 2.
Refer to section on ADC board for explanation
/*************************************************************************
Setup Digital IO Interface Board in computer, board base address is Hex 300.
*************************************************************************/
function MODE_0()
/************************************ ***********************************
Writes to sixteen cascaded HUGHES 4038 32-pixel SLM waveform generator chips.
```

```
Actually generates pixel "ON" phases which get either amplified or attenuated
by programmable gain amplifiers.
*****************************************************************************/
function ENABLE_WEIGHTS()
function WRITE_INPUT_VECTOR(unsigned char lsb,unsigned char msb)
Simply output values to omputer address ports (interface)
  outportb(ILED_LO,lsb);
  outportb(ILED_HI,msb);
/*****************************************************************************
Write the output from hidden layer as input to output layer only when using
all hardware thresholding.
*****************************************************************************/
function WRITE_HIDDEN1_BINARY()
similar to above, but different  addresses.
 outportb(HLED_LO,msb);
  outportb(HLED_HI,lsb);
/*****************************************************************************
Write the output from hidden layer as input to output layer when using
ADC board.
*****************************************************************************/
function PRESENT_HIDDEN1_VECTOR()
  outportb(HLED_LO,lsb);
  outportb(HLED_HI,msb);
/*****************************************************************************
This routine writes the values of weights found in WEIGHT_LAYER1[16][16]
```

```
to the corresponding hardware programmable gain amplifier. The programmable
gain amplifiers provides a gray level driving signal to the weight mask.
**************************************************************************/
function WRITE_LAYER1_WEIGHTS()
    BOARD1[32],          /* Board 1 registers */
    BOARD2[32],          /* Board 2 registers */
    BOARD3[32],          /* Board 3 registers */
    BOARD4[32],          /* Board 4 registers */
    BOARD5[32],          /* Board 5 registers */
    BOARD6[32],          /* Board 6 registers */
    BOARD7[32],          /* Board 7 registers */
    BOARD8[32],          /* Board 8 registers */
    OUTWEIGHTS1[256],    /* Output array for writing to SLM1 */
    D0 = 1,              /* Typecasting numbers for bit shifting */
    D1 = 2,
    D2 = 4,
    D3 = 8,
    D4 = 16,
    D5 = 32,
    D6 = 64,
    D7 = 128;
/**************************************************************************
Convert the weights and biases into usable forms for the optical weights
Recall optical weights are limited in range.
**************************************************************************
```

```
Mid transmission for column 16, will be used for active thresholding.
*****************************************************************************/
      Optical_layer1[15][j] = 90;
/*****************************************************************************
       Assign corresponding weight to sim driver board
  /** Now set up ordinal serial sequence and concatenate all 8 **/
  /** board data lines to be clocked out **/
/*****************************************************************************
The following defines driver board data line in output port of  interface
 BOARD #__BIT POSITION___BIT
  BOARD1    0        1
  BOARD2    1        2
  BOARD3    2        4
  BOARD4    3        8
  BOARD5    4        16
  BOARD6    5        32
  BOARD7    6        64
  BOARD8    7        128
/*****************************************************************************/
/*****************************************************************************
This routine writes the values of weights found in WEIGHT_LAYER2 [16][16]
to the corresponding hardware programmable gain amplifier. The programmable
gain amplifiers provides a gray level driving signal to the weight mask.
*****************************************************************************/
function WRITE_LAYER2_WEIGHTS()
```

Similar to above.

/**********

PROGRAM : TRAIN.C

DESCRIPTION: Provides user with a training menu.

**********/

ONE RUN function trains network for a single Hamming Distance criteria.

**********/

function one_run()

"Enter Hamming Distance Training Criteria (between 0 and 3 inclusive)

/**********

MULTIPLE RUN function trains network for a user prompted starting and

ending consecutive Hamming Distance criteria.

**********/

function multiple_runs()

Enter starting Hamming Distance Training Criteria (between 0 and 2 inclusive)

Enter ending Hamming Distance Training Criteria (between start and 3 inclusive)

/**********

TRAIN MENU FUNCTION prompts user for training options.

**********/

function train_menu()

/**********

/**********

FILE: RANK.C

From Numerical Recipes in C.

DESCRIPTION: Ranks an array of N values in ascending order. Used in

```
Madaline Rule 2 Training.
**********************************************************************/

INDEXX function produces an array pointer which indexes the values passed
in ascending order.
**********************************************************************/

function Indexx(integer n,double arrin[],integer indx[])
Similar to quicksort
/**********************************************************************

RANK function takes the index array and converts it to a ordered rank array.
**********************************************************************/

function Rank(integer n,integer indx[],integer irank[])
```

CHAPTER 6

Training

In multilayer networks, it is straightforward to adapt the neurons in the output layer since network desired responses are equivalent to the desired response of the output neurons. Given the desired responses, the adaptation of the output layer can be realized with algorithms such as the LMS.

In layered nets, the fundamental difficulty lies in obtaining desired responses for neurons in layers other than the output layer. The backpropagation algorithm [1,5,11,12] is a method for establishing desired responses for "hidden layer" neurons.

6.1 Algorithm selection

Although backpropagation is a well known training algorithm, it is only useful when the nonlinear activation function is continuous and differentiable at all points. The hardlimiting threshold logic function is not differentiable, but is used because it is less costly to implement in hardware. Threshold activation functions can be realized with analog comparators. Very few hardlimiting function, multilayer network training algorithms have been developed [15,16,20,21,22]. A multilayer extension of the LMS, known as MR2, was chosen for ease of coding[16].

6.1 MR2 training rule

This training rule is based on the minimum disturbance principle[16]. Its main objective is to reduce the number of output errors, or the Hamming Distance (HD). The HD is simply the number of differences between the actual output vector Y_k and the target or desired vector D_k, and is expressed mathematically as

$$HD = \sum (D_k - Y_k) \qquad (6.1)$$

The algorithm is described as follows:

1. Present an input vector to the network.
2. Calculate HD
3. If HD < some criteria present next input vector, otherwise go to 4.
4. Rank order neurons. The first layer neuron whose analog response is closest to zero is given a trial adaptation in the direction to reverse its binary output. When reversal takes place, the new output is propagated through the subsequent layers and the network produces a new output.
5. Calculate new HD. If new HD < previous HD, accept change, otherwise revert back to previous weights.
6. Adaptively switch the neuron whose analog output is next closest to zero. Go to step 5 until all neurons in the first layer have been tried.
7. If more adaptation is required, choose pairs of neurons in the first layer and adapt accordingly.
8. If more adaptation is required, choose three neurons and adapt accordingly.
9. If more adaptation is required, choose four neurons and adapt accordingly.

10. Go to next layer and and adapt according to steps 4 through 9.

11. If more adaptation is required, adapt output layer. This is straight forward since desired output values are known.

12. Present next input pattern and go to step 1.

The source code for this algorithm is found in Appendix 4.

6.3 Training set

We selected two letters from the alphabet, I and F, and represented them with 12 patterns each. The patterns have one random pixel error, and also include translation for the letter I. These patterns are shown in Figure 6.1.

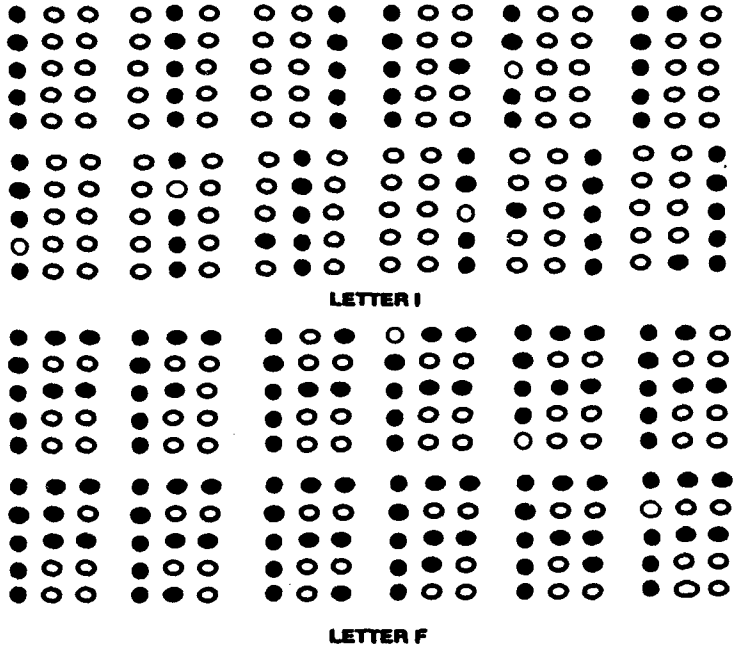


Figure 6.1. Training Set

6.4 Test set

The network was tested with 10 patterns not previously presented. Six representations of the letter I and 4 representations of the letter F (Figure 6.2) were presented to the network after the network was trained with the training set.

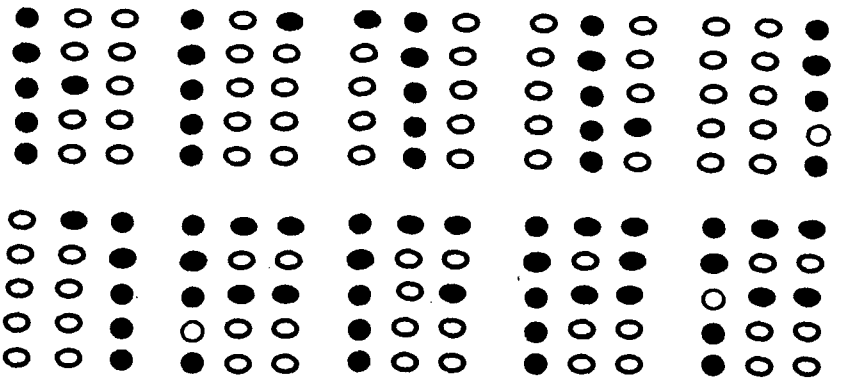


Figure 6.2. Test Set

6.5 Results

The network was trained in two modes(Figure 6.3) . Mode one is a computer simulation of the optical layers and incorporates input dependent thresholding. The top line in the plot diagram shows percent recognition vs hamming distance for the computer simulation. Mode 2 reads the neuron analog values through an Analog to Digital converter for rank ordering during training. For testing the network , the photodiode signals are input to hard limiting comparators which provide a binary output for the input pattern presented to the optical hardware. The lower performance of the

optoelectronic system can be accounted for by optical misalignment of the detectors and by a number of nonfunctioning pixels in each of the SLM's

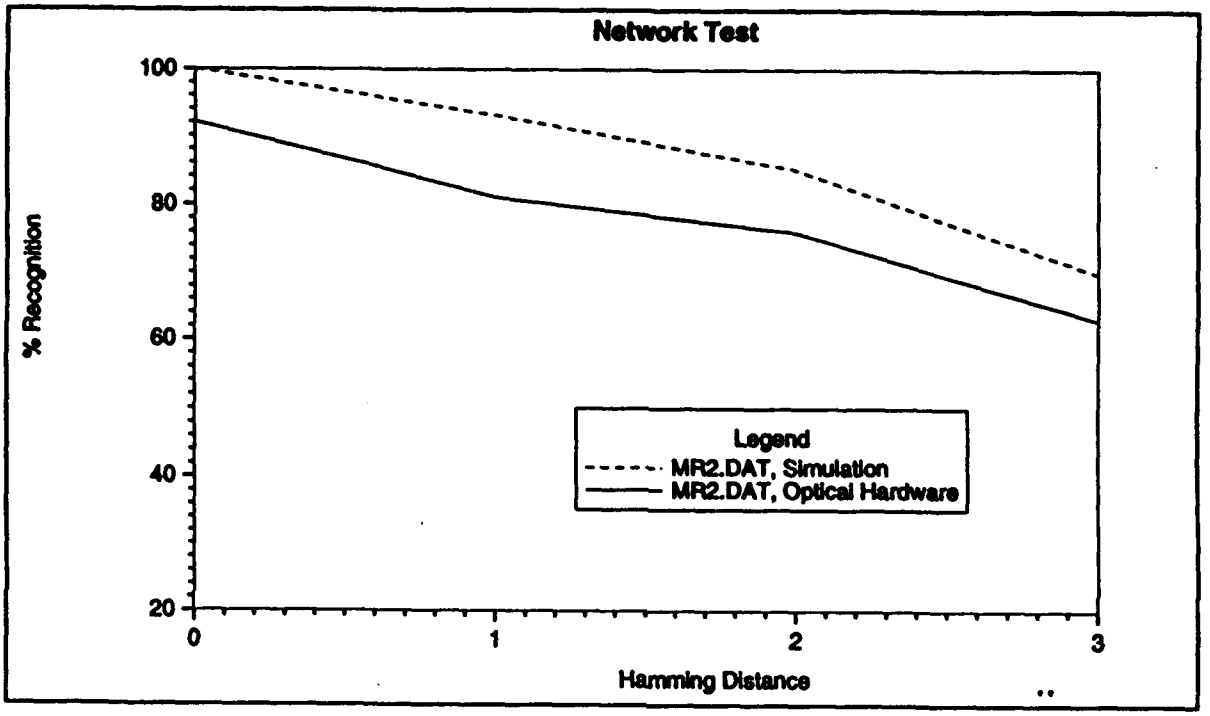


Figure 6.3. Results

CHAPTER 7

Conclusions

It has been shown that hybrid optical neural networks are very promising for pattern classification. The technology used in the implementation of the optoelectronic neural network can be incorporated at the integrated circuit level. This means that fully interconnected optoelectronic networks of many inputs can be designed and constructed. Optics offer a solution for the communication bottleneck encountered by an all electronic device, while electronics offer high gain and ease of interface to current platforms. The MR2 rule shows that electro optical ANN networks utilizing hard limiting or threshold logic activation functions are trainable . This offers a significant alternative to using backpropagation.

There is much work to be done in the area of optical neural networks. Current technology can be used to design and demonstrate an integrated optical chip device for neural network computation. Semiconductor lasers and detectors are easily implementable at integrated circuit scales.The crossbar configuration is readily implemented at the chip level in the form of waveguides [38, 43, 45].High density SLM's have been demonstrated at the integrated circuit level [30,31,32,33,58] and can be clocked at MHz rates [60]. Everything is in place to take the findings of this thesis to a practical level and develop next generation devices. A strong recommendation is made for this work to be carried further and an optoelectronic neural chip be manufactured.

Appendix 1

LMS algorithm

Consider the case of a single neuron. For adaptation the LMS rule is:

$$W_{k+1} = W_k + \frac{\alpha}{|X_k|^2} \varepsilon_k X_k \qquad (1)$$

where W_{k+1} is the next value of the weight vector, W_k is the present value of the weight vector, X_k is the present value of the input vector, and ε_k is the difference between the desired and actual output before adaptation. With binary +1,-1 values for the input vector elements, $|X_k|^2$ is equal to the number of weights.

For each adaptation cycle, the above recursion is applied. This reduces the error by the fraction α. Suppose that for the k^{th} iteration the error is:

$$\varepsilon_k = d_k - X_k^T W_k \qquad (2)$$

The error is reduced by changing the weights, that is

$$\Delta\varepsilon_k = \Delta(d_k - X_k^T W_k) = -X_k^T \Delta W_k \qquad (3)$$

Using equation 1, the weight change is

$$\Delta W_k = W_{k+1} - W_k = \frac{\alpha}{|X_k|^2} \varepsilon_k X_k \qquad (4)$$

Now combine equations 3 and 4 to obtain:

$$\Delta\varepsilon_k = -X_k^T \frac{\alpha\, \varepsilon_k X_k}{|X_k|^2}$$

$$= -X_k^T X_k \frac{\alpha\, \varepsilon_k}{|X_k|^2}$$

$$= -\alpha\varepsilon_k \qquad (5)$$

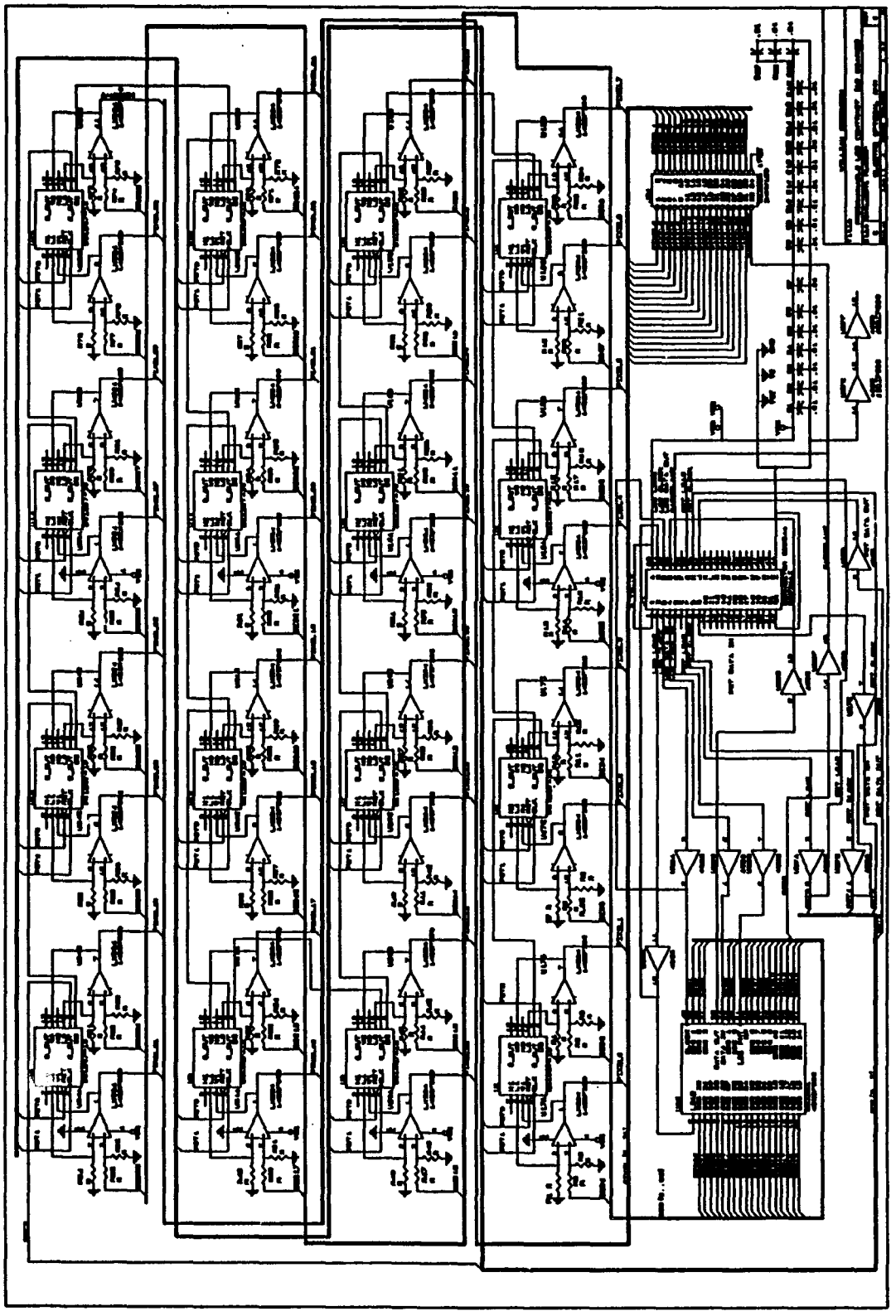
The error is reduced by factor of α as the weights are changed while holding the input pattern fixed. A new pattern then starts the next adaptation cycle. Since factor α reduces the error, it must be chosen to control stability and speed of convergence. This value is usually between 1.0 and 0.1.

Appendix 2

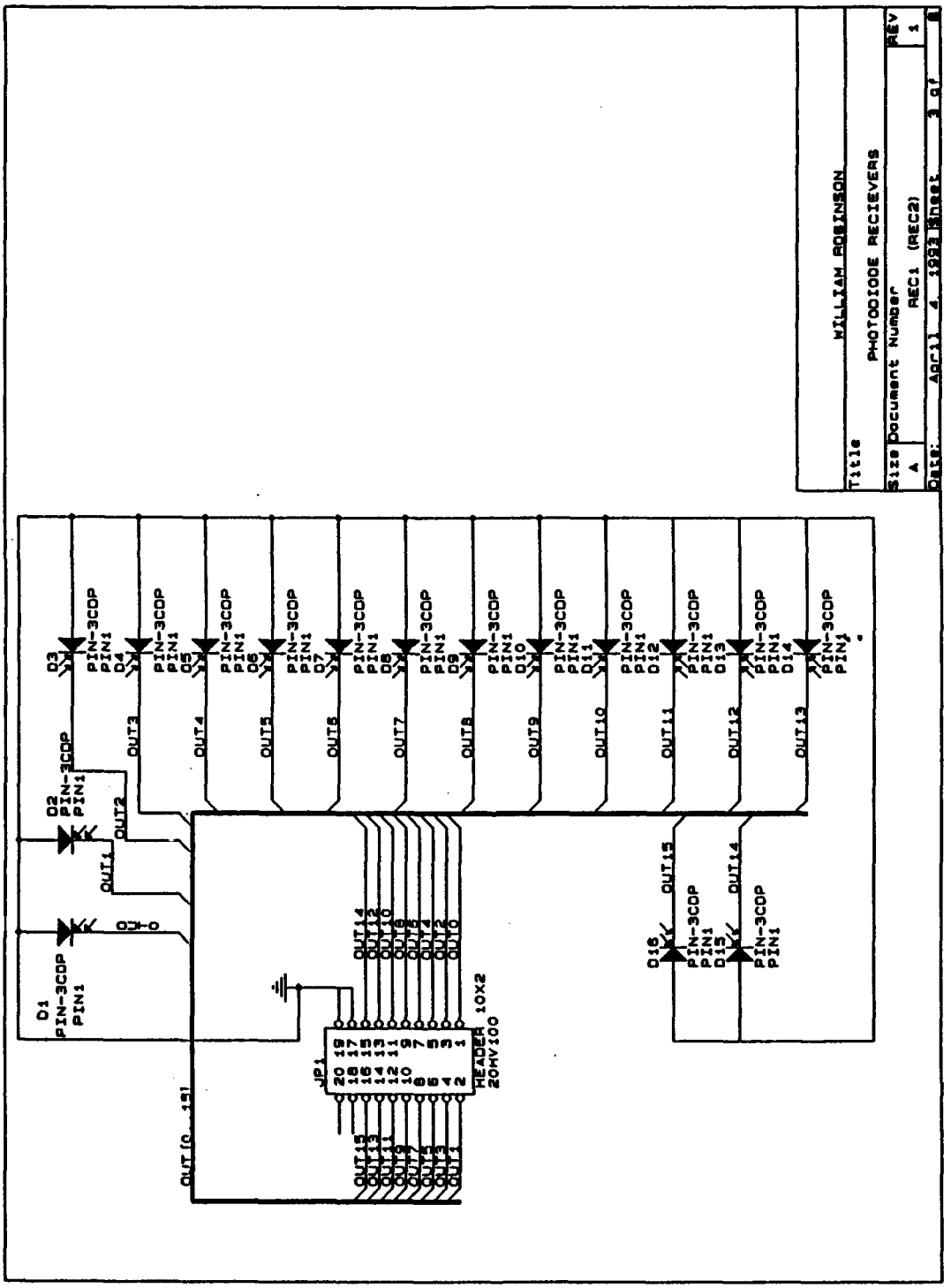
Schematics

All detailed schematics were drawn with **OrCAD SDT III** schematic layout tools on a PC. Printed circuit boards were routed with **OrCAD PCB** board design tools. All printed circuit boards, for the exception of the SLM driver boards, were layed out and manufactured at the Engineering Services Branch of Armstrong Laboratory's Technical Services Division. Eight major design schematics were created and are shown in subsequent pages.

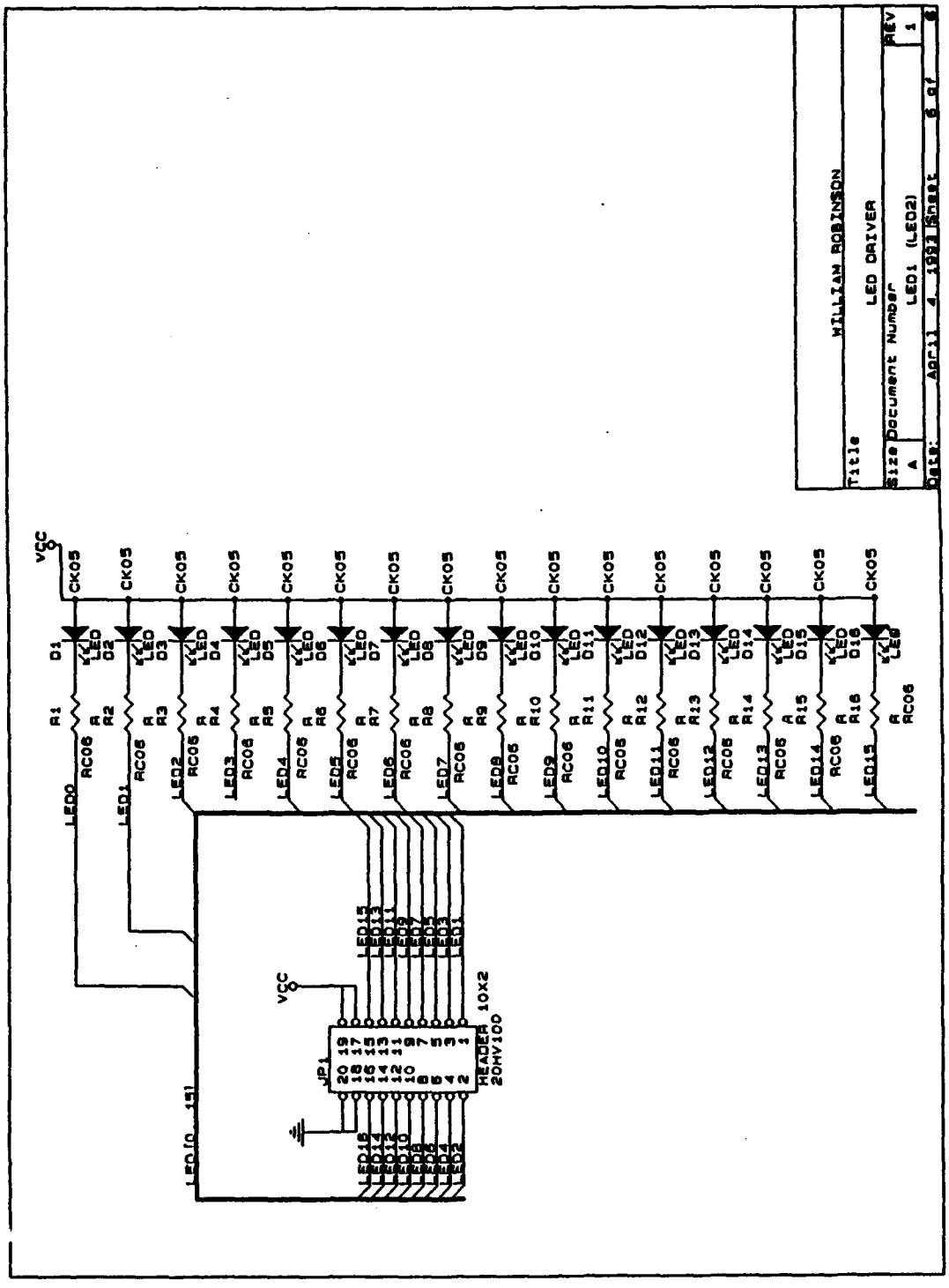
LCD pixel driver: DRIVER.SCH



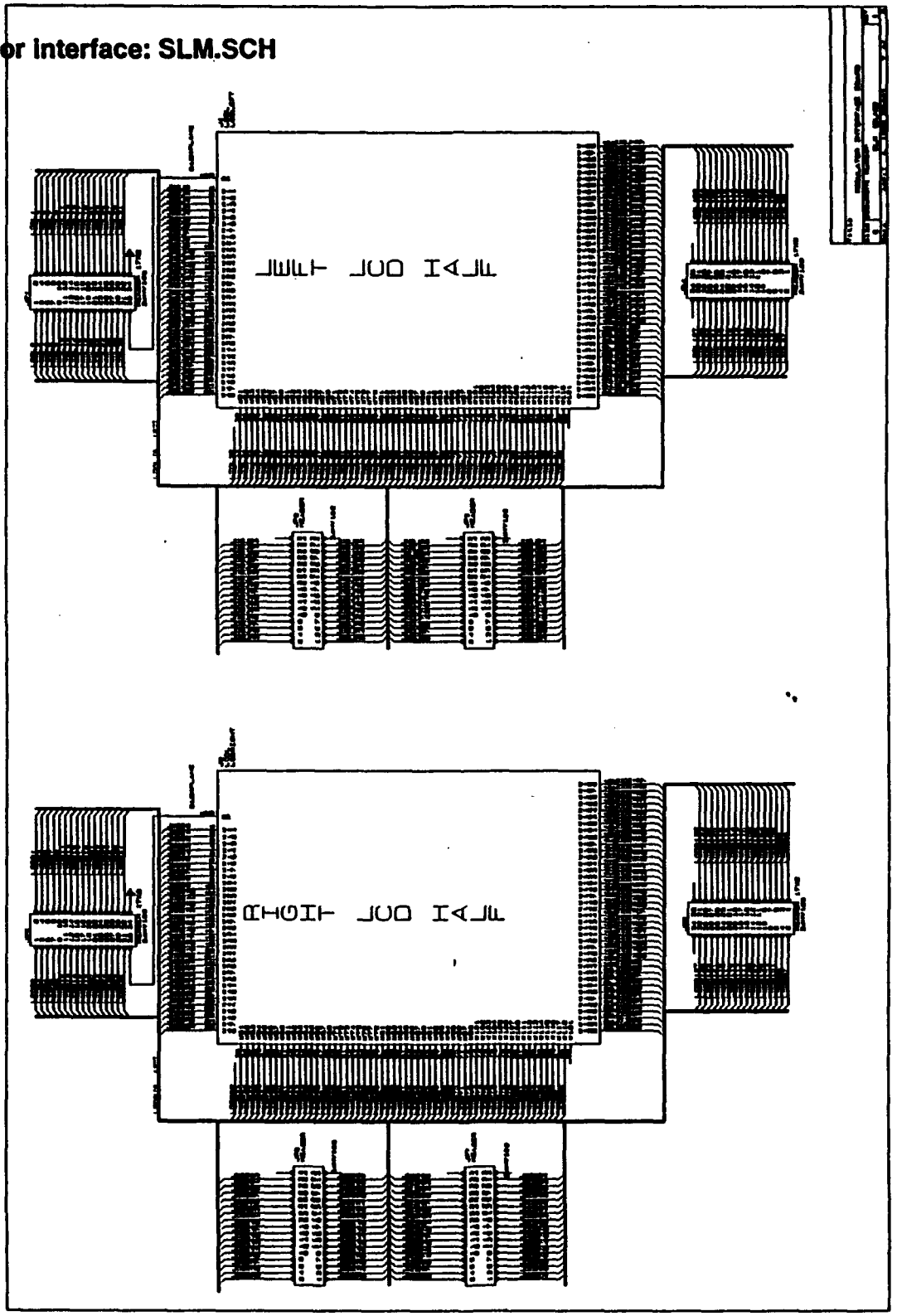
Photodiode receiver stage: REC1.SCH



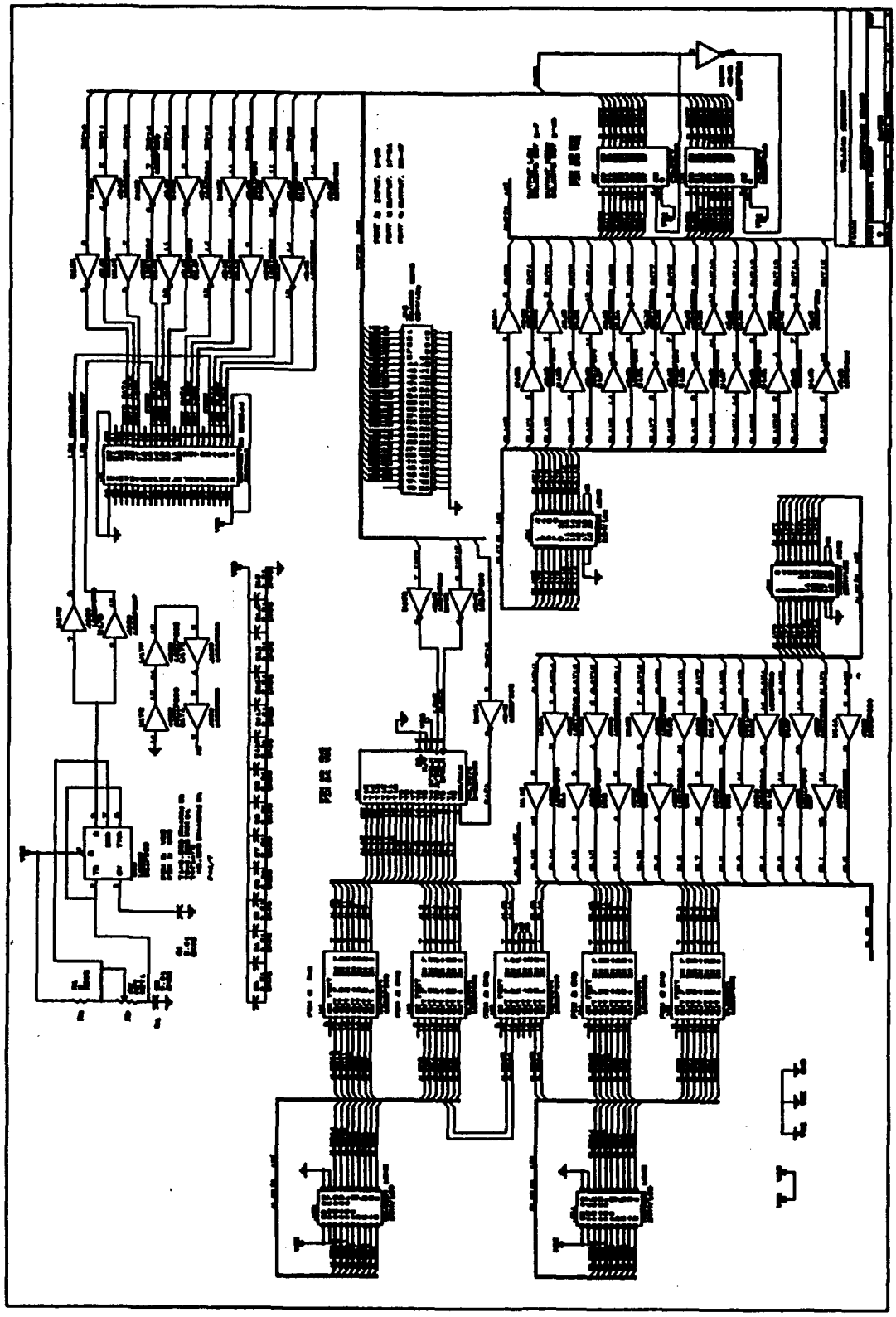
LED linear array: LED.SCH



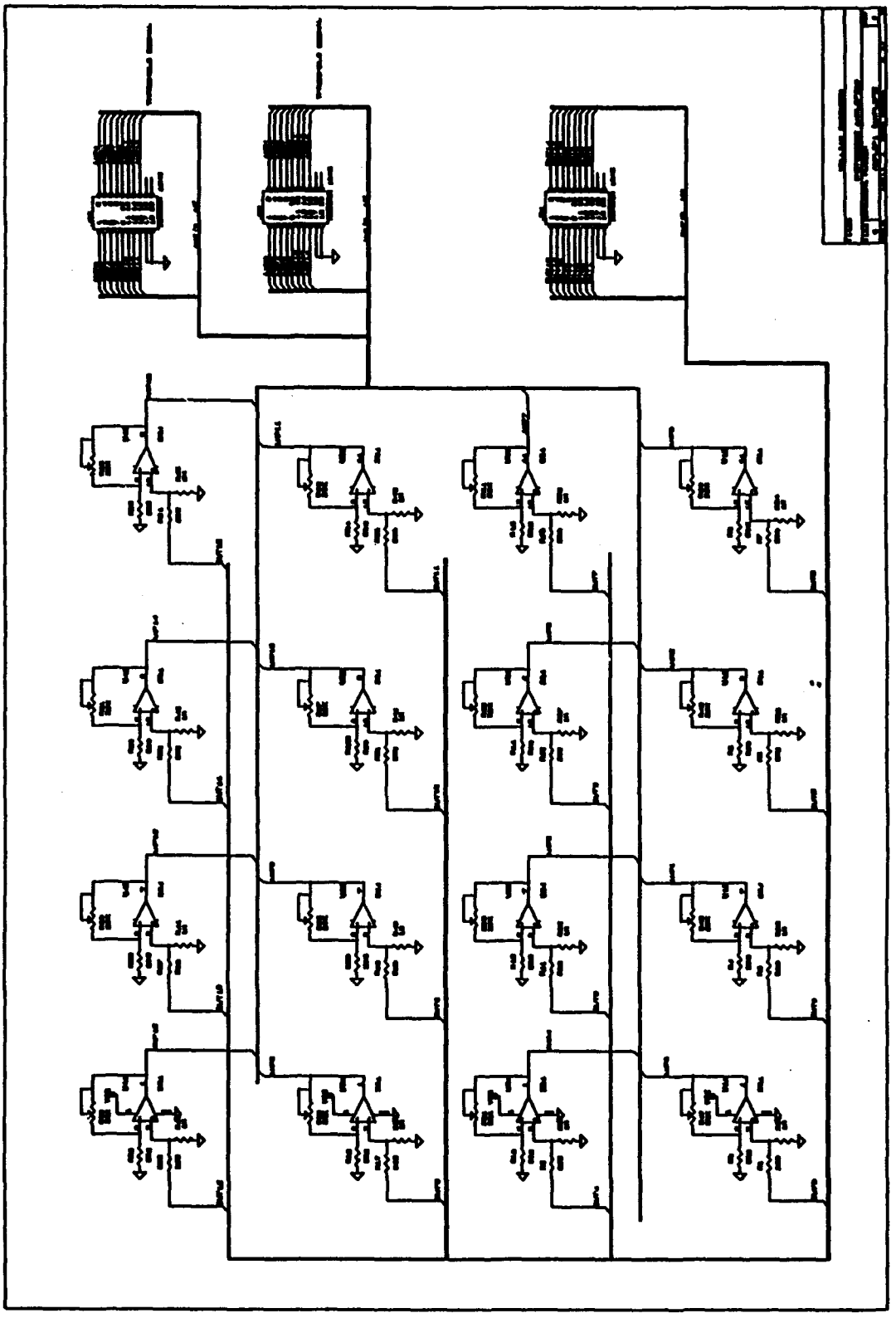
Modulator Interface: SLM.SCH



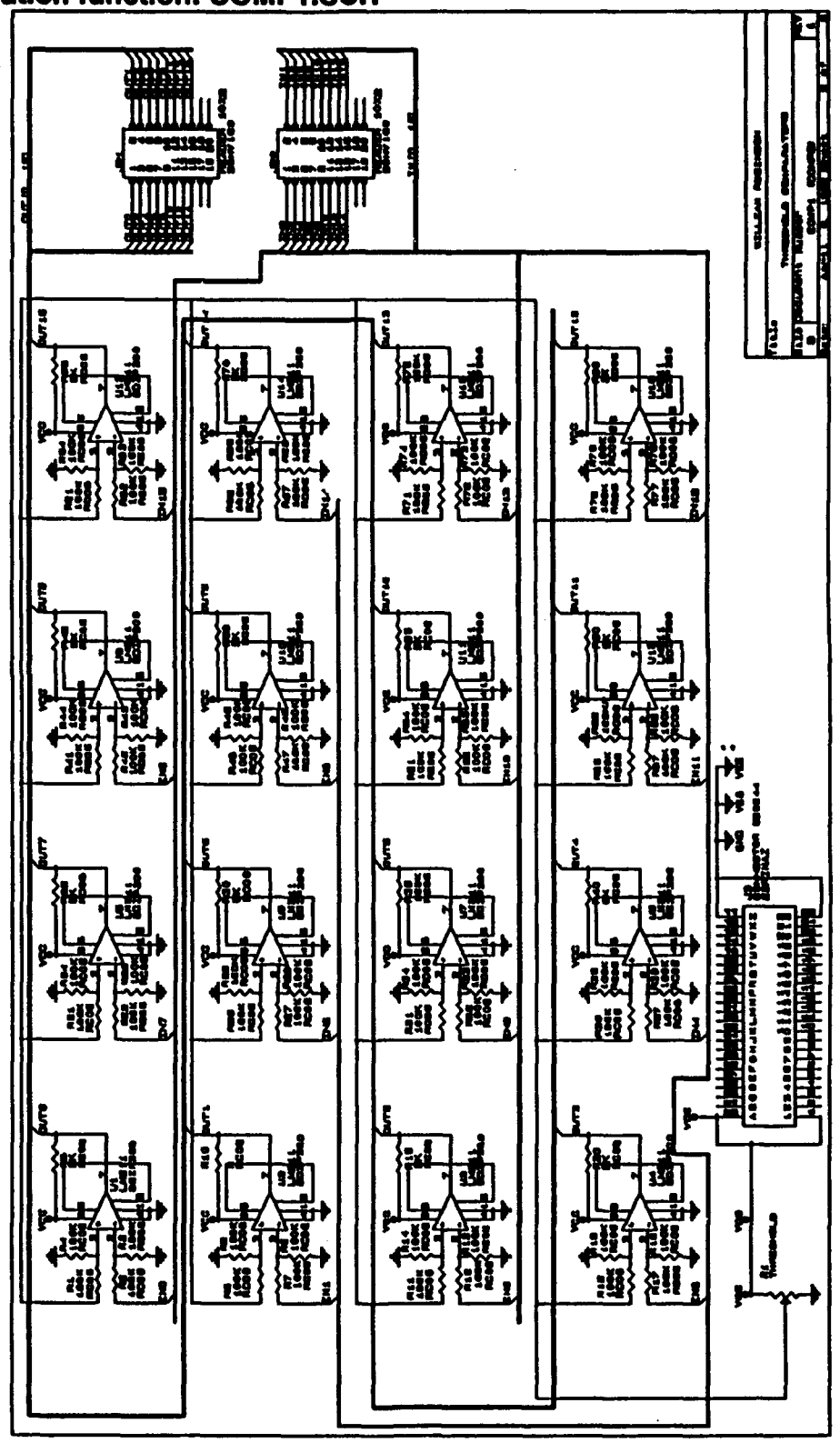
Digital interface and SLM clock: INTER.SCH



Photodiode amplifier: AMPLIF.SCH



Threshold activation function: COMP1.SCH



Appendix 3

SPICE listings

The SPICE simulations were performed with Microsim's PSPICE on a MacIntosh IIFE.

THRESHOLD COMPARATOR

```
*WILLIAM ROBINSON
*ANN PROJECT
*ANALYSIS REQUESTS
.DC LIN VIN 0.5 1.5 0.050
*CIRCUIT COMPONENTS
R1 1 2 100K
R2 2 0 100K
R3 3 4 100K
R4 4 0 100K
R5 100 5 5K
*SOURCES
VDD 100 0 DC 5
VIN 3 0 DC 0.5
VREF 1 0 DC 1
*--------------------------------------------------------
* connections:   non-inverting input
```

```
*                  | inverting input
*                  | | positive power supply
*                  | | | negative power supply
*                  | | | | open collector output
*                  | | | | | output ground
*                  | | | | | |
.subckt LM311      1 2 3 4 5 6
*
f1     9  3 v1 1    iee    3  7 dc 100.0E-6    vi1  21  1 dc .45
vi2  22  2 dc .45    q1     9 21  7 qin    q2     8 22  7 qin
q3     9  8  4 qmo    q4     8  8  4 qmi .model qin
PNP(Is=800.0E-18 Bf=500) .model qmi NPN(Is=800.0E-18
Bf=1002) .model qmo NPN(Is=800.0E-18 Bf=1000 Cjc=1E-15
Tr=118.8E-9)    e1    10  6  9  4  1    v1    10 11 dc 0    q5
5 11  6 qoc .model qoc NPN(Is=800.0E-18 Bf=34.49E3 Cjc=1E-15
Tf=364.6E-12 Tr=79.34E-9)    dp     4  3 dx    rp     3  4
6.818E3 .model dx  D(Is=800.0E-18 Rs=1)
*
.ends
*-------------------------------------------------------------
*COMPARATOR
X1 4 2 100 0 5 0 LM311
```

```
*OUTPUT REQUESTS
.PRINT DC V(5)
.PROBE
.END
```

Print data

VIN	V(5)
5.000E-01	6.405E-02
5.500E-01	6.405E-02
6.000E-01	6.407E-02
6.500E-01	6.412E-02
7.000E-01	6.423E-02
7.500E-01	6.454E-02
8.000E-01	6.534E-02
8.500E-01	6.737E-02
9.000E-01	7.244E-02
9.500E-01	8.563E-02
1.000E+00	2.454E-01
1.050E+00	5.000E+00
1.100E+00	5.000E+00
1.150E+00	5.000E+00

1.200E+00	5.000E+00
1.250E+00	5.000E+00
1.300E+00	5.000E+00
1.350E+00	5.000E+00
1.400E+00	5.000E+00
1.450E+00	5.000E+00
1.500E+00	5.000E+00

VREF	V(5)
5.000E-01	5.000E+00
5.500E-01	5.000E+00
6.000E-01	5.000E+00
6.500E-01	5.000E+00
7.000E-01	5.000E+00
7.500E-01	5.000E+00
8.000E-01	5.000E+00
8.500E-01	5.000E+00
9.000E-01	5.000E+00
9.500E-01	5.000E+00
1.000E+00	2.454E-01
1.050E+00	8.563E-02
1.100E+00	7.244E-02

1.150E+00	6.737E-02
1.200E+00	6.534E-02
1.250E+00	6.454E-02
1.300E+00	6.423E-02
1.350E+00	6.412E-02
1.400E+00	6.407E-02
1.450E+00	6.405E-02
1.500E+00	6.405E-02

```
 LCD DRIVER
*WILLIAM ROBINSON
*ANN PROJECT
*ANALYSIS REQUESTS
.DC RES RPOT(R) 40 10K 40
*CIRCUIT COMPONENTS
R1 100 2 29.3K
R2 2 0 100
R3 3 0 39
R4 3 4 RPOT 1
.MODEL RPOT RES(R=1)
*SOURCES
VDD 100 0 DC 5
*LM 324 MODEL
*-------------------------------------------------------------*
connections:   non-inverting input
*                | inverting input
*                | | positive power supply
*                | | | negative power supply
*                | | | | output
*                | | | | | .subckt LM324    1 2 3 4 5
```

```
*
c1   11 12 2.887E-12   c2    6  7 30.00E-12   dc    5 53 dx
de   54  5 dx   dlp  90 91 dx   dln  92 90 dx   dp    4  3
dx   egnd 99  0 poly(2) (3,0) (4,0) 0 .5 .5   fb    7 99
poly(5) vb vc ve vlp vln 0 21.22E6 -20E6 20E6 20E6 -20E6
ga    6  0 11 12 188.5E-6   gcm   0  6 10 99 59.61E-9   iee
3 10 dc 15.09E-6   hlim 90  0 vlim 1K   q1   11  2 13 qx
q2   12  1 14 qx   r2    6  9 100.0E3   rc1   4 11 5.305E3
rc2   4 12 5.305E3   re1  13 10 1.845E3   re2  14 10 1.845E3
ree  10 99 13.25E6   ro1   8  5 50   ro2   7 99 25   rp    3
4 9.082E3   vb    9  0 dc 0   vc    3 53 dc 1.500   ve   54
4 dc 0.65   vlim  7  8 dc 0   vlp  91  0 dc 40   vln   0 92
dc 40 .model dx D(Is=800.0E-18 Rs=1) .model qx
PNP(Is=800.0E-18 Bf=166.7)
.ends
*-----------------------------------------------------------
*OP AMP
X1 2 3 100 0 4 LM324
*CIRCUIT INPUT
*VPULSE 1 0 PULSE(0 5 .5US .1US .1US 20US 30US)
```

```
*OUTPUT REQUESTS
.PRINT DC V(4)
.PROBE
.END
```

Print data

R	V(4)
4.000E+01	7.401E-02
8.000E+01	8.760E-02
1.200E+02	9.988E-02
1.600E+02	1.119E-01
2.000E+02	1.243E-01
2.400E+02	1.375E-01
2.800E+02	1.517E-01
3.200E+02	1.669E-01
3.600E+02	1.831E-01
4.000E+02	2.000E-01
4.400E+02	2.173E-01

4.800E+02	2.350E-01
5.200E+02	2.528E-01
5.600E+02	2.708E-01
6.000E+02	2.888E-01
6.400E+02	3.068E-01
6.800E+02	3.249E-01
7.200E+02	3.429E-01
7.600E+02	3.610E-01
8.000E+02	3.791E-01
8.400E+02	3.971E-01
8.800E+02	4.152E-01
9.200E+02	4.332E-01
9.600E+02	4.513E-01
1.000E+03	4.694E-01
1.040E+03	4.874E-01
1.080E+03	5.055E-01
1.120E+03	5.236E-01
1.160E+03	5.416E-01
1.200E+03	5.597E-01
1.240E+03	5.778E-01
1.280E+03	5.958E-01
1.320E+03	6.139E-01

1.360E+03	6.320E-01
1.400E+03	6.500E-01
1.440E+03	6.681E-01
1.480E+03	6.861E-01
1.520E+03	7.042E-01
1.560E+03	7.223E-01
1.600E+03	7.403E-01
1.640E+03	7.584E-01
1.680E+03	7.764E-01
1.720E+03	7.945E-01
1.800E+03	8.306E-01
1.840E+03	8.487E-01
1.880E+03	8.667E-01
1.920E+03	8.848E-01
1.960E+03	9.028E-01
2.000E+03	9.209E-01
2.040E+03	9.390E-01
2.080E+03	9.570E-01
2.120E+03	9.751E-01
2.160E+03	9.931E-01
2.200E+03	1.011E+00

2.240E+03	1.029E+00
2.280E+03	1.047E+00
2.320E+03	1.065E+00
2.360E+03	1.083E+00
2.400E+03	1.101E+00
2.440E+03	1.120E+00
2.480E+03	1.138E+00
2.520E+03	1.156E+00
2.560E+03	1.174E+00
2.600E+03	1.192E+00
2.640E+03	1.210E+00
2.680E+03	1.228E+00
2.720E+03	1.246E+00
2.760E+03	1.264E+00
2.800E+03	1.282E+00
2.840E+03	1.300E+00
2.880E+03	1.318E+00
2.920E+03	1.336E+00
2.960E+03	1.354E+00
3.000E+03	1.372E+00
3.040E+03	1.390E+00

3.080E+03	1.408E+00
3.120E+03	1.426E+00
3.160E+03	1.444E+00
3.200E+03	1.462E+00
3.240E+03	1.480E+00
3.280E+03	1.499E+00
3.320E+03	1.517E+00
3.360E+03	1.535E+00
3.400E+03	1.553E+00
3.440E+03	1.571E+00
3.480E+03	1.589E+00
3.520E+03	1.607E+00
3.560E+03	1.625E+00
3.600E+03	1.643E+00
3.640E+03	1.661E+00
3.680E+03	1.679E+00
3.720E+03	1.697E+00
3.760E+03	1.715E+00
3.800E+03	1.733E+00
3.840E+03	1.751E+00
3.880E+03	1.769E+00

3.920E+03	1.787E+00
3.960E+03	1.805E+00
4.000E+03	1.823E+00
4.040E+03	1.841E+00
4.080E+03	1.859E+00
4.120E+03	1.877E+00
4.160E+03	1.895E+00
4.200E+03	1.913E+00
4.240E+03	1.931E+00
4.280E+03	1.950E+00
4.320E+03	1.968E+00
4.360E+03	1.986E+00
4.400E+03	2.004E+00
4.440E+03	2.022E+00
4.480E+03	2.040E+00
4.520E+03	2.058E+00
4.560E+03	2.076E+00
4.600E+03	2.094E+00
4.640E+03	2.112E+00
4.680E+03	2.130E+00
4.720E+03	2.148E+00

4.760E+03	2.166E+00
4.800E+03	2.184E+00
4.840E+03	2.202E+00
4.880E+03	2.220E+00
4.920E+03	2.238E+00
4.960E+03	2.256E+00
5.000E+03	2.274E+00
5.040E+03	2.292E+00
5.080E+03	2.310E+00
5.120E+03	2.328E+00
5.160E+03	2.346E+00
5.200E+03	2.364E+00
5.240E+03	2.382E+00
5.280E+03	2.400E+00
5.320E+03	2.418E+00
5.360E+03	2.436E+00
5.400E+03	2.454E+00
5.440E+03	2.472E+00
5.480E+03	2.490E+00
5.520E+03	2.508E+00
5.560E+03	2.526E+00

5.600E+03	2.544E+00
5.640E+03	2.563E+00
5.680E+03	2.581E+00
5.720E+03	2.599E+00
5.760E+03	2.617E+00
5.800E+03	2.635E+00
5.840E+03	2.653E+00
5.880E+03	2.671E+00
5.920E+03	2.689E+00
5.960E+03	2.707E+00
6.000E+03	2.725E+00
6.040E+03	2.743E+00
6.080E+03	2.761E+00
6.120E+03	2.779E+00
6.160E+03	2.797E+00
6.200E+03	2.815E+00
6.240E+03	2.833E+00
6.280E+03	2.851E+00
6.320E+03	2.869E+00
6.360E+03	2.887E+00
6.400E+03	2.905E+00
6.440E+03	2.923E+00

6.480E+03	2.941E+00
6.520E+03	2.959E+00
6.560E+03	2.977E+00
6.600E+03	2.995E+00
6.640E+03	3.013E+00
6.680E+03	3.031E+00
6.720E+03	3.049E+00
6.760E+03	3.067E+00
6.800E+03	3.085E+00
6.840E+03	3.103E+00
6.880E+03	3.121E+00
6.920E+03	3.139E+00
6.960E+03	3.157E+00
7.000E+03	3.175E+00
7.040E+03	3.193E+00
7.080E+03	3.211E+00
7.120E+03	3.229E+00
7.160E+03	3.247E+00
7.200E+03	3.265E+00
7.240E+03	3.283E+00

7.280E+03	3.301E+00
7.320E+03	3.319E+00
7.360E+03	3.337E+00
7.400E+03	3.355E+00
7.440E+03	3.373E+00
7.480E+03	3.391E+00
7.520E+03	3.409E+00
7.560E+03	3.427E+00
7.600E+03	3.445E+00
7.640E+03	3.463E+00
7.680E+03	3.481E+00
7.720E+03	3.499E+00
7.760E+03	3.517E+00
7.800E+03	3.535E+00
7.840E+03	3.553E+00
7.880E+03	3.571E+00
7.920E+03	3.589E+00
7.960E+03	3.607E+00
8.000E+03	3.625E+00
8.040E+03	3.643E+00
8.080E+03	3.661E+00
8.120E+03	3.679E+00

8.160E+03	3.697E+00
8.200E+03	3.715E+00
8.240E+03	3.733E+00
8.280E+03	3.751E+00
8.320E+03	3.769E+00
8.360E+03	3.787E+00
8.400E+03	3.805E+00
8.440E+03	3.823E+00
8.480E+03	3.841E+00
8.520E+03	3.858E+00
8.560E+03	3.875E+00
8.600E+03	3.892E+00
8.640E+03	3.907E+00
8.680E+03	3.921E+00
8.720E+03	3.933E+00
8.760E+03	3.943E+00
8.800E+03	3.951E+00
8.840E+03	3.958E+00
8.880E+03	3.964E+00
8.920E+03	3.970E+00

8.960E+03	3.974E+00
9.000E+03	3.978E+00
9.040E+03	3.981E+00
9.080E+03	3.985E+00
9.120E+03	3.987E+00
9.160E+03	3.990E+00
9.200E+03	3.992E+00
9.240E+03	3.994E+00
9.280E+03	3.996E+00
9.320E+03	3.998E+00
9.360E+03	4.000E+00
9.400E+03	4.002E+00
9.440E+03	4.003E+00
9.480E+03	4.005E+00
9.520E+03	4.006E+00
9.560E+03	4.007E+00
9.600E+03	4.008E+00
9.640E+03	4.010E+00
9.680E+03	4.011E+00
9.720E+03	4.012E+00
9.760E+03	4.013E+00
9.800E+03	4.014E+00

9.840E+03	4.015E+00
9.880E+03	4.015E+00
9.920E+03	4.016E+00
9.960E+03	4.017E+00
1.000E+04	4.018E+00

```
PHOTOVOLTAIC RECEIVER
*WILLIAM ROBINSON
*ANN PROJECT
*ANALYSIS REQUESTS
.DC LIN VIN 0.050 0.350 0.025
*CIRCUIT COMPONENTS
R1 1 2 909
R2 2 0 1MEG
R3 3 0 909
R4 3 4 5K
*SOURCES
VDD 100 0 DC 5
VIN 1 0 DC 0.25
*LM 324 MODEL
*-------------------------------------------------------*
connections:   non-inverting input
*                | inverting input
*                | | positive power supply
*                | | | negative power supply
*                | | | | output
*                | | | | |
```

```
.subckt LM324    1 2 3 4 5
*
c1   11 12 2.887E-12
  c2    6  7 30.00E-12
  dc    5 53 dx
  de   54  5 dx
  dlp  90 91 dx
  dln  92 90 dx
  dp    4  3 dx
  egnd 99  0 poly(2) (3,0) (4,0) 0 .5 .5
  fb    7 99 poly(5) vb vc ve vlp vln 0 21.22E6 -20E6 20E6
  20E6 -20E6
  ga    6  0 11 12 188.5E-6
  gcm   0  6 10 99 59.61E-9
  iee   3 10 dc 15.09E-6
  hlim 90  0 vlim 1K
  q1   11  2 13 qx
  q2   12  1 14 qx
  r2    6  9 100.0E3
  rc1   4 11 5.305E3
  rc2   4 12 5.305E3
  re1  13 10 1.845E3
```

```
  re2   14 10 1.845E3
  ree   10 99 13.25E6
  ro1    8  5 50
  ro2    7 99 25
  rp     3  4 9.082E3
  vb     9  0 dc 0
  vc     3 53 dc 1.500
  ve    54  4 dc 0.65
  vlim   7  8 dc 0
  vlp   91  0 dc 40
  vln    0 92 dc 40
.model dx D(Is=800.0E-18 Rs=1)
.model qx PNP(Is=800.0E-18 Bf=166.7)
.ends
*-----------------------------------------------------------
*OP AMP
X1 2 3 100 0 4 LM324
*OUTPUT REQUESTS
.PRINT DC V(4)
.PROBE
.END
```

Print data

VIN	V(4)
5.000E-02	3.287E-01
7.500E-02	4.910E-01
1.000E-01	6.533E-01
1.250E-01	8.156E-01
1.500E-01	9.779E-01
1.750E-01	1.140E+00
2.000E-01	1.303E+00
2.250E-01	1.465E+00
2.500E-01	1.627E+00
2.750E-01	1.789E+00
3.000E-01	1.952E+00
3.250E-01	2.114E+00
3.500E-01	2.276E+00

```
DFG2 CIRCUIT
*WILLIAM ROBINSON
*ANN PROJECT
*ANALYSIS REQUEST
.DC VIN -15 15 .5
*RESISTORS
R1 1 5 10K
R2 2 4 10K
R3 4 5 15K
R4 5 8 15K
R5 1 2 10K
R6 20 2 10K
R7 6 8 10K
R8 1 6 10K
R9 21 6 10K
RF 5 10 10K
R10 1 11 10K
R11 11 22 10K
R12 11 13 10K
R13 1 14 10K
R14 14 23 10K
```

```
R15 14 16 10K
R16 13 5 25K
R17 16 5 50K
*DIODES
D1 2 3 D1N4154
D2 3 4 D1N4154
D3 7 6 D1N4154
D4 8 7 D1N4154
D5 11 12 D1N4154
D6 12 13 D1N4154
D7 15 14 D1N4154
D8 16 15 D1N4154
*OP AMPS
X1 0 2 30 31 3 LM324
X2 0 6 30 31 7 LM324
X3 0 5 30 31 10 LM32
X4 0 11 30 31 12 LM324
X5 0 14 30 31 15 LM324
*SOURCES
V1 20 0 DC 5
V2 0 21 DC 5
V3 22 0 DC 10
```

```
V4 0 23 DC 10
VCC 30 0 DC 15
VDD 0 31 DC 15
VIN 1 0 DC 1
.model D1N4154  D(Is=0.1p Rs=4 CJO=2p Tt=3n Bv=60 Ibv=0.1p)
*OP AMP SUBCIRCUIT
* connections:   non-inverting input
*                | inverting input
*                | | positive power supply
*                | | | negative power supply
*                | | | | output
*                | | | | |
.subckt OP-07    1 2 3 4 5
*
c1   11 12 8.661E-12   c2    6  7 30.00E-12   dc    5 53 dx
de   54  5 dx   dlp  90 91 dx   dln  92 90 dx   dp    4  3
dx   egnd 99  0 poly(2) (3,0) (4,0) 0 .5 .5   fb    7 99
poly(5) vb vc ve vlp vln 0 221.0E6 -200E6 200E6 200E6 -200E6
ga    6  0 11 12 113.1E-6   gcm   0  6 10 99 56.69E-12   iee
10  4 dc 6.002E-6   hlim 90  0 vlim 1K   q1   11  2 13 qx
q2   12  1 14 qx   r2    6  9 100.0E3   rc1   3 11 8.841E3
```

```
rc2    3 12 8.841E3    re1   13 10 219.4    re2   14 10 219.4
ree   10 99 33.32E6    ro1    8  5 40    ro2    7 99 20    rp     3
4 12.03E3    vb     9   0 dc 0    vc     3 53 dc 1    ve    54   4
dc 1    vlim   7   8 dc 0    vlp   91   0 dc 30    vln    0 92 dc 30
.model dx D(Is=800.0E-18 Rs=1) .model qx NPN(Is=800.0E-18
Bf=3.000E3)
.ends
*-------------------------------------------------------------
*-------------------------------------------------------------
* connections:   non-inverting input
*                | inverting input
*                | | positive power supply
*                | | | negative power supply
*                | | | | output
*                | | | | |
.subckt LM324     1 2 3 4 5
*
c1    11 12 2.887E-12    c2     6   7 30.00E-12    dc     5 53 dx
de    54   5 dx    dlp   90 91 dx    dln   92 90 dx    dp     4   3
dx    egnd 99   0 poly(2) (3,0) (4,0) 0 .5 .5    fb     7 99
poly(5) vb vc ve vlp vln 0 21.22E6 -20E6 20E6 20E6 -20E6
ga     6   0 11 12 188.5E-6    gcm    0   6 10 99 59.61E-9    iee
```

```
3 10 dc 15.09E-6   hlim 90  0 vlim 1K   q1   11  2 13 qx
q2   12  1 14 qx   r2    6  9 100.0E3   rc1   4 11 5.305E3
rc2   4 12 5.305E3   re1  13 10 1.845E3   re2  14 10 1.845E3
ree  10 99 13.25E6   ro1   8  5 50   ro2   7 99 25   rp    3
4 9.082E3   vb    9  0 dc 0   vc    3 53 dc 1.500   ve   54
4 dc 0.65   vlim  7  8 dc 0   vlp  91  0 dc 40   vln   0 92
dc 40 .model dx D(Is=800.0E-18 Rs=1) .model qx
PNP(Is=800.0E-18 Bf=166.7)
.ends
*---------------------------------------------------------
*OUTPUT
PRINT DC V(10)
.PROBE
.END
```

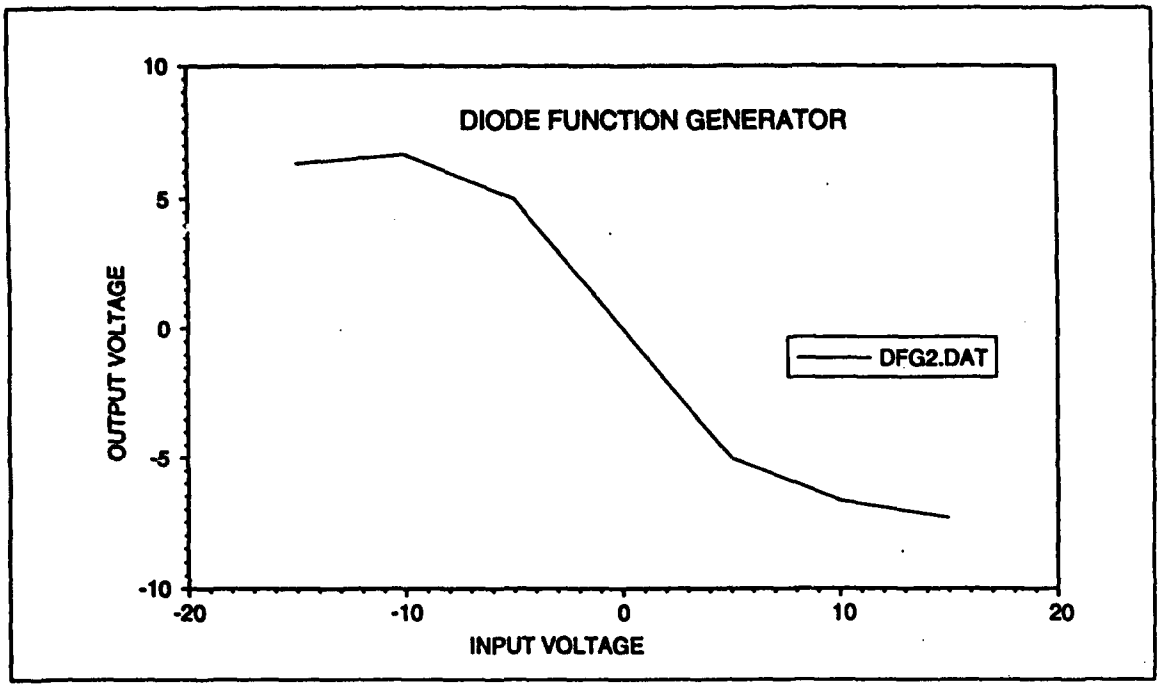
Print data

VIN	V(10)
-1.500E+01	6.333E+00
-1.450E+01	6.367E+00
-1.400E+01	6.400E+00
-1.350E+01	6.433E+00
-1.300E+01	6.467E+00
-1.250E+01	6.500E+00

-1.200E+01	6.533E+00
-1.150E+01	6.567E+00
-1.100E+01	6.600E+00
-1.050E+01	6.633E+00
-1.000E+01	6.666E+00
-9.500E+00	6.500E+00
-9.000E+00	6.333E+00
-8.500E+00	6.166E+00
-8.000E+00	6.000E+00
-7.500E+00	5.833E+00
-7.000E+00	5.666E+00
-6.500E+00	5.500E+00
-6.000E+00	5.333E+00
-5.500E+00	5.166E+00
-5.000E+00	4.999E+00
-4.500E+00	4.499E+00
-4.000E+00	3.999E+00
-3.500E+00	3.499E+00
-3.000E+00	2.999E+00
-2.500E+00	2.499E+00
-2.000E+00	1.999E+00

-1.500E+00	1.499E+00
-1.000E+00	9.991E-01
-5.000E-01	4.991E-01
0.000E+00	-8.353E-04
5.000E-01	-5.008E-01
1.000E+00	-1.001E+00
1.500E+00	-1.501E+00
2.000E+00	-2.001E+00
2.500E+00	-2.501E+00
3.000E+00	-3.001E+00
3.500E+00	-3.501E+00
4.000E+00	-4.001E+00
4.500E+00	-4.501E+00
5.000E+00	-5.000E+00
5.500E+00	-5.167E+00
6.000E+00	-5.333E+00
6.500E+00	-5.500E+00
7.000E+00	-5.667E+00
7.500E+00	-5.833E+00
8.000E+00	-6.000E+00
8.500E+00	-6.167E+00

9.000E+00	-6.333E+00
9.500E+00	-6.500E+00
1.000E+01	-6.667E+00
1.050E+01	-6.733E+00
1.100E+01	-6.800E+00
1.150E+01	-6.867E+00
1.200E+01	-6.933E+00
1.250E+01	-7.000E+00
1.300E+01	-7.067E+00
1.350E+01	-7.133E+00
1.400E+01	-7.200E+00
1.450E+01	-7.267E+00
1.500E+01	-7.333E+00


DIODE FUNCTION GENERATOR
OUTPUT VOLTAGE
INPUT VOLTAGE
DFG2.DAT
10
5
0
-5
-10
-20
-10
0
10
20

Appendix 4

Source code listing

Overview

The optical hardware is controlled by a PC compatible computer and two interface boards. Much software was developed not only to control the hardware, but also to train with the MR2 rule. To run the system, type **MENU** at the user's prompt and select the available choices.

The source code listings are well documented and provide a good explanation for all the functions. The system software consists of 12 separate files listed below followed by a brief description. **Uniform.c** contains functions for uniformly distributing network weights during weight initialization. **Rank.c** contains function for array indexing and ranking. **Network3.c** saves and recalls network weights for training and testing of patterns. **File.c** provide file input and output capability. **Train.c** provides a menu for training the network. **Test3.c** tests patterns after network has been trained. **Share3.c** provides MR2 training rule functions shared by optical system software, optical simulation software, and by traditional MR2 algorithm training. **Driver3.c** provides hardware digital and analog input and output functions. **Optical3.c , Sim3_d.c** and **3layer.c** are the actual MR2 training software for the three systems.

Appendix 4

Source code listing

Overview

The optical hardware is controlled by a PC compatible computer and two interface boards. Much software was developed not only to control the hardware, but also to train with the MR2 rule. To run the system, type **MENU** at the user's prompt and select the available choices.

The source code listings are well documented and provide a good explanation for all the functions. The system software consists of 12 separate files listed below followed by a brief description. **Uniform.c** contains functions for uniformly distributing network weights during weight initialization. **Rank.c** contains function for array indexing and ranking. **Network3.c** saves and recalls network weights for training and testing of patterns. **File.c** provide file input and output capability. **Train.c** provides a menu for training the network. **Test3.c** tests patterns after network has been trained. **Share3.c** provides MR2 training rule functions shared by optical system software, optical simulation software, and by traditional MR2 algorithm training. **Driver3.c** provides hardware digital and analog input and output functions. **Optical3.c , Sim3_d.c** and **3layer.c** are the actual MR2 training software for the three systems.

```
/*****************************************************************************
PROGRAM : MENU3.C
WRITTEN  BY: William Robinson
CLASS: Master's Thesis
DESCRIPTION: Provides simple menu interface for Optical ANN. Allows user
to select various modes of training.
*****************************************************************************/
#include "include3.c"
/*****************************************************************************
SYSTEM PARAMETERS function asks user for choice of system (Optical or
Software only) and for weight distribution bounds.
*****************************************************************************/
void system_parameters(void)
{
int num,num1;

clrscr();
gotoxy(1,5);
printf("Please enter initial weight uniform distribution parameters");
printf("\n Lower Bound : ");
scanf("%d",&Min);
printf("\n Upper Bound : ");
scanf("%d",&Max);
printf("\n Please enter number of iterations for training set (ie 3):");
scanf("%d",&Iterations);
clrscr();
do{
  printf("Please select 1: Optical or 2: Software Only : ");
  scanf("%d",&num);
  } while ((num < 1) | (num > 2));
if (num==1){
  OPTICAL = 1;
  SOFTWARE =0;
  }
else{
  SOFTWARE = 1;
 OPTICAL =0;
  do{
   printf("\n Please select training option ");
printf("\n 1. Normal MR2 training ");
printf("\n 2. Optical hardware simulation ");
printf("\n Enter option selected:   ");
    scanf("%d",&num1);
  } while ((num1 < 1) | (num1 > 2));
 SOFTWARE_VERSION = num1;
 }
system_selected = 1;
}
/*****************************************************************************
Main Program
*****************************************************************************/
int main(void)
{
int choice;
get_it=0;
```

```
save_it=0;
OPTICAL=0;
SOFTWARE=0;
system_selected=0;
/*****************************************************************************
Display the ANN Menu
*****************************************************************************/
do {
     clrscr();
     printf("\n");
     printf ("\t\tOptical ANN Main Menu\n");
     printf (" \n");
     printf ("\t 0. Quit\n");
     printf ("\t 1. System parameters \n");
     printf ("\t 2. File menu\n");
     printf ("\t 3. Train menu\n");
     printf ("\t 4. Test menu\n");
     printf ("\t->");
     scanf("%d",&choice);
     switch (choice) {
     case 0: exit(0);
              break;
     case 1: system_parameters();
             break;
     case 2: file_menu();
             break;
     case 3: train_menu();
              break;
     case 4: test_menu();
              break;
     default:break;
     }

   } while (choice != 0);
return 0;
}
/*****************************************************************************
FILE: include3.c
WRITTEN BY: WILLIAM ROBINSON
DESCRIPTION: Contains headers, definitions, and global variables used in the
optical neural network system simulation.
*****************************************************************************/
/*****************************************************************************
Standard include files
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <time.h>
#include <dos.h>
#include <string.h>
#include <ctype.h>
/******DEFINITIONS*************************************************************/
/******ADC  BOARD**************************************************************/
```

```
#define ADC_BASE        0x0320
#define TRIGGER         0
#define SET_GAIN        (ADC_BASE + 3)
#define GAIN            3
#define ADC_CONVERT     (ADC_BASE + 1)  /* 12 BIT CONVERSION */
#define SET_CHANNEL     (ADC_BASE + 2)
#define READ_LOW        ADC_BASE
#define READ_HIGH       (ADC_BASE + 1)
#define EOC             (ADC_BASE + 2)
#define ADC_BUSY        0x7F
/******DIGITAL I/O BOARD**********************************************/
#define BASE 0x300      /* Interface board base address */
#define SSBIT    0x00   /* Wiper Stack Select Bit */
#define ILED_LO  0x301  /* Base + 1  Port C */
#define ILED_HI  0x302  /* Base + 2  Port B */
#define HLED_LO  0x300  /* Base + 0  Port A */
#define HLED_HI  0x306  /* Base + 6  Port C */
#define HOUT_LO  0x30A  /* Base + 10 Port C */
#define HOUT_HI  0x309  /* Base + 9  Port B */
#define H2OUT_LO 0x308  /* Base + 8  Port A */
#define H2OUT_HI 0x30E  /* Base +14  Port C */
#define LCD1_DATA 1     /* Write to Base + 5  Port B0 */
#define LCD1_LOAD 2     /* Write to Base + 5  Port B1 */
#define LCD1_CLOCK 1    /* Write to Base + 4  Port A0 */
#define LCD2_DATA 4     /* Write to Base + 5  Port B2 */
#define LCD2_LOAD 8     /* Write to Base + 5  Port B3 */
#define LCD2_CLOCK 4    /* Write to Base + 4  Port A2 */
#define LAYER1_LOAD 32  /* Write to Base + 5  Port B5 */
#define LAYER1_CLOCK 2  /* Write to Base + 4  Port A1 */
#define LAYER2_LOAD 128 /* Write to Base + 5  Port B7 */
#define LAYER2_CLOCK 8  /* Write to Base + 4  Port A3 */

/******48 CHANNEL ANALOG INPUT BOARD**********************************/
/*********************************************************************
Function prototypes
*********************************************************************/
/*UNIFORM.C*/
double u32(void);
double uniform_dist(double a, double b);

/*RANK.C*/
void Indexx(int n,double arrin[],int indx[]);
void Rank(int n,int indx[],int irank[]);

/*DRIVER3.C*/
void ADC_LAYER1(void);
void ADC_LAYER2(void);
void MODE_0(void);
void ENABLE_WEIGHTS(void);
void WRITE_INPUT_VECTOR(unsigned char lsb,unsigned char msb);
void PRESENT_HIDDEN1_VECTOR(void);
void WRITE_HIDDEN1_BINARY(void);
void WRITE_LAYER1_WEIGHTS(void);
void WRITE_LAYER2_WEIGHTS(void);
void LEDS_OFF (void);
```

```
/*SHARE3.C*/
void initialize_weights(void);
void initialize_adapted(void);
void binarize_inputs(unsigned char lsb,unsigned char msb);
void generate_output_neurons(void);
void generate_neurons(void);
unsigned int hamming_distance(int pattern_number);
unsigned int test_output(int test_pattern_number);
void rank_hidden1_neurons(void);
void rank_hidden2_neurons(void);
void save_current_weights1(void);
void save_current_weights2(void);
void save_current_weights3(void);
void restore_prior_weights1(void);
void restore_prior_weights2(void);
void restore_prior_weights3(void);
void singlets1(int pattern_number);
void doublets1(int pattern_number);
void triplets1(int pattern_number);
void quadlets1(int pattern_number);
void singlets2(int pattern_number);
void doublets2(int pattern_number);
void triplets2(int pattern_number);
void quadlets2(int pattern_number);
void adapt_output_layer(int pattern_number);
void adapt_weights1(int neuron_number);
void adapt_weights2(int neuron_number);
void adapt_weights3(int neuron_number);

/*OPTICAL3*/
void read_hidden1_binary(void);
void read_hidden2_binary(void);
void read_hidden1_analog(void);
void read_hidden2_analog(void);
void compare_hidden1_neurons(void);
void compare_hidden2_neurons(void);
void optical_ann(void);

/*SIM3_D.C*/
double dynamic_threshold(int input[16]);
void Write_sim1_weights(void);
void Write_sim2_weights(void);
void simulate_layer1_neurons(void);
void simulate_layer2_neurons(void);
void simulate(void);

/*3LAYER.C*/
void generate_layer1_neurons(void);
void generate_layer2_neurons(void);
void normal_training(void);

/*NETWORK3.C */
int save_weight_layers(void);
int get_weight_layers(void);
```

```
/*FILE.C*/
void get_file(void);
void save_file(void);
void file_menu(void);

/*TRAIN.C*/
void one_run(void);
void multiple_runs(void);
void train_menu(void);

/*TEST3.C*/
unsigned int test_set(int test_pattern_number);
void test_network(void);
void one_test(void);
void multiple_tests(void);
void test_menu(void);

/*MENU3_D.C*/
void system_parameters(void);
/*************************************************************************
Global variables
*************************************************************************/
double Min=-60;
double Max=60;          /*For weights uniform distribution*/

int  Nflag = 1;
      /* for random number generator */

FILE  *in_file,
      *out_file;        /* file pointers for network weights */

double Seed1 = 12345.0,     /* initial seed for the random number gen */
       Seed2 = 67890.0;     /* initial seed for the random number gen */

char   in_name[12],out_name[12];
/*************************************************************************
Training Vectors are hardcoded in alpha arrays for easier coding.
The following are global arrays.
*************************************************************************/
/* Input Vectors */
unsigned char
 /* Hardware input vectors */
 Letter_lsb[24]={0x1F,0x1D,0x9F,0x3F,0x17,0xA0,0xE8,0xE0,0xE0,0x00,
0x00,0x08,0xBF,0xBF,0x9F,0xAF,0xBE,0xBF,0x3F,0xFF,0xBF, 0xBF,0xBF,0xBD},
Letter_msb[24]={0x80,0x83,0x80,0x80,0x80,0x80,0x83,0x83,0x8B,0xF4,
0xFC,0xFC,0x94,0x84,0x94,0x94,0x94,0x90,0x94,0x94,0x96,0xD4,0xB4,0x94},

  /* Target output vectors */
  Test_lsb[10]={0x9F,0x1F,0xE1,0xE0,0x00,0x20,0xB7,0x3F,0xBF,0xBB},
 Test_msb[10]={0x80,0x82,0x83,0xA3,0xDC,0xFC,0x94,0x94,0x9C,0x94};

 /* Target output vectors */
unsigned int Letter_target[24]={1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2};
unsigned int Letter_test[10]={1,1,1,1,1,1,1,2,2,2,2};
```

```
/*****************************************************************************/
int input_pattern[16];    /* +1,-1, or 0 binary inputs */
int CRITERIA;          /* Hamming distance requirement */
double Hidden1_analog_value[16];
int  Hidden1_binary_value[16];
int  Hidden1_rank_value[16];
int  Hidden1_adapted[16];       /*If binary value has been inverted */
double Hidden2_analog_value[16];
int  Hidden2_binary_value[16];
int  Hidden2_rank_value[16];
int  Hidden2_adapted[16];       /*If binary value has been inverted */
double Output_analog_value[5];
unsigned int  Output_binary_value[5];
int  Output_adapted[5];
int HammDist;    /* Hamming distance */
nt pattern_learned[24];
int iterations;
int get_it;
int save_it;
int system_selected;
int OPTICAL;
int SOFTWARE;
int SOFTWARE_VERSION;
double THRESHOLD1;
double THRESHOLD2;
double
    Weight_layer1[16][16],    /* Holds computer weights */
    Weight_layer2[16][16],
    Weight_layer3[16][5],
    Temp_layer1[16][16],      /* Temporary weight storage */
    Temp_layer2[16][16],
    Temp_layer3[16][5],
    Sim_layer1[16][16],       /* For storing simulated weights */
    Sim_layer2[16][16];
unsigned char
    Optical_layer1[16][16],   /* External hardware optical weights */
    Optical_layer2[16][16],
    Threshold_column;         /*Used for dynamic Thresholding */
/*****************************************************************************
User included files
*****************************************************************************/
#include "uniform.c"
#include "rank.c"
#include "network3.c"
#include "file.c"
#include "train.c"
#include "test3.c"
#include "share3.c"
#include "driver3.c"
#include "optical3.c"
#include "sim3_d.c"
#include "3layer.c"
/*****************************************************************************
```

FILE:share3.c

```
WRITTEN BY: William Robinson
DESCRIPTION: Contains functions shared by simulation and optical
hardware during training phase.
*****************************************************************************/
/*****************************************************************************
INITIALIZE WEIGHTS function uniformly distributes weight values in each
of the weight layers.
*****************************************************************************/
void initialize_weights(void)
{
int i,j;
for (i=0;i<16;i++)
     for (j=0;j<16;j++){
uniform_dist(Min,Max);
Weight_layer2[i][j] = uniform_dist(Min,Max);
        }
for (i=0;i<16;i++)
     for (j=0;j<3;j++)
Weight_layer3[i][j] = uniform_dist(Min,Max);
}
/*****************************************************************************
INITIALIZE adapted function clears hidden neuron adapted value.
*****************************************************************************/
void initialize_adapted(void)
{
int i;
for (i=0;i<16;i++){
     Hidden1_adapted[i] = 0;
     Hidden2_adapted[i] = 0;
     }
}
/*****************************************************************************
BINARIZE_INPUTS function reads msb and lsb input vector values and assigns
a corresponding +1 or 0. No symmetry is used. This is only used to
calculate hidden neuron analog values.
*****************************************************************************/
void binarize_inputs(unsigned char lsb,unsigned char msb)
{
int temp[15];
int number;
     temp[0]= lsb&1;
     temp[1]= (lsb&2)>>1;
     temp[2]= (lsb&4)>>2;
     temp[3]= (lsb&8)>>3;
     temp[4]= (lsb&16)>>4;
     temp[5]= (lsb&32)>>5;
     temp[6]= (lsb&64)>>6;
     temp[7]= (lsb&128)>>7;
     temp[8]= msb&1;
     temp[9]= (msb&2)>>1;
     temp[10]= (msb&4)>>2;
     temp[11]= (msb&8)>>3;
     temp[12]= (msb&16)>>4;
     temp[13]= (msb&32)>>5;
     temp[14]= (msb&64)>>6;
```

```
    temp[15] = 1;    /* Bias input set to +1 */
if (SOFTWARE_VERSION==1){

   for (number=0;number<16;number ++)
      if (temp[number]) input_pattern[number] = 1;
      else input_pattern[number]=-1;
   }
else {
   for (number=0;number<16;number ++)
      if (temp[number]) input_pattern[number] = 1;
      else input_pattern[number]=0;
   }
}
/*************************************************************************
GENERATE OUTPUT NEURONS function calculates the weighted sum for each
output neuron.
*************************************************************************/
void generate_output_neurons(void)
{
int number;
int column;
/*initialize values */
for (number=0;number<3;number++)
    Output_analog_value[number]=0;
for (number=0;number<3;number++){
    for (column=0;column<16;column++)
Output_analog_value[number] +=
  Hidden2_binary_value[column] * Weight_layer3[column][number];
       }
for (number=0;number<3;number++){
  if (Output_analog_value[number] >=0)
       Output_binary_value[number] = 1;
    else
       Output_binary_value[number] = 0;
   }
}
/*************************************************************************
GENERATE NEURONS function generates neurons for all 3 layers independent
of configuration.
*************************************************************************/
void generate_neurons(void)
{
   if (OPTICAL==1){
       compare_hidden1_neurons();
       PRESENT_HIDDEN1_VECTOR();
       compare_hidden2_neurons();
       generate_output_neurons();
       }
   else if (SOFTWARE_VERSION==2){
       simulate_layer1_neurons();
       simulate_layer2_neurons();
       generate_output_neurons();
       }

     else{
```

```
        generate_layer1_neurons();
        generate_layer2_neurons();
        generate_output_neurons();
        }
}
/*************************************************************************
HAMMING DISTANCE function looks at the number of differences in the actual
versus desired output vector.
*************************************************************************/
unsigned int hamming_distance(int pattern_number)
{
unsigned int actual_vector;
unsigned int differences;
unsigned int ex_or_vector;
int number;

actual_vector = (Output_binary_value[0] * 1) + (Output_binary_value[1] * 2) +
(Output_binary_value[2] * 4);
ex_or_vector = actual_vector ^ Letter_target[pattern_number];
differences = (ex_or_vector & 1) + ((ex_or_vector & 2) >>1) +
((ex_or_vector & 4) >> 2);
return(differences);
}
/*************************************************************************
TEST OUTPUT function looks for the test output neuron.
*************************************************************************/
unsigned int test_output(int test_pattern_number)
{
unsigned int actual_vector;
unsigned int differences;
unsigned int and_vector;
    actual_vector =  (Output_binary_value[0] * 1) + (Output_binary_value[1] * 2) +
(Output_binary_value[2] * 4);
and_vector = actual_vector & Letter_target[test_pattern_number];
differences = (and_vector & 1) + ((and_vector & 2) >>1) +
((and_vector & 4) >> 2);
return(differences);
}
/*************************************************************************
RANK HIDDEN1 NEURONS function ranks the value of the hidden neurons in
ascending order.
*************************************************************************/
void rank_hidden1_neurons(void)
{
int l=15;
int number;
double numarry[16];
int indxarry[16];
int rankarry[16];
numarry[0]=10000;  /* dummy entry */
for (number=0;number<15;number++)
     numarry[number+1]= Hidden1_analog_value[number];
Indexx(l,numarry,indxarry);
Rank(l,indxarry,rankarry);
for (number=0;number<15;number++)
```

```
    Hidden1_rank_value[number] = rankarry[number+1];
}

/*****************************************************************************
RANK HIDDEN2 NEURONS function ranks the value of the hidden neurons in
ascending order.
*****************************************************************************/
void rank_hidden2_neurons(void)
{
int i=15;
int number;
double numarry[16];
int indxarry[16];
int rankarry[16];
numarry[0]=10000;  /* dummy entry */
for (number=0;number<15;number++)
    numarry[number+1]= Hidden2_analog_value[number];

Indexx(i,numarry,indxarry);
Rank(i,indxarry,rankarry);
for (number=0;number<15;number++)
    Hidden2_rank_value[number] = rankarry[number+1];
}
/*****************************************************************************
SAVE CURRENT WEIGHTS1 function stores weight layer1 values.
*****************************************************************************/
void save_current_weights1(void)
{
int row,col;
for (row=0;row<16;row++)
    for(col=0;col<16;col++)
    Temp_layer1[row][col]=Weight_layer1[row][col];
}
/*****************************************************************************
RESTORE WEIGHTS1 function restores weight layer1 values.
*****************************************************************************/
void restore_prior_weights1(void)
{
int row,col;
for (row=0;row<16;row++)
    for(col=0;col<16;col++)
        Weight_layer1[row][col]=Temp_layer1[row][col];
if (OPTICAL==1) WRITE_LAYER1_WEIGHTS();
else if (SOFTWARE_VERSION==2) Write_sim1_weights();
}
/*****************************************************************************
SAVE CURRENT WEIGHTS2 function stores weight layer2 values.
*****************************************************************************/
void save_current_weights2(void)
{
int row,col;
for (row=0;row<16;row++)
    for(col=0;col<16;col++)
    Temp_layer2[row][col]=Weight_layer2[row][col];
}
```

```
/*************************************************************************
RESTORE WEIGHTS2 function restores weight layer2 values.
*************************************************************************/
void restore_prior_weights2(void)
{
int row,col;
for (row=0;row<16;row++)
     for(col=0;col<16;col++)
          Weight_layer2[row][col]=Temp_layer2[row][col];
if (OPTICAL==1) WRITE_LAYER2_WEIGHTS();
else if (SOFTWARE_VERSION==2) Write_sim2_weights();
}
/*************************************************************************
SAVE CURRENT WEIGHTS3 function stores weight layer3 values.
*************************************************************************/
void save_current_weights3(void)
{
int row,col;
for (row=0;row<16;row++)
     for(col=0;col<3;col++)
     Temp_layer3[row][col]=Weight_layer3[row][col];
}
/*************************************************************************
RESTORE WEIGHTS3 function restores weight layer3 values.
*************************************************************************/
void restore_prior_weights3(void)
{
int row,col;
for (row=0;row<16;row++)
     for(col=0;col<3;col++)
          Weight_layer3[row][col]=Temp_layer3[row][col];
}
/*************************************************************************
ADAPT WEIGHTS1 function adapts one neuron at a time until its binary value
is inverted.
*************************************************************************/
void adapt_weights1(int neuron_number)
{
int adapt_counter;
double adjust_value;
int row,column;
int inverted;
   if (OPTICAL==1){
     compare_hidden1_neurons();
     if (Hidden1_binary_value[neuron_number]==1)
       inverted = 0;
     else
       inverted = 1;
     }
   else if (SOFTWARE_VERSION==1){
     generate_layer1_neurons();
     if (Hidden1_analog_value[neuron_number]==1)
       inverted = -1;
     else
       inverted = 1;
```

```
    }
  else{
    simulate_layer1_neurons();
    if (Hidden1_analog_value[neuron_number]==1)
      inverted = 0;
    else
      inverted = 1;
    }
adjust_value=.005 ;
gotoxy(1,7);
printf("Hidden1 Neuron:\n");
printf("Value:\n");
printf("Binary:\n");
printf("Inverted:\n");
if ((OPTICAL==1) || (SOFTWARE_VERSION==2))
  printf("Dynamic Threshold Layer 1 Output:");
for (adapt_counter=0;adapt_counter<1000;adapt_counter++){
 for (row = 0; row < 16; row++){



    /* Test for adaptation direction */
    if (Hidden1_binary_value[neuron_number]==1){
    if (((Input_pattern[row]<=0)&&(Weight_layer1[row][neuron_number]<0))||
      ((Input_pattern[row]==1)&&(Weight_layer1[row][neuron_number]>0)))
      Weight_layer1[row][neuron_number] +=
      Weight_layer1[row][neuron_number]*(-adjust_value);
    else
      Weight_layer1[row][neuron_number] +=
      Weight_layer1[row][neuron_number]*adjust_value;
    }
    else {
    if (((Input_pattern[row]<=0)&&(Weight_layer1[row][neuron_number]<0))||
      ((Input_pattern[row]==1)&&(Weight_layer1[row][neuron_number]>0)))
      Weight_layer1[row][neuron_number] +=
      Weight_layer1[row][neuron_number]*adjust_value;
    else
      Weight_layer1[row][neuron_number] +=
      Weight_layer1[row][neuron_number]*(-adjust_value);
    }
   }
  if (OPTICAL==1){
    WRITE_LAYER1_WEIGHTS();
    compare_hidden1_neurons();
    }
  else if(SOFTWARE_VERSION==2)
    simulate_layer1_neurons();
  else
     generate_layer1_neurons();
  gotoxy(18,7);
  printf("%d",neuron_number);
  clreol();
  gotoxy(9,8);
  printf("%.2f",Hidden1_analog_value[neuron_number]);
  clreol();
```

```
    gotoxy(10,9);
    printf("%d",Hidden1_binary_value[neuron_number]);
    clreol();
    gotoxy(12,10);
    printf("%d",inverted);
    clreol();
    if ((OPTICAL==1) || (SOFTWARE_VERSION==2)){
      gotoxy(35,11);
      printf("%.2lf",THRESHOLD1);
      clreol();
      }

    /*Test if neuron is adapted */
    if (Hidden1_binary_value[neuron_number] == inverted) adapt_counter=1000;
    }
}
/*************************************************************************
ADAPT WEIGHTS2 function adapts one neuron at a time until its binary value
is inverted.
*************************************************************************/

void adapt_weights2(int neuron_number)
{
int adapt_counter;
double adjust_value;
int row,column;
int inverted;
    if (OPTICAL==1){
      compare_hidden2_neurons();
      if (Hidden2_binary_value[neuron_number]==1)
        inverted = -1;
      else
        inverted = 1;
      }
    else if (SOFTWARE_VERSION==1){
      generate_layer2_neurons();
      if (Hidden2_analog_value[neuron_number]==1)
        inverted = -1;
      else
        inverted = 1;
      }
    else{
      simulate_layer2_neurons();
      if (Hidden2_analog_value[neuron_number]==1)
        inverted = -1;
      else
        inverted = 1;
      }
adjust_value=.005 ;
gotoxy(1,7);
printf("Hidden2 Neuron:\n");
printf("Value:\n");
printf("Binary:\n");
printf("Inverted:\n");
if ((OPTICAL==1) || (SOFTWARE_VERSION==2))
```

```
  printf("Dynamic Threshold Layer 2 Output:");
for (adapt_counter=0;adapt_counter<1000;adapt_counter++){
 for (row = 0; row < 16; row++){
     /* Test for adaptation direction */
     if (Hidden2_binary_value[neuron_number]==1){
 if (((Hidden1_binary_value[row]<=0)&&(Weight_layer2[row][neuron_number]<0))||
 ((Hidden1_binary_value[row]==1)&&(Weight_layer2[row][neuron_number]>0)))
        Weight_layer2[row][neuron_number] +=
        Weight_layer2[row][neuron_number]*(-adjust_value);
      else
        Weight_layer2[row][neuron_number] +=
        Weight_layer2[row][neuron_number]*adjust_value;
      }
      else {
 if (((Hidden1_binary_value[row]<=0)&&(Weight_layer2[row][neuron_number]<0))||
 ((Hidden1_binary_value[row]==1)&&(Weight_layer2[row][neuron_number]>0)))
        Weight_layer2[row][neuron_number] +=
        Weight_layer2[row][neuron_number]*adjust_value;
      else
        Weight_layer2[row][neuron_number] +=
        Weight_layer2[row][neuron_number]*(-adjust_value);
      }
     }

   if (OPTICAL){
WRITE_LAYER2_WEIGHTS();
     compare_hidden2_neurons();
     }
   else if(SOFTWARE_VERSION==2)
     simulate_layer2_neurons();
   else
     generate_layer2_neurons();
   gotoxy(17,7);
   printf("%d",neuron_number);
   clreol();
   gotoxy(9,8);
   printf("%.2f",Hidden2_analog_value[neuron_number]);
   clreol();
   gotoxy(10,9);
   printf("%d",Hidden2_binary_value[neuron_number]);
   clreol();
   gotoxy(12,10);
   printf("%d",inverted);
   clreol();
   if ((OPTICAL==1) || (SOFTWARE_VERSION==2)){
   gotoxy(35,11);
     printf("%.2f",THRESHOLD2);
     clreol();
    }
   /*Test if neuron is adapted */
   if (Hidden2_binary_value[neuron_number] == inverted) adapt_counter=1000;
   }
}
/*************************************************************************
SINGLETS2 function adapts one neuron at a time. If change results in
```

```
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
**********************************************************************/
void singlets2(int pattern_number)
{
int number;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */
/*Rank analog neurons in order closest to zero*/
rank_hidden2_neurons();
/**********************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
**********************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
    save_current_weights2();
    CurrentHammDist = hamming_distance(pattern_number);
  /* Search for matching rank value */
  for (neuron_number=0; neuron_number < 15; neuron_number++)
      if(Hidden2_rank_value[neuron_number] == adapt_number){
         adapt_weights2(neuron_number);
         number=neuron_number;
         neuron_number=20;    /* exit  adapt loop */
         }

    generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
    HammDist=NewHammDist;
    /* Test for adaptation results */
   if (NewHammDist >= CurrentHammDist){
         restore_prior_weights2();
         Hidden2_adapted[number]=0;
         generate_neurons();
         }
    else
         Hidden2_adapted[number]=1;
   if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
         adapt_number = 20; /*Exit Singlets */
    }
}
/**********************************************************************
DOUBLETS2 function adapts two neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
**********************************************************************/
void doublets2(int pattern_number)
{
int number1,number2;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */
/*Rank analog neurons in order closest to zero*/
```

```
rank_hidden2_neurons();
/**************************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
**************************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
    save_current_weights2();
    CurrentHammDist = hamming_distance(pattern_number);
    /* Search for matching rank value and adapt in pairs neurons which have
    not been adapted */
  for (neuron_number=0; neuron_number < 15; neuron_number++)
      if((Hidden2_rank_value[neuron_number] == adapt_number)
       &&(Hidden2_adapted[neuron_number] == 0)){
        adapt_weights2(neuron_number);
        number1=neuron_number;
        neuron_number=20;    /* exit  adapt loop */
    }
     if (adapt_number <=14)
       for (neuron_number=0; neuron_number < 15; neuron_number++)
          if((Hidden2_rank_value[neuron_number] == adapt_number+1)
            &&(Hidden2_adapted[neuron_number] == 0)){
            adapt_weights2(neuron_number);
            number2=neuron_number;
            neuron_number=20;    /* exit  adapt loop */
            }
    generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
   HammDist=NewHammDist;
   /* Test for adaptation results */
     if (NewHammDist >= CurrentHammDist){
          restore_prior_weights2();
          Hidden2_adapted[number1]=0;
          Hidden2_adapted[number2]=0;

          generate_neurons();
          }
    else{
          Hidden2_adapted[number1]=1;
          Hidden2_adapted[number2]=1;
         }
     if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
          adapt_number = 20; /*Exit Doublets */
     }
}
/**************************************************************************
TRIPLETS2 function adapts three neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
**************************************************************************/
void triplets2(int pattern_number)
{
int number1,number2,number3;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
```

```
unsigned int NewHammDist; /* Used to compare hamming distances */
/*Rank analog neurons in order closest to zero*/
rank_hidden2_neurons();

/*************************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
*************************************************************************/
for (adapt_number = 2; adapt_number < 16; adapt_number++){
    save_current_weights2();
    CurrentHammDist = hamming_distance(pattern_number);
    /* Search for matching rank value and adapt in pairs neurons which have
    not been adapted */
    for (neuron_number=0; neuron_number < 15; neuron_number++)
      if((Hidden2_rank_value[neuron_number] == adapt_number)
        &&(Hidden2_adapted[neuron_number] == 0)){
          adapt_weights2(neuron_number);
          number1=neuron_number;
          neuron_number=20;    /* exit  adapt loop */
          }
    if (adapt_number <=14)
      for (neuron_number=0; neuron_number < 15; neuron_number++)
        if((Hidden2_rank_value[neuron_number] == adapt_number+1)
          &&(Hidden2_adapted[neuron_number] == 0)){
          adapt_weights2(neuron_number);
          number2=neuron_number;
          neuron_number=20;    /* exit  adapt loop */
          }

    if (adapt_number <=13)
      for (neuron_number=0; neuron_number < 15; neuron_number++)
        if((Hidden2_rank_value[neuron_number] == adapt_number+2)
          &&(Hidden2_adapted[neuron_number] == 0)){
          adapt_weights2(neuron_number);
          number3=neuron_number;
          neuron_number=20;    /* exit  adapt loop */
          }
    generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
   HammDist=NewHammDist;
   /* Test for adaptation results */
    if (NewHammDist >= CurrentHammDist){
        restore_prior_weights2();
        Hidden2_adapted[number1]=0;
        Hidden2_adapted[number2]=0;
        Hidden2_adapted[number3]=0;
        generate_neurons();
        }
    else{
        Hidden2_adapted[number1]=1;
        Hidden2_adapted[number2]=1;
        Hidden2_adapted[number3]=1;
       }
    if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
        adapt_number = 20; /*Exit Triplets */
```

```
    }
}
/*****************************************************************
QUADLETS2 function adapts three neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
*****************************************************************/
void quadlets2(int pattern_number)
{
int number1,number2,number3,number4;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */

/*Rank analog neurons in order closest to zero*/

rank_hidden2_neurons();
/*****************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
*****************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
   save_current_weights2();
   CurrentHammDist = hamming_distance(pattern_number);
   /* Search for matching rank value and adapt in pairs neurons which have
   not been adapted */
   for (neuron_number=0; neuron_number < 15; neuron_number++)
     if((Hidden2_rank_value[neuron_number] == adapt_number)
      &&(Hidden2_adapted[neuron_number] == 0)){
       adapt_weights2(neuron_number);
       number1=neuron_number;
       neuron_number=20;    /* exit  adapt loop */
       }
   if (adapt_number <=14)
     for (neuron_number=0; neuron_number < 15; neuron_number++)
       if((Hidden2_rank_value[neuron_number] == adapt_number+1)
         &&(Hidden2_adapted[neuron_number] == 0)){
         adapt_weights2(neuron_number);
         number2=neuron_number;

         neuron_number=20;    /* exit  adapt loop */
         }
    if (adapt_number <=13)
      for (neuron_number=0; neuron_number < 15; neuron_number++)
        if((Hidden2_rank_value[neuron_number] == adapt_number+2)
         &&(Hidden2_adapted[neuron_number] == 0)){
         adapt_weights2(neuron_number);
         number3=neuron_number;
         neuron_number=20;    /* exit  adapt loop */
         }
    if (adapt_number <=12)
      for (neuron_number=0; neuron_number < 15; neuron_number++)
        if((Hidden2_rank_value[neuron_number] == adapt_number+3)
         &&(Hidden2_adapted[neuron_number] == 0)){
```

```
            adapt_weights2(neuron_number);
            number4=neuron_number;
            neuron_number=20;    /* exit  adapt loop */
            }
     generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
    HammDist=NewHammDist;
    /* Test for adaptation results */
    if (NewHammDist >= CurrentHammDist){
        restore_prior_weights2();
        Hidden2_adapted[number1]=0;
        Hidden2_adapted[number2]=0;
        Hidden2_adapted[number3]=0;
        Hidden2_adapted[number4]=0;
        generate_neurons();
        }
    else{
        Hidden2_adapted[number1]=1;
        Hidden2_adapted[number2]=1;
        Hidden2_adapted[number3]=1;
        Hidden2_adapted[number4]=1;
       }
    if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
        adapt_number = 20; /*Exit Quadlets */
    }
}
/*************************************************************************
SINGLETS1 function adapts one neuron at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
*************************************************************************/
void singlets1(int pattern_number)
{
int number;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */
/*Rank analog neurons in order closest to zero*/
rank_hidden1_neurons();
/*************************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
*************************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
    save_current_weights1();
    CurrentHammDist = hamming_distance(pattern_number);
  /* Search for matching rank value */
  for (neuron_number=0; neuron_number < 15; neuron_number++)
     if(Hidden1_rank_value[neuron_number] == adapt_number){
        adapt_weights1(neuron_number);
        number=neuron_number;
        neuron_number=20;    /* exit  adapt loop */
        }
```

```
    generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
    HammDist=NewHammDist;
    /* Test for adaptation results */
    if (NewHammDist >= CurrentHammDist){
        restore_prior_weights1();
        Hidden1_adapted[number]=0;
        generate_neurons();
        }
    else
        Hidden1_adapted[number]=1;
    if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
        adapt_number = 20; /*Exit Singlets */
    }
}
/*************************************************************************
DOUBLETS1 function adapts two neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
*************************************************************************/
void doublets1(int pattern_number)
{
int number1,number2;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */
/*Rank analog neurons in order closest to zero*/
rank_hidden1_neurons();
/*************************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
*************************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
    save_current_weights1();
    CurrentHammDist = hamming_distance(pattern_number);
    /* Search for matching rank value and adapt in pairs neurons which have
    not been adapted */
  for (neuron_number=0; neuron_number < 15; neuron_number++)
      if((Hidden1_rank_value[neuron_number] == adapt_number)
        &&(Hidden1_adapted[neuron_number] == 0)){
         adapt_weights1(neuron_number);
         number1=neuron_number;
         neuron_number=20;    /* exit  adapt loop */
    }
     if (adapt_number <=14)
       for (neuron_number=0; neuron_number < 15; neuron_number++)
          if((Hidden1_rank_value[neuron_number] == adapt_number+1)
            &&(Hidden1_adapted[neuron_number] == 0)){
            adapt_weights1(neuron_number);
            number2=neuron_number;
            neuron_number=20;    /* exit  adapt loop */
            }
    generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
```

```
  HammDist=NewHammDist;
  /* Test for adaptation results */
   if (NewHammDist >= CurrentHammDist){
        restore_prior_weights1();
        Hidden1_adapted[number1]=0;
        Hidden1_adapted[number2]=0;

        generate_neurons();
        }
    else{
        Hidden1_adapted[number1]=1;
        Hidden1_adapted[number2]=1;
       }
    if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
        adapt_number = 20; /*Exit Doublets */
    }
}
/*************************************************************************
TRIPLETS1 function adapts three neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
*************************************************************************/
void triplets1(int pattern_number)
{
int number1,number2,number3;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */
/*Rank analog neurons in order closest to zero*/
rank_hidden1_neurons();

/*************************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
*************************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
    save_current_weights1();
    CurrentHammDist = hamming_distance(pattern_number);
    /* Search for matching rank value and adapt in pairs neurons which have
    not been adapted */
    for (neuron_number=0; neuron_number < 15; neuron_number++)
      if((Hidden1_rank_value[neuron_number] == adapt_number)
       &&(Hidden1_adapted[neuron_number] == 0)){
        adapt_weights1(neuron_number);
        number1=neuron_number;
        neuron_number=20;    /* exit  adapt loop */
        }
    if (adapt_number <=14)
      for (neuron_number=0; neuron_number < 15; neuron_number++)
        if((Hidden1_rank_value[neuron_number] == adapt_number+1)
          &&(Hidden1_adapted[neuron_number] == 0)){
          adapt_weights1(neuron_number);
          number2=neuron_number;
          neuron_number=20;    /* exit  adapt loop */
```

```
        }

    if (adapt_number <=13)
      for (neuron_number=0; neuron_number < 15; neuron_number++)
         if((Hidden1_rank_value[neuron_number] == adapt_number+2)
           &&(Hidden1_adapted[neuron_number] == 0)){
           adapt_weights1(neuron_number);
           number3=neuron_number;
           neuron_number=20;    /* exit  adapt loop */
           }
    generate_neurons();
    NewHammDist = hamming_distance(pattern_number);
   HammDist=NewHammDist;
   /* Test for adaptation results */
    if (NewHammDist >= CurrentHammDist){
         restore_prior_weights1();
         Hidden1_adapted[number1]=0;
         Hidden1_adapted[number2]=0;
         Hidden1_adapted[number3]=0;
         generate_neurons();
         }
    else{
         Hidden1_adapted[number1]=1;
         Hidden1_adapted[number2]=1;
         Hidden1_adapted[number3]=1;
        }
    if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
         adapt_number = 20; /*Exit Triplets */
    }
}
/*************************************************************************
QUADLETS1 function adapts three neurons at a time. If change results in
reducing Hamming Distance then change is kept. Otherwise weights are
restored to original values.
*************************************************************************/
void quadlets1(int pattern_number)
{
int number1,number2,number3,number4;
int neuron_number;
int adapt_number;
unsigned int CurrentHammDist; /* Hamming distance value */
unsigned int NewHammDist; /* Used to compare hamming distances */

/*Rank analog neurons in order closest to zero*/

rank_hidden1_neurons();
/*************************************************************************
Adapt in direction to change corresponding binary output
of hidden layer
*************************************************************************/
for (adapt_number = 1; adapt_number < 16; adapt_number++){
    save_current_weights1();
    CurrentHammDist = hamming_distance(pattern_number);
    /* Search for matching rank value and adapt in pairs neurons which have
    not been adapted */
```

```
  for (neuron_number=0; neuron_number < 15; neuron_number++)
    if((Hidden1_rank_value[neuron_number] == adapt_number)
     &&(Hidden1_adapted[neuron_number] == 0)){
      adapt_weights1(neuron_number);
      number1=neuron_number;
      neuron_number=20;    /* exit  adapt loop */
      }
  if (adapt_number <=14)
    for (neuron_number=0; neuron_number < 15; neuron_number++)
      if((Hidden1_rank_value[neuron_number] == adapt_number+1)
       &&(Hidden1_adapted[neuron_number] == 0)){
       adapt_weights1(neuron_number);
       number2=neuron_number;

       neuron_number=20;    /* exit  adapt loop */
       }
   if (adapt_number <=13)
     for (neuron_number=0; neuron_number < 15; neuron_number++)
       if((Hidden1_rank_value[neuron_number] == adapt_number+2)
        &&(Hidden1_adapted[neuron_number] == 0)){
        adapt_weights1(neuron_number);
        number3=neuron_number;
        neuron_number=20;    /* exit  adapt loop */
        }
   if (adapt_number <=12)
     for (neuron_number=0; neuron_number < 15; neuron_number++)
       if((Hidden1_rank_value[neuron_number] == adapt_number+3)
        &&(Hidden1_adapted[neuron_number] == 0)){
        adapt_weights1(neuron_number);
        number4=neuron_number;
        neuron_number=20;    /* exit  adapt loop */
        }
    generate_neurons();
   NewHammDist = hamming_distance(pattern_number);
   HammDist=NewHammDist;
   /* Test for adaptation results */
   if (NewHammDist >= CurrentHammDist){
       restore_prior_weights1();
       Hidden1_adapted[number1]=0;
       Hidden1_adapted[number2]=0;
       Hidden1_adapted[number3]=0;
       Hidden1_adapted[number4]=0;
       generate_neurons();
       }
   else{
       Hidden1_adapted[number1]=1;
       Hidden1_adapted[number2]=1;
       Hidden1_adapted[number3]=1;
       Hidden1_adapted[number4]=1;
      }
   if ((NewHammDist <= CRITERIA) && (test_output(pattern_number)))
       adapt_number = 20; /*Exit Quadlets */
   }
}
/*****************************************************************************
```

```
ADAPT WEIGHTS 3 function adapts third layer neurons.
**********************************************************************/
void adapt_weights3(int neuron_number)
{
int adapt_counter;
double adjust_value;
int row,column;
int inverted;
float threshold;
if (Output_analog_value[neuron_number]>=0)
    inverted = 0;
  else
    inverted = 1;
adjust_value=.005 ;
gotoxy(1,13);
printf("Output neuron:\n");
printf("Value:\n");
printf("Binary:\n");
printf("Inverted:");
for (adapt_counter=0;adapt_counter<1000;adapt_counter++){
 for (row = 0; row < 16; row++){
    /* Test for adaptation direction */
    if (Output_binary_value[neuron_number]==1){
 if (((Hidden2_binary_value[row]<0)&&(Weight_layer3[row][neuron_number]<0))||
   ((Hidden2_binary_value[row]>0)&&(Weight_layer3[row][neuron_number]>0)))
       Weight_layer3[row][neuron_number] +=
       Weight_layer3[row][neuron_number]*(-adjust_value);
     else
       Weight_layer3[row][neuron_number] +=
       Weight_layer3[row][neuron_number]*adjust_value;
    }
    else {
 if (((Hidden2_binary_value[row]<0)&&(Weight_layer3[row][neuron_number]<0))||
   ((Hidden2_binary_value[row]>0)&&(Weight_layer3[row][neuron_number]>0)))
       Weight_layer3[row][neuron_number] +=
       Weight_layer3[row][neuron_number]*adjust_value;
     else
       Weight_layer3[row][neuron_number] +=
       Weight_layer3[row][neuron_number]*(-adjust_value);
    }
   }
  generate_output_neurons();
  gotoxy(15,13);
  printf("%d",neuron_number);
  clreol();
  gotoxy(9,14);
  printf("%.2f",Output_analog_value[neuron_number]);
  clreol();
  gotoxy(10,15);
  printf("%d",Output_binary_value[neuron_number]);
  clreol();
  gotoxy(12,16);
  printf("%d",inverted);
  clreol();
  /*Test if neuron is adapted */
```

```
   if (Output_binary_value[neuron_number] == inverted)                adapt_counter=1000;
   }
}
/*************************************************************************
ADAPT OUTPUT LAYER function adapts one neuron at a time until its binary
value is inverted.
*************************************************************************/
void adapt_output_layer(int pattern_number)
{
unsigned int actual_vector;
unsigned int ex_or_vector;
unsigned int adapt[3];
int number;
int adapt_number;
int column;
generate_output_neurons();
save_current_weights3();
generate_output_neurons();
actual_vector = (Output_binary_value[0] * 1) + (Output_binary_value[1] * 2) +
(Output_binary_value[2] * 4);
ex_or_vector = actual_vector ^ Letter_target[pattern_number];
/*************************************************************************
Adapt in direction to match corresponding binary output
*************************************************************************/
for (number=0;number<3;number++){
    adapt[number]= (ex_or_vector & (unsigned int) pow(2,number))>>number;
    if (adapt[number]==1)
        adapt_weights3(number);
    }
generate_output_neurons();
HammDist = hamming_distance(pattern_number);
if (HammDist <=CRITERIA)
   printf("Corresponding output neurons adapted, get next pattern \n");
  else {
     restore_prior_weights3();
     generate_output_neurons();
     }
}
/*************************************************************************
FILE: NETWORK3.C
WRITTEN BY: William Robinson
DESCRIPTION: Saves and recalls weight values from a user specified file.
*************************************************************************/
/*************************************************************************
SAVE WEIGHTS LAYERS saves Weight_layer1 and Weight_layer2 arrays to a file.
*************************************************************************/
int save_weight_layers(void)
{
int row,column;
   if ((out_file = fopen(out_name, "w")) == NULL)
   {
      clrscr();
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
```

```
for (row=0;row<16;row++)
   for (column=0;column<16;column++)
      fprintf(out_file, "%f\n",Weight_layer1[row][column]);
for (row=0;row<16;row++)
   for (column=0;column<16;column++)
      fprintf(out_file, "%f\n",Weight_layer2[row][column]);
for (row=0;row<3;row++)
   for (column=0;column<16;column++)
      fprintf(out_file, "%f\n",Weight_layer3[row][column]);
for (row=0;row<24;row++)
      fprintf(out_file, "Pattern[%d]=%d\n",row,pattern_learned[row]);
fprintf(out_file, "Iterations performed:=%d\n",iterations);
/*Close the file*/
fclose(out_file);
return 0;
}
/*************************************************************************
GET WEIGHT LAYERS retrieves weight layers 1 and 2 from a file and assigns
them to Weight_layer1 and Weight_layer2 arrays.
*************************************************************************/
int get_weight_layers(void)
{
int row,column;
double temp;
if ((in_file = fopen(in_name, "r"))== NULL)
  {
    clrscr();
    fprintf(stderr, "Cannot open input file.\n");
    return 1;
  }
for (row=0;row<16;row++)
   for (column=0;column<16;column++){
      fscanf(in_file, "%lf\n",&temp);
      Weight_layer1[row][column]=temp;
      }
for (row=0;row<16;row++)
   for (column=0;column<16;column++){
      fscanf(in_file, "%lf\n",&temp);
      Weight_layer2[row][column]=temp;
      }
for (row=0;row<3;row++)
   for (column=0;column<16;column++){
      fscanf(in_file, "%lf\n",&temp);
      Weight_layer3[row][column]=temp;
      }
/*Close the file*/
fclose(in_file);
return 0;
}
/*************************************************************************
FILE : SIM3_D.C
WRITTEN BY: William Robinson
DESCRIPTION: Simulates Optical ANN hardware and trains with MR2 Rule.
Incorporates input dependent dynamic thresholding.
*************************************************************************/
```

```
/*************************************************************************
DYNAMIC THRESHOLD function calculates the threshold sum for each input
pattern through column sixteen.
*************************************************************************/
double dynamic_threshold(int input[16])
{
int input_neuron;
int Optical_mid = 90;
float sum;
sum=0;
for (input_neuron=0;input_neuron<16;input_neuron++)
    sum+= (input[input_neuron]*(Optical_mid));
return(sum);
}
/*************************************************************************
WRITE SIM1 WEIGHTS function converts the weights and biases into usable
forms for the  optical simulation weights
*************************************************************************/
void Write_sim1_weights(void)
{
int i,j;
  for (i = 0; i < 15; i++)
     for (j = 0; j < 15; j++) {
       if (Weight_layer1[i][j] < -60)
          Sim_layer1[i][j] = 30;
       else if(Weight_layer1[i][j] > 60)
          Sim_layer1[i][j] = 150;
       else
          Sim_layer1[i][j] = (int)Weight_layer1[i][j]+90;
       }
}
/*************************************************************************
WRITE SIM2 WEIGHTS function converts the weights and biases into usable
forms for the  optical simulation weights
*************************************************************************/
void Write_sim2_weights(void)
{
int i,j;
  for (i = 0; i < 15; i++)
     for (j = 0; j < 15; j++) {
       if (Weight_layer2[i][j] < -60)
          Sim_layer2[i][j] = 30;
       else if(Weight_layer2[i][j] > 60)
          Sim_layer2[i][j] = 150;
       else
          Sim_layer2[i][j] = (int)Weight_layer2[i][j]+90;
       }
}
/*************************************************************************
SIMULTATE LAYER 1 NEURONS function calculates the weighted sum for each
neuron in layer 1.
*************************************************************************/
void simulate_layer1_neurons(void)
{
int number;
```

```
int row,column;
Write_sim1_weights();
THRESHOLD1= (dynamic_threshold(Input_pattern));
/****Generate Hidden Layer 1 Neurons ****************************************/
/*Initialize values */
for (number=0;number<16;number++)
     Hidden1_analog_value[number]=0;
for (number=0;number<16;number++)
for (column=0;column<16;column++)
     Hidden1_analog_value[number] += Input_pattern[column] * Sim_layer1[column][number];
for (number=0;number<15;number++){
     if (Hidden1_analog_value[number] >= (dynamic_threshold(Input_pattern)))
        Hidden1_binary_value[number] = 1;
     else
        Hidden1_binary_value[number] = 0;
    }
Hidden1_binary_value[15] = 1; /*Bias neuron */
}
/*****************************************************************************
SIMULATE LAYER 2 NEURONS function calculates the weighted sum for each
neuron in layer 2.
*****************************************************************************/
void simulate_layer2_neurons(void)
{
int number;
int row,column;
Write_sim2_weights();
THRESHOLD2= (dynamic_threshold(Hidden1_binary_value));
/****Generate Hidden Layer 2 Neurons ****************************************/
/*Initialize values */
for (number=0;number<16;number++)
     Hidden2_analog_value[number]=0;
for (number=0;number<16;number++)
     for (column=0;column<16;column++)
         Hidden2_analog_value[number] +=
  Hidden1_binary_value[column] * Sim_layer2[column][number];
for (number=0;number<15;number++){
   if(Hidden2_analog_value[number] >=(dynamic_threshold(Hidden1_binary_value)))
        Hidden2_binary_value[number] = 1;
     else
        Hidden2_binary_value[number] = -1;
    }
Hidden2_binary_value[15] = 1; /*Bias neuron */
}
/*****************************************************************************
Madaline Rule Two Algorithm
*****************************************************************************/
void simulate(void)
{
int outer_loop;
int pattern_counter;     /* pattern number */
int i;
clrscr();
gotoxy(1,2);
printf("TRAINING...\n");
```

```
printf("Hamming Distance Criteria is: %d",CRITERIA);
gotoxy(1,4);
printf("Adapting Weigths for pattern: ");
/* Uniform distribution of weights */
initialize_weights();
/*****Begin training*************************************************************/
for (outer_loop=0;outer_loop<iterations;outer_loop++){
for (pattern_counter=0;pattern_counter<24;pattern_counter++){
/*pattern_counter=2; */
    initialize_adapted();

/*******Present input vector*****************************************************/
    binarize_inputs(Letter_lsb[pattern_counter], Letter_msb[pattern_counter]);

/*******Check output vector and calculate Hamming Distance***********************/
    simulate_layer1_neurons();
    simulate_layer2_neurons();
    generate_output_neurons();
    gotoxy(31,4);
    clreol();
    gotoxy(31,4);
    printf("%d",pattern_counter);
    HammDist = hamming_distance(pattern_counter);
    if (HammDist > CRITERIA)
       singlets1(pattern_counter);
    if (HammDist > CRITERIA)
        doublets1(pattern_counter);
    if (HammDist > CRITERIA)
        triplets1(pattern_counter);
    if (HammDist > CRITERIA)
        quadlets1(pattern_counter);
    if (HammDist > CRITERIA)
       singlets2(pattern_counter);
    if (HammDist > CRITERIA)
       doublets2(pattern_counter);
    if (HammDist > CRITERIA)
        triplets2(pattern_counter);
    if (HammDist > CRITERIA)
      quadlets2(pattern_counter);
    if (HammDist > CRITERIA)
      adapt_output_layer(pattern_counter);
    if (HammDist > CRITERIA)
      pattern_learned[pattern_counter]=0;
    else
      pattern_learned[pattern_counter]=1;
/*********************************************************************************
    Apply next pattern
*********************************************************************************/
    }
}
/*********************************************************************************
    Store weights in network file and exit
*********************************************************************************/
save_weight_layers();
}
```

```
/*************************************************************************
FILE: OPTICAL3.C
WRITTEN BY: William Robinson
DESCRIPTION: Performs Madaline Rule 2 training with optical hardware.
*************************************************************************/
/*************************************************************************
READ_HIDDEN1_BINARY function reads binary hidden outputs and assign to
neurons in hidden layer . Function is used for testing binary outputs
generated by the hardware and is not used during training.
*************************************************************************/
void read_hidden1_binary(void)
{
unsigned char hidden1_vector_low;
unsigned char hidden1_vector_high;
int number;
hidden1_vector_low=inportb(HOUT_LO);
hidden1_vector_high=inportb(HOUT_HI);
for (number=0; number<8; number++){
     Hidden1_binary_value[number] =
     (hidden1_vector_low & (unsigned int) pow(2,number))>>number;
     Hidden1_binary_value[number+8]=
     (hidden1_vector_high & (unsigned int) pow(2,number))>>number;
    }

Hidden1_binary_value[15] = 1;   /*BIAS INPUT TO LAYER 2*/
}
/*************************************************************************
READ_HIDDEN2_BINARY function reads binary hidden outputs and assign to
neurons in hidden layer. Function is used for testing binary outputs
generated by the hardware and is not used during training.
*************************************************************************/
void read_hidden2_binary(void)
{
unsigned char hidden2_vector_low;
unsigned char hidden2_vector_high;
int number;
hidden2_vector_low=inportb(HOUT_LO);
hidden2_vector_high=inportb(HOUT_HI);
for (number=0; number<8; number++){
     Hidden2_binary_value[number] =
     (hidden2_vector_low & (unsigned int) pow(2,number))>>number;
     Hidden2_binary_value[number+8]=
     (hidden2_vector_high & (unsigned int) pow(2,number))>>number;
    }
/* Feeds -1 or +1 to software layer 3 which accepts symmetric +/-1 inputs */
for (number=0; number<15; number++){
     if (!Hidden2_binary_value[number])
        Hidden2_binary_value[number] = -1;
    }
Hidden2_binary_value[15] = 1;  /* BIAS INPUT TO LAYER 3*/
}
/*************************************************************************
READ HIDDEN1 ANALOG reads optical layer 1 output neurons analog value.
*************************************************************************/
void read_hidden1_analog(void)
```

```
{
   ADC_LAYER1();
}
/*************************************************************************
READ HIDDEN2 ANALOG reads optical layer 1 output neurons analog value.
*************************************************************************/
void read_hidden2_analog(void)
{
   ADC_LAYER2();
}
/*************************************************************************
COMPARE HIDDEN1 NEURONS function reads ADC layer 1 channels and compares
their value against threshold column for binary output determination.
*************************************************************************/
void compare_hidden1_neurons(void)
{
int neuron;
read_hidden1_analog();
THRESHOLD1=Hidden1_analog_value[15];
for (neuron=0;neuron<15;neuron++){
    if (Hidden1_analog_value[neuron] >= Hidden1_analog_value[15])
      Hidden1_binary_value[neuron]=1;
    else
      Hidden1_binary_value[neuron]=0;
    }
Hidden1_binary_value[15] = 1; /*Bias neuron */
}
/*************************************************************************
COMPARE HIDDEN2 NEURONS function reads ADC layer 1 channels and compares
their value against threshold column for binary output determination.
*************************************************************************/
void compare_hidden2_neurons(void)
{
int neuron;
read_hidden2_analog();
THRESHOLD2=Hidden2_analog_value[15];
for (neuron=0;neuron<15;neuron++){
    if (Hidden2_analog_value[neuron] >= Hidden2_analog_value[15])
      Hidden2_binary_value[neuron]=1;
    else
      Hidden2_binary_value[neuron]=-1;
    }
Hidden2_binary_value[15] = 1; /*Bias neuron */
}
/*************************************************************************
OPTICAL ANN function performs Madaline Two Rule training on Optical
ANN system.
*************************************************************************/
void optical_ann(void)
{
int outer_loop;
int pattern_counter;     /* pattern number */
int i;
MODE_0();         /* Configure interface board inside 386 computer */
ENABLE_WEIGHTS();  /* Allows SLM to be programmed */
```

```
clrscr();
gotoxy(1,2);
printf("TRAINING...\n");
printf("Hamming Distance Criteria is: %d",CRITERIA);
gotoxy(1,4);
printf("Adapting Weigths for pattern: ");
/* Uniform distribution of weights */
initialize_weights();
/**************************************************************************
Get weight layers 1 and 2 stored in computer memory and transfer
 to Optical hardware weights using interface board.
**************************************************************************/
WRITE_LAYER1_WEIGHTS();
WRITE_LAYER2_WEIGHTS();
/*****Begin training*******************************************************/
for (outer_loop=0;outer_loop<iterations;outer_loop++){
for (pattern_counter=0;pattern_counter<24;pattern_counter++){
/*pattern_counter=2; */
    initialize_adapted();
/*******Present input vector***********************************************/
    binarize_inputs(Letter_lsb[pattern_counter],
Letter_msb[pattern_counter]);
    WRITE_INPUT_VECTOR(Letter_lsb[pattern_counter],
      Letter_msb[pattern_counter]);
    compare_hidden1_neurons();
    PRESENT_HIDDEN1_VECTOR();
    compare_hidden2_neurons();
    gotoxy(31,4);
    clreol();
    gotoxy(31,4);
    printf("%d",pattern_counter);
/*******Check output vector and calculate Hamming Distance***************/
    HammDist = hamming_distance(pattern_counter);
    if (HammDist > CRITERIA)
        singlets1(pattern_counter);
    if (HammDist > CRITERIA)
        doublets1(pattern_counter);
    if (HammDist > CRITERIA)
        triplets1(pattern_counter);
    if (HammDist > CRITERIA)
        quadlets1(pattern_counter);
   if (HammDist > CRITERIA)
    • singlets2(pattern_counter);
    if (HammDist > CRITERIA)
      doublets2(pattern_counter);
    if (HammDist > CRITERIA)
      triplets2(pattern_counter);
    if (HammDist > CRITERIA)
      quadlets2(pattern_counter);
    if (HammDist > CRITERIA)
      adapt_output_layer(pattern_counter);
    if (HammDist > CRITERIA)
       pattern_learned[pattern_counter]=0;
    else
  pattern_learned[pattern_counter]=1;
```

```
/*************************************************************************
    Apply next pattern
*************************************************************************/
    }
}
/*************************************************************************
    Store weights in network file and exit
*************************************************************************/
save_weight_layers();
}
/*************************************************************************
/*************************************************************************
FILE : 3LAYER.C
WRITTEN BY: William Robinson
DESCRIPTION: Trains a three layer ANN consisting of  16x1--16x1--5x1

with MR2 Rule. Provides user with comparison against simulation and
optical hardware performance.
*************************************************************************/
/*************************************************************************
GENERATE LAYER 1 NEURONS function calculates the weighted sum for each
neuron in layer 1.
*************************************************************************/
void generate_layer1_neurons(void)
{
int number;
int column;
/****Generate Hidden Layer 1 Neurons ************************************/
/*Initialize values */
for (number=0;number<16;number++)
    Hidden1_analog_value[number]=0;
for (number=0;number<16;number++)
    for (column=0;column<16;column++)
        Hidden1_analog_value[number] +=
      Input_pattern[column] * Weight_layer1[column][number];
for (number=0;number<15;number++)
    if (Hidden1_analog_value[number] >= 0)
      Hidden1_binary_value[number] = 1;
    else
      Hidden1_binary_value[number] = -1;
Hidden1_binary_value[15] = 1; /*Bias neuron */
}
/*************************************************************************
GENERATE LAYER 2 NEURONS function calculates the weighted sum for each
neuron in layer 2.
*************************************************************************/
void generate_layer2_neurons(void)
{
int number;
int column;
/****Generate Hidden Layer 2 Neurons ************************************/
/*Initialize values */
for (number=0;number<16;number++)
    Hidden2_analog_value[number]=0;
for (number=0;number<16;number++)
```

```
    for (column=0;column<16;column++)
        Hidden2_analog_value[number] +=
      Hidden1_binary_value[column] * Weight_layer2[column][number];
for (number=0;number<15;number++)
  if(Hidden2_analog_value[number] >= 0)
        Hidden2_binary_value[number] = 1;
    else
        Hidden2_binary_value[number] = -1;
Hidden2_binary_value[15] = 1; /*Bias neuron */
}
/*************************************************************************
    Madaline Rule Two Algorithm
*************************************************************************/
void normal_training(void)
{
int outer_loop;
int pattern_counter;     /* pattern number */
int i;
clrscr();
gotoxy(1,2);
printf("TRAINING...\n");
printf("Hamming Distance Criteria is: %d",CRITERIA);
gotoxy(1,4);
printf("Adapting Weigths for pattern: ");
/* Uniform distribution of weights */
initialize_weights();

/*****Begin training*****************************************************/
for (outer_loop=0;outer_loop<Iterations;outer_loop++){
for (pattern_counter=0;pattern_counter<24;pattern_counter++){
    initialize_adapted();
/*******Present input vector*********************************************/
    binarize_inputs(Letter_lsb[pattern_counter],
Letter_msb[pattern_counter]);
/*******Check output vector and calculate Hamming Distance***************/
    generate_layer1_neurons();
    generate_layer2_neurons();
    generate_output_neurons();
    gotoxy(31,4);
    clreol();
    gotoxy(31,4);
    printf("%d",pattern_counter);
    HammDist = hamming_distance(pattern_counter);
    if (HammDist > CRITERIA)
      singlets1(pattern_counter);
    if (HammDist > CRITERIA)
       doublets1(pattern_counter);
    if (HammDist > CRITERIA)
      triplets1(pattern_counter);
    if (HammDist > CRITERIA)
      quadlets1(pattern_counter);
    if (HammDist > CRITERIA)
       singlets2(pattern_counter);
    if (HammDist > CRITERIA)
      doublets2(pattern_counter);
```

```
    if (HammDist > CRITERIA)
     triplets2(pattern_counter);
    if (HammDist > CRITERIA)
       quadlets2(pattern_counter);
    if (HammDist > CRITERIA)
      adapt_output_layer(pattern_counter);
if (HammDist > CRITERIA)
       pattern_learned[pattern_counter]=0;
    else
pattern_learned[pattern_counter]=1;
/*************************************************************************
Apply next pattern
*************************************************************************/
    }
}
/*************************************************************************
Store weights in network file and exit
*************************************************************************/
save_weight_layers();
}
PROGRAM : TEST3.C
WRITTEN  BY: William Robinson
DESCRIPTION: Provides user with a test menu for the optical neural network.
*************************************************************************/
/*************************************************************************
TEST SET function looks for the test output neuron.
*************************************************************************/
unsigned int test_set(int test_pattern_number)
{
unsigned int actual_vector;
unsigned int differences;
unsigned int and_vector;
    actual_vector =
    (Output_binary_value[0] * 1) +
    (Output_binary_value[1] * 2) +
    (Output_binary_value[2] * 4);
and_vector = actual_vector & Letter_test[test_pattern_number];
differences = (and_vector & 1) + ((and_vector & 2) >>1) +((and_vector & 4) >> 2);
return(differences);
}
/*************************************************************************
TEST NETWORK function test network for given Hamming Distance criteria.
This is the actual test function. The two functions following this one
are for file I/o.
*************************************************************************/
void test_network(void)
{int test_pattern;
get_weight_layers();
if (OPTICAL==1){
   MODE_0();    /* Configure interface board inside 386 computer */
   ENABLE_WEIGHTS(); /* Allows SLM to be programmed */
   WRITE_LAYER1_WEIGHTS();
   WRITE_LAYER2_WEIGHTS();
   }
```

```
clrscr();
printf("TESTING...Hamming Distance Criteria is: %d",CRITERIA);
gotoxy(1,2);
printf("Testing Network for pattern: ");
/*****Begin testing*************************************************************/
for (test_pattern=0;test_pattern<10;test_pattern++){
     gotoxy(31,2);
     clreol();
     gotoxy(31,2);
     printf("%d",test_pattern);
/*******Present input vector*****************************************************/
     binarize_inputs(Test_lsb[test_pattern],Test_msb[test_pattern]);
     if (OPTICAL==1){
       WRITE_INPUT_VECTOR(Test_lsb[test_pattern],
       Test_msb[test_pattern]);
       read_hidden1_binary();
       WRITE_HIDDEN1_BINARY();
       read_hidden2_binary();
       generate_output_neurons();
       }
     else if (SOFTWARE_VERSION==2){
       simulate_layer1_neurons();
       simulate_layer2_neurons();
       generate_output_neurons();
       }
\else{
       generate_layer1_neurons();
       generate_layer2_neurons();
       generate_output_neurons();
       }
/*******Check output vector *****************************************************/
     if(test_set(test_pattern)){
       gotoxy(5,3+test_pattern);
       printf("Pattern  %d  recognized",test_pattern);
       }
     else{
       gotoxy(5,3+test_pattern);
       printf("Pattern  %d  not recognized",test_pattern);
       }
/*****************************************************************************
     Apply next pattern
*****************************************************************************/
     }
printf("\n Hit any key to continue");
getch();
}
/*****************************************************************************
ONE TEST function test network for a single Hamming Distance criteria.
*****************************************************************************/
void one_test(void){
char num[2];
char copy[5];
char temp[12];
clrscr();
if (system_selected){
```

```
 if (get_it){
   do{
    printf("Enter Hamming Distance Training Criteria (between 0 and 3): ");
    scanf("%d",&CRITERIA);
    } while ((CRITERIA < 0) | (CRITERIA > 3));
   strcpy(temp,in_name);
   itoa(CRITERIA,num,10);
   strcpy(copy,".00");
   strcat(copy,num);
   strcat(in_name,copy);
   test_network();
   strcpy(in_name,temp);
   }
 else{
    printf("Need filename to get weights\n");
    printf("Exit to MAIN menu and select FILE menu (hit any key)\n");
    getch();
    }
 }
else{
   printf("Need to select system\n");
   printf("Exit to MAIN menu and select system parameters (hit any key)\n");

   getch();
   }
}
/*****************************************************************************
MULTIPLE TESTS function tests network for a user prompted starting and
ending consecutive Hamming Distance criterias.
*****************************************************************************/
void multiple_tests(void)
{
char num[2];
char copy[5];
char temp[12];
int start,finish;
clrscr();
if (system_selected){
  if (get_it){
    do{
     printf("Enter starting Hamming Distance Training Criteria (between 0 and 2):  ");
     scanf("%d",&start);
     } while ((start < 0) | (start > 2));
    do{
     printf("Enter ending Hamming Distance Training Criteria (between start and 3):  ");
     scanf("%d",&finish);
     } while ((finish <= start) | (finish > 3));
    for (CRITERIA=start;CRITERIA<=finish;CRITERIA++){
      strcpy(temp,in_name);
      itoa(CRITERIA,num,10);
      strcpy(copy,".00");
      strcat(copy,num);
      strcat(in_name,copy);
      test_network();
      strcpy(in_name,temp);
```

```
        }
      }
    else{
      printf("Need filename to get weights\n");
      printf("Exit to MAIN menu and select FILE menu (hit any key)\n");
      getch();
      }
   }
else{
   printf("Need to select system\n");
   printf("Exit to MAIN menu and select system parameters (hit any key)\n");
   getch();
   }
get_it=0;
}
/*************************************************************************
TEST MENU FUNCTION prompts user for training options.
*************************************************************************/
void test_menu(void)
{
int  test_choice;
/*************************************************************************
Display the ANN Menu
*************************************************************************/
do {
     clrscr();
     printf("\n");
     printf ("\t\tOptical ANN Test Menu\n");
     printf (" \n");
     printf ("\t 0. Return to main menu\n");
     printf ("\t 1. Single test\n");
     printf ("\t 2. Multiple tests\n");
     printf ("\t->");
     scanf("%d",&test_choice);
     switch (test_choice) {
               case 0: {
                         get_it=0;
                         return;
                         }

              case 1: one_test();
                         break;
               case 2: multiple_tests();
                         break;
              default:break;
             }
   } while (test_choice != 0);
}
/*************************************************************************************
PROGRAM: DRIVER3.C
WRITTEN BY: William Robinson
DESCRIPTION: The following routines provide hardware I/O interface to
optical ANN and host computer. I/O is performed through a 96 bit DIGIO-96
computer interface board located at base address HEX 300 utilizing an
in-house engineered break-out board, and by a 48 channel analog to digital
```

```
convertion board at base address HEX 320. This board reads 32 optical analog
outputs corresponding to 16 optical outputs for layer 1 and 16 optical
outputs for layer 2.
*****************************************************************************/
/*****************************************************************************/
void ADC_LAYER1(void)
{
unsigned int channel;
int low_byte,high_byte,adc_counts;
unsigned char check_EOC;
double analog;

/*****Read layer 1 channels*****/
for (channel = 0; channel < 16; channel++){
   outportb(SET_CHANNEL,channel);
   outportb(ADC_CONVERT,TRIGGER);
   do
    {
    check_EOC=inportb(EOC);
    printf("EOC = %u \n",check_EOC);
    }
   while (check_EOC > 127);
   low_byte=inportb(READ_LOW);
   high_byte=inportb(READ_HIGH);
   adc_counts=((high_byte*16) + (low_byte/16));  /*TOTAL ADC COUNTS */
   Hidden1_analog_value[channel] = adc_counts * .00122;
  }
}
/*****************************************************************************/
void ADC_LAYER2(void)
{
unsigned int channel;
int low_byte,high_byte,adc_counts;
unsigned char check_EOC;
double analog;
    /*****Read layer 2 channels*****/
for (channel = 16; channel < 32; channel++){
   outportb(SET_CHANNEL,channel);
   outportb(ADC_CONVERT,TRIGGER);
   do
    {
    check_EOC=inportb(EOC);
    printf("EOC = %u \n",check_EOC);
    }
   while (check_EOC > 127);
   low_byte=inportb(READ_LOW);
   high_byte=inportb(READ_HIGH);
   adc_counts=((high_byte*16) + (low_byte/16));  /*TOTAL ADC COUNTS */
   Hidden2_analog_value[channel-16] = adc_counts * .00122;
   }
```

```
}
/*************************************************************************
Setup Digital IO Interface Board in computer, board base address is Hex 300.
*************************************************************************/
void MODE_0(void)
{
  outportb(BASE+3,128);      /* Mode 0 Output */
  outportb(BASE+7,128);      /* Mode 0 Output */
  outportb(BASE+11,155);     /* Mode 0 Input */
  outportb(BASE+15,137);     /* Mode 0 Input/Output */
}
/*************************************************************************
Writes to sixteen cascaded HUGHES 4038 32-pixel SLM waveform generator chips.
Actually generates pixel "ON" phases which get either amplified or attenuated
by programmable gain amplifiers.
*************************************************************************/
void ENABLE_WEIGHTS(void)
{
 int i;
 char j;
 outportb(BASE + 5,0); /* Turn on SLM pixels */
 for (i=0;i<=255;i++)
   {
   outportb(BASE + 4,LCD1_CLOCK + LCD2_CLOCK);  /* Clock it in */
   delay(1);
   outportb(BASE + 4,0);
   delay(1);
   }
  /* Load data into chip registers */
 outportb(BASE + 5,LCD1_LOAD + LCD2_LOAD);
}
/*************************************************************************
Write the input vector read from training file in MR 2 routine.
*************************************************************************/
void WRITE_INPUT_VECTOR(unsigned char lsb,unsigned char msb)
{
 unsigned char tmp[8];
/* Input LED's are reversed in order, swap order ***********************
     bit0 --- bit7
     bit1 --- bit6
     bit2 --- bit5
     bit3 --- bit4
     bit4 --- bit3
     bit5 --- bit2
     bit6 --- bit1
     bit7 --- bit0
*************************************************************************/
   tmp[0] = (msb&64)>>6;
   tmp[1] = (msb&32)>>4;
   tmp[2] = (msb&16)>>2;
```

```
    tmp[3] = (msb&8);
    tmp[4] = (msb&4)<<2;
    tmp[5] = (msb&2)<<4;
    tmp[6] = (msb&1)<<6;
    tmp[7] = 1;          /* Bias neuron */

    msb = tmp[0]|tmp[1]|tmp[2]|tmp[3]|tmp[4]|tmp[5]|tmp[6]|tmp[7];

    tmp[0] = (lsb&128)>>7;
    tmp[1] = (lsb&64)>>5;
    tmp[2] = (lsb&32)>>3;
    tmp[3] = (lsb&16)>>1;
    tmp[4] = (lsb&8)<<1;
    tmp[5] = (lsb&4)<<3;
    tmp[6] = (lsb&2)<<5;
    tmp[7] = (lsb&1)<<7;

    lsb = tmp[0]|tmp[1]|tmp[2]|tmp[3]|tmp[4]|tmp[5]|tmp[6]|tmp[7];

   outportb(ILED_LO,lsb);
   outportb(ILED_HI,msb);
}
/*****************************************************************************
Write the output from hidden layer as input to output layer only when using
all hardware thresholding.
*****************************************************************************/
void WRITE_HIDDEN1_BINARY(void)
{
  unsigned char tmp[8],msb,lsb;
/*****************************************************************************
        Read the hidden layer outputs
*****************************************************************************/
   msb = inportb(HOUT_HI);
   lsb = inportb(HOUT_LO);
/* Input LED's are reversed in order, swap order ************************
      bit0 --- bit7
      bit1 --- bit6
      bit2 --- bit5
      bit3 --- bit4
      bit4 --- bit3
      bit5 --- bit2
      bit6 --- bit1
      bit7 --- bit0
*****************************************************************************/
    tmp[0] = (msb&64)>>6;
    tmp[1] = (msb&32)>>4;
    tmp[2] = (msb&16)>>2;
    tmp[3] = (msb&8);
    tmp[4] = (msb&4)<<2;
    tmp[5] = (msb&2)<<4;
    tmp[6] = (msb&1)<<6;
```

```
  tmp[7] = 1;          /* Bias neuron */

  msb = tmp[0]|tmp[1]|tmp[2]|tmp[3]|tmp[4]|tmp[5]|tmp[6]|tmp[7];
  tmp[0] = (lsb&128)>>7;
  tmp[1] = (lsb&64)>>5;
  tmp[2] = (lsb&32)>>3;
  tmp[3] = (lsb&16)>>1;
  tmp[4] = (lsb&8)<<1;
  tmp[5] = (lsb&4)<<3;
  tmp[6] = (lsb&2)<<5;
  tmp[7] = (lsb&1)<<7;
  lsb = tmp[0]|tmp[1]|tmp[2]|tmp[3]|tmp[4]|tmp[5]|tmp[6]|tmp[7];

 outportb(HLED_LO,msb);
 outportb(HLED_HI,lsb);
}
/*************************************************************************
Write the output from hidden layer as input to output layer when using
ADC board.
*************************************************************************/
void PRESENT_HIDDEN1_VECTOR(void)
{
  int i;
  unsigned char tmp[8],msb,lsb;
  for (i=0;i<8;i++){
     if (Hidden1_binary_value[i])
      tmp[i]=1;
     else
      tmp[i]=0;
     tmp[i]=(tmp[i] << i);
     }
   tmp[7] = 1;          /* Bias neuron */
   lsb = tmp[0]|tmp[1]|tmp[2]|tmp[3]|tmp[4]|tmp[5]|tmp[6]|tmp[7];
  for (i=0;i<8;i++){
     if (Hidden1_binary_value[i+8])
      tmp[i]=1;
     else
      tmp[i]=0;
     tmp[i]=(tmp[i] << i);
     }
  tmp[7] = 1;          /* Bias neuron */
  msb = tmp[0]|tmp[1]|tmp[2]|tmp[3]|tmp[4]|tmp[5]|tmp[6]|tmp[7];
  outportb(HLED_LO,lsb);
  outportb(HLED_HI,msb);
}
/*************************************************************************
This routine writes the values of weights found in WEIGHT_LAYER1[16][16]
to the corresponding hardware programmable gain amplifier. The programmable
gain amplifiers provides a gray level driving signal to the weight mask.
*************************************************************************/
```

```
void WRITE_LAYER1_WEIGHTS(void)
{
    int  i,j,k,n;
    unsigned char temp,a,b,
     BOARD1[32],          /* Board 1 registers */
     BOARD2[32],          /* Board 2 registers */
     BOARD3[32],          /* Board 3 registers */
     BOARD4[32],          /* Board 4 registers */
     BOARD5[32],          /* Board 5 registers */
     BOARD6[32],          /* Board 6 registers */
     BOARD7[32],          /* Board 7 registers */
     BOARD8[32],          /* Board 8 registers */
     OUTWEIGHTS1[256],    /* Output array for writing to SLM1 */
     D0 = 1,              /* Typecasting numbers for bit shifting */
     D1 = 2,
     D2 = 4,
     D3 = 8,
     D4 = 16,
     D5 = 32,
     D6 = 64,
     D7 = 128;
/*************************************************************************
Convert the weights and biases into usable forms for the  optical weights
*************************************************************************/
  for (i = 0; i < 15; i++)
     for (j = 0; j < 15; j++) {
       if (Weight_layer1[i][j] < -60)
          Optical_layer1[i][j] = 150;
       else if(Weight_layer1[i][j] > 60)
          Optical_layer1[i][j] = 30;
       else
          Optical_layer1[i][j] = (int)(-Weight_layer1[i][j])+90;
       }
/*************************************************************************
 Mid transmission for column 16, will be used for active thresholding.
*************************************************************************/
    for (j = 0; j < 15; j++)
       Optical_layer1[15][j] = 90;
/*************************************************************************
        Assign correspoding weight to sim driver board
*************************************************************************/
  for (i=0,n=0;i<=3;i++)
    for (j=0;j<=7;j++,n++)
       BOARD1[n]= (unsigned char)Optical_layer1[i][j];
 for (i=4,n=0;i<=7;i++)
    for (j=0;j<=7;j++,n++)
       BOARD2[n]= (unsigned char)Optical_layer1[i][j];
 for (i=8,n=0;i<=11;i++)
    for (j=0;j<=7;j++,n++)
     BOARD3[n]= (unsigned char)Optical_layer1[i][j];
 for (i=12,n=0;i<=15;i++)
```

```
  for (j=0;j<=7;j++,n++)
      BOARD4[n]= (unsigned char)Optical_layer1[i][j];
 for (i=0,n=0;i<=3;i++)
    for (j=0;j<=7;j++,n++)
      BOARD5[n]= (unsigned char)Optical_layer1[i][15-j];
 for (i=4,n=0;i<=7;i++)
    for (j=0;j<=7;j++,n++)
      BOARD6[n]= (unsigned char)Optical_layer1[i][15-j];
 for (i=8,n=0;i<=11;i++)
    for (j=0;j<=7;j++,n++)
      BOARD7[n]= (unsigned char)Optical_layer1[i][15-j];
 for (i=12,n=0;i<=15;i++)
    for (j=0;j<=7;j++,n++)
      BOARD8[n]= (unsigned char)Optical_layer1[i][15-j];
/**Fix order error for SLM1 **/
 for (n=8;n<=15;n++)
    {
    temp=BOARD1[n]; BOARD1[n]=BOARD1[n+1]; BOARD1[n+1]=temp;
    temp=BOARD2[n]; BOARD2[n]=BOARD2[n+1]; BOARD2[n+1]=temp;
    temp=BOARD3[n]; BOARD3[n]=BOARD3[n+1]; BOARD3[n+1]=temp;
    temp=BOARD4[n]; BOARD4[n]=BOARD4[n+1]; BOARD4[n+1]=temp;
    temp=BOARD5[n]; BOARD5[n]=BOARD5[n+1]; BOARD5[n+1]=temp;
    temp=BOARD6[n]; BOARD6[n]=BOARD6[n+1]; BOARD6[n+1]=temp;
    temp=BOARD7[n]; BOARD7[n]=BOARD7[n+1]; BOARD7[n+1]=temp;
   temp=BOARD8[n]; BOARD8[n]=BOARD8[n+1]; BOARD8[n+1]=temp;
    n++;
    }
for (n=24;n<=31;n++)
    {
    temp=BOARD1[n]; BOARD1[n]=BOARD1[n+1]; BOARD1[n+1]=temp;
    temp=BOARD2[n]; BOARD2[n]=BOARD2[n+1]; BOARD2[n+1]=temp;
    temp=BOARD3[n]; BOARD3[n]=BOARD3[n+1]; BOARD3[n+1]=temp;
    temp=BOARD4[n]; BOARD4[n]=BOARD4[n+1]; BOARD4[n+1]=temp;
    temp=BOARD5[n]; BOARD5[n]=BOARD5[n+1]; BOARD5[n+1]=temp;
    temp=BOARD6[n]; BOARD6[n]=BOARD6[n+1]; BOARD6[n+1]=temp;
    temp=BOARD7[n]; BOARD7[n]=BOARD7[n+1]; BOARD7[n+1]=temp;
    temp=BOARD8[n]; BOARD8[n]=BOARD8[n+1]; BOARD8[n+1]=temp;
    n++;
    }
  /** Now set up ordinal serial sequence and concatenate all 8 **/
  /** board data lines to be clocked out **/
/*************************************************************************
 The following defines driver board data line in output port of interface
 BOARD #__BIT POSITION___BIT
  BOARD1     0          1
  BOARD2     1          2
  BOARD3     2          4
  BOARD4     3          8
  BOARD5     4          16
```

```
BOARD6    5       32
BOARD7    6       64
BOARD8    7       128
**********************************************************************/
for (n=0,l=31;n<=255;n++,l--)
  {
  OUTWEIGHTS1[n] = ((BOARD1[l]&D7)>>7) | ((BOARD2[l]&D7)>>6)
        | ((BOARD3[l]&D7)>>5) | ((BOARD4[l]&D7)>>4)
        | ((BOARD5[l]&D7)>>3) | ((BOARD6[l]&D7)>>2)
        | ((BOARD7[l]&D7)>>1) | (BOARD8[l]&D7);
  n++;
  OUTWEIGHTS1[n] = ((BOARD1[l]&D6)>>6) | ((BOARD2[l]&D6)>>5)
        | ((BOARD3[l]&D6)>>4) | ((BOARD4[l]&D6)>>3)
        | ((BOARD5[l]&D6)>>2) | ((BOARD6[l]&D6)>>1)
        | (BOARD7[l]&D6) | ((BOARD8[l]&D6)<<1);
  n++;
  OUTWEIGHTS1[n] = ((BOARD1[l]&D5)>>5) | ((BOARD2[l]&D5)>>4)
        | ((BOARD3[l]&D5)>>3) | ((BOARD4[l]&D5)>>2)
        | ((BOARD5[l]&D5)>>1) | (BOARD6[l]&D5)
        | ((BOARD7[l]&D5)<<1) | ((BOARD8[l]&D5)<<2);
  n++;
  OUTWEIGHTS1[n] = ((BOARD1[l]&D4)>>4) | ((BOARD2[l]&D4)>>3)
        | ((BOARD3[l]&D4)>>2) | ((BOARD4[l]&D4)>>1
        | (BOARD5[l]&D4) | ((BOARD6[l]&D4)<<1)
        | (BOARD7[l]&D4)<<2) | ((BOARD8[l]&D4)<<3);
  n++;
  OUTWEIGHTS1[n] = ((BOARD1[l]&D3)>>3) | ((BOARD2[l]&D3)>>2)
        | ((BOARD3[l]&D3)>>1) | (BOARD4[l]&D3)
        | ((BOARD5[l]&D3)<<1) | ((BOARD6[l]&D3)<<2)
        | ((BOARD7[l]&D3)<<3) | ((BOARD8[l]&D3)<<4);
  n++;
  OUTWEIGHTS1[n] = ((BOARD1[l]&D2)>>2) | ((BOARD2[l]&D2)>>1)
        | (BOARD3[l]&D2) | ((BOARD4[l]&D2)<<1)
        | ((BOARD5[l]&D2)<<2) | ((BOARD6[l]&D2)<<3)
        | ((BOARD7[l]&D2)<<4) | ((BOARD8[l]&D2)<<5);
  n++;
  OUTWEIGHTS1[n] = ((BOARD1[l]&D1)>>1) | (BOARD2[l]&D1)
        | ((BOARD3[l]&D1)<<1) | ((BOARD4[l]&D1)<<2)
        | ((BOARD5[l]&D1)<<3) | ((BOARD6[l]&D1)<<4)
        | ((BOARD7[l]&D1)<<5) | ((BOARD8[l]&D1)<<6);
  n++;
  OUTWEIGHTS1[n] = (BOARD1[l]&D0) | ((BOARD2[l]&D0)<<1)
        | ((BOARD3[l]&D0)<<2) | ((BOARD4[l]&D0)<<3)
```

```
            | ((BOARD5[I]&D0)<<4) | ((BOARD6[I]&D0)<<5)
            | ((BOARD7[I]&D0)<<6) | ((BOARD8[I]&D0)<<7);
    }
 /** Now clock out the weights serially **/
 /** First set up load lines high so data can be clocked in **/
 outportb(BASE + 5,LAYER1_LOAD);
 for (I=0,J=0;J<=15;J++)
    {
    /* First clock out SSBIT */
    outportb(BASE + 4,0);
    outportb(BASE + 12,SSBIT);
    outportb(BASE + 4,LAYER1_CLOCK);
    /* Now clock out next 16 bits in the following order */
    /* MSB1.....LSB1 MSB0.....LSB0 */
    for (k=0;k<=15;k++)
      {
      outportb(BASE + 4,0);
      outportb(BASE +12,OUTWEIGHTS1[I]);
      outportb(BASE + 4,LAYER1_CLOCK);
      I++;
      }
    }
 /** Now load data in programmable potentiometer registers **/
 outportb(BASE + 5, 0);
}
/*************************************************************************/
/*************************************************************************
This routine writes the values of weights found in WEIGHT_LAYER1[16][16]
to the corresponding hardware programmable gain amplifier. The programmable
gain amplifiers provides a gray level driving signal to the weight mask.
*************************************************************************/
void WRITE_LAYER2_WEIGHTS(void)
{
  int I,J,k,n;
  unsigned char
    temp,a,b,
    BOARD9[32],          /* Board 9 registers */
    BOARD10[32],         /* Board 10 registers */
    BOARD11[32],         /* Board 11 registers */
    BOARD12[32],         /* Board 12 registers */
    BOARD13[32],         /* Board 13 registers */
    BOARD14[32],         /* Board 14 registers */
    BOARD15[32],         /* Board 15 registers */
```

```
  BOARD16[32],          /* Board 16 registers */
   OUTWEIGHTS2[256],    /* Output array for writing to SLM2 */
   D0 = 1,              /* Typecasting numbers for bit shifting */
   D1 = 2,
   D2 = 4,
   D3 = 8,
   D4 = 16,
   D5 = 32,
   D6 = 64,
   D7 = 128;
/*************************************************************************
Convert the weights and biases into usable forms for the optical weights
*************************************************************************/
  for (i = 0; i < 15; i++)
    for (j = 0; j < 15; j++) {
      if (Weight_layer2[i][j] < -60)
Optical_layer2[i][j] = 150;
      else if(Weight_layer2[i][j] > 60)
        Optical_layer2[i][j] = 30;
      else
        Optical_layer2[i][j] = (int)(-Weight_layer2[i][j])+90;
      }
/*************************************************************************
Mid transmission for column 16, will be used for active thresholding.
*************************************************************************/
    for (j = 0; j < 15; j++)
      Optical_layer2[15][j] = 90;
/*************************************************************************
       Assign correspoding weight to slm driver board
*************************************************************************/
  for (i=0,n=0;i<=3;i++)
    for (j=0;j<=7;j++,n++)
      BOARD9[n]= (unsigned char)Optical_layer2[i][j];
  for (i=4,n=0;i<=7;i++)
    for (j=0;j<=7;j++,n++)
      BOARD10[n]= (unsigned char)Optical_layer2[i][j];
 for (i=8,n=0;i<=11;i++)
   for (j=0;j<=7;j++,n++)
      BOARD11[n]= (unsigned char)Optical_layer2[i][j];
 for (i=12,n=0;i<=15;i++)
   for (j=0;j<=7;j++,n++)
      BOARD12[n]= (unsigned char)Optical_layer2[i][j];
 for (i=0,n=0;i<=3;i++)
```

```
    for (j=0;j<=7;j++,n++)
      BOARD13[n]= (unsigned char)Optical_layer2[i][15-j];
  for (i=4,n=0;i<=7;i++)
    for (j=0;j<=7;j++,n++)
      BOARD14[n]= (unsigned char)Optical_layer2[i][15-j];
  for (i=8,n=0;i<=11;i++)
    for (j=0;j<=7;j++,n++)
      BOARD15[n]= (unsigned char)Optical_layer2[i][15-j];
  for (i=12,n=0;i<=15;i++)
    for (j=0;j<=7;j++,n++)
      BOARD16[n]= (unsigned char)Optical_layer2[i][15-j];
/**Fix order error for SLM2 **/
  for (n=8;n<=15;n++)
     {
     temp=BOARD9[n]; BOARD9[n]=BOARD9[n+1]; BOARD9[n+1]=temp;
     temp=BOARD10[n]; BOARD10[n]=BOARD10[n+1]; BOARD10[n+1]=temp;
     temp=BOARD11[n]; BOARD11[n]=BOARD11[n+1]; BOARD11[n+1]=temp;
     temp=BOARD12[n]; BOARD12[n]=BOARD12[n+1]; BOARD12[n+1]=temp;
     temp=BOARD13[n]; BOARD13[n]=BOARD13[n+1]; BOARD13[n+1]=temp;
     temp=BOARD14[n]; BOARD14[n]=BOARD14[n+1]; BOARD14[n+1]=temp;
     temp=BOARD15[n]; BOARD15[n]=BOARD15[n+1]; BOARD15[n+1]=temp;
     temp=BOARD16[n]; BOARD16[n]=BOARD16[n+1]; BOARD16[n+1]=temp;
     n++;
     }
  for (n=24;n<=31;n++)
     {
     temp=BOARD9[n]; BOARD9[n]=BOARD9[n+1]; BOARD9[n+1]=temp;
     temp=BOARD10[n]; BOARD10[n]=BOARD10[n+1]; BOARD10[n+1]=temp;
     temp=BOARD11[n]; BOARD11[n]=BOARD11[n+1]; BOARD11[n+1]=temp;
     temp=BOARD12[n]; BOARD12[n]=BOARD12[n+1]; BOARD12[n+1]=temp;
     temp=BOARD13[n]; BOARD13[n]=BOARD13[n+1]; BOARD13[n+1]=temp;
     temp=BOARD14[n]; BOARD14[n]=BOARD14[n+1]; BOARD14[n+1]=temp;
     temp=BOARD15[n]; BOARD15[n]=BOARD15[n+1]; BOARD15[n+1]=temp;
     temp=BOARD16[n]; BOARD16[n]=BOARD16[n+1]; BOARD16[n+1]=temp;
     n++;
     }
  /** Now set up ordinal serial sequence and concatenate all 8 **/
  /** board data lines to be clocked out **/
/*************************************************************************
 The following defines driver board data line in output port of interface
___BOARD #__BIT POSITION___BIT VALUE_____________________________________
  BOARD8     7         128
  BOARD9     0          1
```

```
BOARD10    1        2
BOARD11    2        4
BOARD12    3        8
BOARD13    4        16
BOARD14    5        32
BOARD15    6        64
BOARD16    7        128
**************************************************************************/
for (n=0,l=31;n<=255;n++,l--)
   {
   OUTWEIGHTS2[n] = ((BOARD9[l]&D7)>>7) | ((BOARD10[l]&D7)>>6)
          | ((BOARD11[l]&D7)>>5) | ((BOARD12[l]&D7)>>4)
          | ((BOARD13[l]&D7)>>3) | ((BOARD14[l]&D7)>>2)
          | ((BOARD15[l]&D7)>>1) | (BOARD16[l]&D7);
   n++;
   OUTWEIGHTS2[n] = ((BOARD9[l]&D6)>>6) | ((BOARD10[l]&D6)>>5)
          | ((BOARD11[l]&D6)>>4) | ((BOARD12[l]&D6)>>3)
          | ((BOARD13[l]&D6)>>2) | ((BOARD14[l]&D6)>>1)
          | (BOARD15[l]&D6) | ((BOARD16[l]&D6)<<1);
   n++;
   OUTWEIGHTS2[n] = ((BOARD9[l]&D5)>>5) | ((BOARD10[l]&D5)>>4)
          | ((BOARD11[l]&D5)>>3) | ((BOARD12[l]&D5)>>2)
          | ((BOARD13[l]&D5)>>1) | (BOARD14[l]&D5)
          | ((BOARD15[l]&D5)<<1) | ((BOARD16[l]&D5)<<2);
   n++;
   OUTWEIGHTS2[n] = ((BOARD9[l]&D4)>>4) | ((BOARD10[l]&D4)>>3)
          | ((BOARD11[l]&D4)>>2) | ((BOARD12[l]&D4)>>1)
          | (BOARD13[l]&D4) | ((BOARD14[l]&D4)<<1)
          | ((BOARD15[l]&D4)<<2) | ((BOARD16[l]&D4)<<3);
   n++;
   OUTWEIGHTS2[n] = ((BOARD9[l]&D3)>>3) | ((BOARD10[l]&D3)>>2)
          | ((BOARD11[l]&D3)>>1) | (BOARD12[l]&D3)
          | ((BOARD13[l]&D3)<<1) | ((BOARD14[l]&D3)<<2)
          | ((BOARD15[l]&D3)<<3) | ((BOARD16[l]&D3)<<4);
   n++;
   OUTWEIGHTS2[n] = ((BOARD9[l]&D2)>>2) | ((BOARD10[l]&D2)>>1)
          | (BOARD11[l]&D2) | ((BOARD12[l]&D2)<<1)
          | ((BOARD13[l]&D2)<<2) | ((BOARD14[l]&D2)<<3)
          | ((BOARD15[l]&D2)<<4) | ((BOARD16[l]&D2)<<5);
   n++;
   OUTWEIGHTS2[n] = ((BOARD9[l]&D1)>>1) | (BOARD10[l]&D1)
          | ((BOARD11[l]&D1)<<1) | ((BOARD12[l]&D1)<<2)
          | ((BOARD13[l]&D1)<<3) | ((BOARD14[l]&D1)<<4)
          | ((BOARD15[l]&D1)<<5) | ((BOARD16[l]&D1)<<6);
```

```
    n++;
    OUTWEIGHTS2[n] = (BOARD9[I]&D0) | ((BOARD10[I]&D0)<<1)
            | ((BOARD11[I]&D0)<<2) | ((BOARD12[I]&D0)<<3)
            | ((BOARD13[I]&D0)<<4) | ((BOARD14[I]&D0)<<5)
            | ((BOARD15[I]&D0)<<6) | ((BOARD16[I]&D0)<<7);
    }
 /** Now clock out the weights serially **/
/** First set up load lines high so data can be clocked in **/
 outportb(BASE + 5,LAYER2_LOAD);
 for (I=0,J=0;J<=15;J++)
    {
    /* First clock out SSBIT */
    outportb(BASE + 4,0);
    outportb(BASE + 13,SSBIT);
    outportb(BASE + 4,LAYER2_CLOCK);
    /* Now clock out next 16 bits in the following order */
    /* MSB1.....LSB1 MSB0.....LSB0 */
    for (k=0;k<=15;k++)
      {
      outportb(BASE + 4,0);
      outportb(BASE +13,OUTWEIGHTS2[I]);
      outportb(BASE + 4,LAYER2_CLOCK);
      I++;
      }
    }
  /** Now load data in programmable potentiometer registers **/
  outportb(BASE + 5, 0);
}
/*************************************************************************/
void LEDS_OFF (void)
{
    outportb (ILED_LO,0x00);
    outportb (ILED_HI,0x00);
    outportb (HLED_LO,0x00);
    outportb (HLED_HI,0x00);
}
/*************************************************************************
PROGRAM : FILE.C
WRITTEN  BY: William Robinson
DESCRIPTION: Provides simple file I/O menu.
*************************************************************************/
/*************************************************************************
SAVE FILE FUNCTION prompts user for network output file base name.
```

```
*****************************************************************************/
void save_file(void)
{
printf("\n Enter name of output file: ");
scanf("%12s", out_name);
save_it=1;
}
/*****************************************************************************
GET FILE FUNCTION prompts user for network output file base name.
*****************************************************************************/
void get_file(void)
{
printf("\n Enter name of input file: ");
scanf("%12s", in_name);
get_it=1;
}
/*****************************************************************************
FILE MENU FUNCTION prompts user for network files used for input and/or
output.
*****************************************************************************/
void file_menu(void)
{
int  file_choice;
/*****************************************************************************
        Display the ANN Menu
*****************************************************************************/
do {

    clrscr();
    printf("\n");
    printf ("\t\tOptical ANN File Menu\n");
    printf (" \n");
    printf ("\t 0. Return to main menu\n");
    printf ("\t 1. Save weights filename\n");
    printf ("\t 2. Get weights file for testing\n");
    printf ("\t 3. Get pattern file(s) for training\n");
    printf ("\t->");
    scanf("%d",&file_choice);
    switch (file_choice) {
        case 0: return;
        case 1: save_file();
    break;
        case 2: get_file();
```

```
    break;
        case 3: printf("Option is not available at this time\n");
    break;
        default:break;
    }*****************************************************************************
PROGRAM : TRAIN.C
WRITTEN  BY: William Robinson
DESCRIPTION: Provides user with a training menu.
*****************************************************************************/
/*****************************************************************************
ONE RUN function trains network for a single Hamming Distance criteria.
*****************************************************************************/
void one_run(void)
{
char num[2];
char copy[5];
char temp[12];
clrscr();
if (system_selected){
  if (save_it){
    do{
    printf("Enter Hamming Distance Training Criteria (between 0 and 4 inclusive):  ");
    scanf("%d",&CRITERIA);
    } while ((CRITERIA < 0) | (CRITERIA > 4));
    strcpy(temp,out_name);
    itoa(CRITERIA,num,10);
    strcpy(copy,".00");
    strcat(copy,num);
    strcat(out_name,copy);
    if (OPTICAL) optical_ann();
    else if(SOFTWARE_VERSION==2) simulate();
    else normal_training();
    strcpy(out_name,temp);
    }
  else{
    printf("Need filename to save weights\n");
    printf("Exit to MAIN menu and select FILE menu (hit any key)\n");
    getch();
     }
  }
 else{
   printf("Need to select system\n");
   printf("Exit to MAIN menu and select system parameters (hit any key)\n");
```

```
  getch();
    }
}
/*****************************************************************************
MULTIPLE RUN function trains network for a user prompted starting and

ending consecutive Hamming Distance criterias.
*****************************************************************************/
void multiple_runs(void)
{
char num[2];
char copy[5];
char temp[12];
int start,finish;
clrscr();
if (system_selected){
  if (save_it){
    do{
     printf("Enter starting Hamming Distance Training Criteria (between 0 and 3 inclusive): ");
     scanf("%d",&start);
     } while ((start < 0) | (start > 3));
    do{
     printf("Enter ending Hamming Distance Training Criteria (between start and 4 inclusive): ");
     scanf("%d",&finish);
     } while ((finish <= start) | (finish > 4));
    for (CRITERIA=start;CRITERIA<=finish;CRITERIA++){
      strcpy(temp,out_name);
      itoa(CRITERIA,num,10);
      strcpy(copy,".00");
      strcat(copy,num);
      strcat(out_name,copy);
      if (OPTICAL) optical_ann();
      else if(SOFTWARE_VERSION==2) simulate();
      else normal_training();
      strcpy(out_name,temp);
      }
    }
  else{
    printf("Need filename to save weights\n");
    printf("Exit to MAIN menu and select FILE menu (hit any key)\n");
    getch();
    }
  }
 else{
```

```
  printf("Need to select system\n");
  printf("Exit to MAIN menu and select system parameters (hit any key)\n");
  getch();
   }
save_it=0;
}
/*****************************************************************************
TRAIN MENU FUNCTION prompts user for training options.
*****************************************************************************/
void train_menu(void)
{
int  train_choice;
/*****************************************************************************
        Display the ANN Menu
*****************************************************************************/
do {
     clrscr();
     printf("\n");
     printf ("\t\tOptical ANN Train Menu\n");
     printf (" \n");
     printf ("\t 0. Return to main menu\n");
     printf ("\t 1. Single run\n");
     printf ("\t 2. Multiple runs\n");
     printf ("\t->");
     scanf("%d",&train_choice);
     switch (train_choice) {
          case 0: {
     save_it=0;
     return;
     }
          case 1: one_run();
     break;
          case 2: multiple_runs();
     break;
          default:break;
     }
   } while (train_choice != 0);
}
/*****************************************************************************
PROGRAM: uniform.c
WRITTEN BY: Danny Shelton
DESCRIPTION: u32 generates a 32 bit random number based Seed1 and Seed2
parameters located in "include3.c". This function is called by uniform_dist()
```

```
which takes two numbers and generates a random number between these two
numbers. The resulting numbers are used to initialize the weight arrays
in the MR2 training algorithm.
*****************************************************************************/
double u32(void) {
    /* 32 bit */
    long int z,k;
    k = Seed1/53668;
    Seed1 = 40014*(Seed1-k*53668)-k*12211;
    if (Seed1<0) Seed1 += 2147483563;
    k = Seed2/52774;
    Seed2 = 40692*(Seed2-k*52774)-k*3791;
    if (Seed2<0) Seed2 += 2147483399;
    z = Seed1-Seed2;
    if (z<1) z += 2147483562;
    return (z*4.665613e-10);
}
double uniform_dist(double a, double b)
{
    Seed1 = rand();
    Seed2 = rand();
    return(a+(b-a)*u32());
}
/*****************************************************************************
FILE: RANK.C
Borrowed from Numerical Recipes in C (see References)
DESCRIPTION: Ranks an array of N values in ascending order. Used in
Madaline Rule 2 Training.
*****************************************************************************/
/*****************************************************************************
INDEXX function produces an array pointers which indexes the values passed
in ascending order.
*****************************************************************************/
void Indexx(int n,double arrin[],int indx[])
{
    int l,j,ir,indxt,i;
    float q;
    for (j=1; j <= n; j++)
        indx[j]=j;
    l=(n >> 1) + 1;
    ir=n;
    for ( ; ; ) {
      if ( l > 1)
```

```
            q=arrin[(indxt=indx[--i])];
          else {
            q=arrin[(indxt=indx[ir])];
            indx[ir]=indx[1];
            if (--ir == 1){
              indx[1]=indxt;
              return;
        }
          }
          i=j;
          j=j << 1;
          while (j <=ir) {
            if (j < ir && arrin[indx[j]] < arrin[indx[j+1]]) j++;
            if (q < arrin[indx[j]]) {
              indx[i]=indx[j];
              j += (i=j);
            }
            else j=ir + 1;
          }
          indx[i]=indxt;
        }
}
/*************************************************************************
RANK function takes the index array and converts it to a ordered rank array.
*************************************************************************/
void Rank(int n,int indx[],int irank[])
{
    int j;
    for (j=1;j <= n;j++) irank[indx[j]]=j;
```

Appendix 5

Test data

File: Simul.000

Hamming Distance: 0

Patterns shown: 96

Number recognized: 10

Number not recognized: 0

Percent recognized: 100

File: Simul.001

Hamming Distance: 1

Patterns shown: 96

Number recognized: 9

Number not recognized: 1

Percent recognized: 90

File: Simul.002

Hamming Distance: 2

Patterns shown: 96

Number recognized: 7

Number not recognized: 3

Percent recognized: 70

File: Simul.003

Hamming Distance: 3

Patterns shown: 96

Number recognized: 5

Number not recognized: 5

Percent recognized: 50

File: Optic.000

Hamming Distance: 0

Patterns shown: 144

Number recognized: 10

Number not recognized: 0

Percent recognized: 90

File: Optic.001

Hamming Distance: 1

Patterns shown: 144

Number recognized: 7

Number not recognized: 3

Percent recognized: 70

File: Optic.002

Hamming Distance: 2

Patterns shown: 144

Number recognized: 6

Number not recognized: 4

Percent recognized: 60

File: Optic.003

Hamming Distance: 3

Patterns shown: 96

Number recognized: 3

Number not recognized: 7

Percent recognized: 30

References

1. M. McCord Nelson, and W.T. Illingworth. "Neural Nets". Addison Wesley, 1991.

2. DARPA. Neural Network Study.AFCEA International Press, 1988

3. D. Stubbs. "Neurocomputers". MD Computing, vol. 5, no. 3, 1988, 14-25.

4. R. Lippmann. "An Introduction to Computing with Neural Nets". IEEE ASSPMagazine, 4-22, April1987.

5. P. Wasserman. Neural Computing : Theory and Practice. Van Nostrand Reinhold, 1989.

6. Y. Pao. Adaptive Pattern Recognition and Neural Networks. Addison-Wesley, 1989.

7. A. Lapedes and R. Farber. "How Neural Networks Work". In Evolutiuon, Learning, and Cognition. ed Y.C. LEE, World Scientific, 1988.

8. T. Geszti. Physical Models of Neural Networks. World Scientific, 1990

9. M.L. Minsky, and S.A. Papert. Perceptrons. Cambridge: MIT Press, 1988.

10. J. Hertz, A. Krogh, and R. Palmer. Introduction to the Theory of Neural Computation. Addison-Wesley Publishing Co., 115-147, 1991.

11. D.E. Rumelhart, G.E. Hinton, and R.J. Williams. "Learning Internal Representation by Error Propagation". In Parallel Distributed Processing: Explorations in the Microstructures of Cognition. Cambridge: MIT Press,1986

12.W. Jones, and J. Hoskins."Back-Propagation". Byte, October 1987, 155-162.

13. T. Maxwell. "Pattern Recognition and Single Layer Networks". In Evolution,

Learning, and Cognition. ed Y.C. Lee, World Scientific, 1988.

14. J.J. Hopfield, and D.W. Tank. "Computing with Neural Circuits: A Model". Science, vol . 233, 625-633. August 1986.

15. P. Bartlett, and T. Downs. "Using Random Weights to Train Multilayer Networks with Hard Limiting neurons". IEEE Transactions on Neural Networks. vol. 3, no. 2, 202-210, March 1992.

16. B. Widrow, G. Winter, and R. Baxter."Layered Neural Nets for Pattern Recognition".IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 36, no. 7,1109-1118, July 1988.

17. A. Murray."Multilayer Perceptron Learning Optimized for On-Chip Implementation: A Noise Robust System". Neural Computation, vol. 4, 366-381, 1992.

18. E. Baum, D. Haussler. "What Size Net Gives Valid Generalization". Neural Computation. vol. 1, 151-160, 1989.

19. R. frye, E. Richman, and C. Wong."Backpropagation Learning and Nonidealities in analog Neural Network Hardware". IEEE Transactions on Neural Networks, vol. 2, no. 1, 110-117, Jan 1991.

20. M. Jabri, and B. Flowers." Weight Perturbation: An Optimal Technique for Analog VLSI Feedforward and Recurrent Multilayer Neural Networks". Neural Computation, vol. 3, 546-564, 1991.

21. D. Gray, and A. Michel."A Training Algorithm for Binary Feedforward Neural Networks". IEEE Transactions on Neural Networks, vol. 3, no. 2, 176-194, March

1992.

22. M. Fukumi, and S. Omah."A New Backpropagation Algorithm with Coupled Neurons". IEEE Transactions on Neural Networks, vol. 2, no. 5, 535-538, Sept. 1991.

23. J.M. Waas, and M. Waring."Spatial Light Modulators: an User's Guide". Semetex Corporation. 1989 Design and Photonics Handbook.

24. K.M. Johnson, and G. Moddel." Motivations for Using Ferroelectric Liquid Crystal Spatial Light Modulators in Neurocomputing". Applied Optics, vol. 28, no. 22, 4888-4899, November 15, 1989.

25. F. Okumura, K. Sera, H. Asada, S. Kaneko, H. Ichinose, K. Tanaka, t. Yokoi, and C. Tani."Ferroelectric Liquid-Crystal Shutter Array with a Si:H TFT Driver". IEEE Transactions on Electron Devices, vol. 37, no. 10, 2201-2205, October 10, 1990.

26. U. Efron, P. Braatz, M. Little, R. Schwartz, and J. Grinberg." Silicon Liquid Crystal Light Valves: Status and Issues". Optical Engineering, Vol 22, no. 6, 682-686, November-December 1983.

27. C. Warde, and J. Thackara." Operating Modes of the Microchannel Spatial Light Modulator". Optical Engineering, vol. 22, no. 6, 695-703, November-December 1983.

28. J. Davis, and J. Waas." Current Status of the Magneto-Optic Spatial Light Modulator". SPIE Proceedings, vol 1150, 1989.

29. N. Collings, W. Crossland, P. Ayliffe, D. Vass, and I. Underwood." Evolutionary Development of Advanced Liquid Crystal Spatial Light Modulators". Applied Optics, vol.

28, no. 22, 4740-4747, November 15, 1989.

30.J. Florence." Optical Characteristics of Deformable Mirror Spatial Light Modulators". Texas Instruments Inc., Central Research Laboratory.

31. J. Florence, T. Lin, and W. Wu." Improved DMD Configurations for Image Correlation". Texas Instruments Inc., Central Research Laboratory.

32. L. Hornbeck." Deformable Mirror Spatial Light Modulators". SPIE Proceedings, vol. 1150, 1989.

33. D. Pape, L. Hornbeck." Characteristics of the Deformable Mirror Device for Optical Information Processing". Optical Engineering, vol.22, no. 6, 675-681, November-December 1983.

34. Y.S. Abu-Mostafa, and D. Psaltis."Optical Neural Computers". Scientific American, 256, 88-95, March 1987.

35. H.J. White, and W.A. Wright. "Holographic Implementation of a Hopfield Model with Discrete Weightings." Applied Optics, vol 27, no.2, 331-338, January 15, 1988.

36. E.G. Paek, J.R. Wullert, and J.S. Patel."Holographic Implementation of a Learning Machine Based on a Multicategory Perceptron Algorithm". Optics Letters, vol.14, no.23, 13030-1305, December 1, 1989.

37. R. Athale, and C. Stirk." Compact Architectures for Adaptive Neural Nets". Optical Engineering, vol. 28, no. 4, 447-455, April 1989.

38. D. Psaltis, C.H. Park, and J. Hong."Higher Order Associative Memories and Their Optical Implementations". Neural Networks, vol 1, 149-163, 1988.

39. N.H. Farhat, D. Psaltis, A. Prata, and E. Paek."Optical Implementation of the

Hopfield Model." Applied Optics,vol. 24, no.10, May 15, 1985.

40. N.H. Farhat."Optoelectronic Analogs of Self-Programming Neural Nets: Architecture and Methodologies for Implementing Fast Stochastic Learning by Simulated Annealing". Applied Optics, vol. 26, no. 23, 5093-5103, December 1, 1987.

41. N.H. Farhat." Optoelectronic Neural Networks and Learning Machines".IEEE Circuits and Devices Magazine,32-41,September 1989.

42. H. Yoshinaga, and K. Kitayama."Experimental Learning in an Optical Perceptronlike Neural Network". Optics Letters, vol.14, no. 14, 716-718, July15, 1989.

43. B. Macukow, and H. Arsenault." Optical Associative Memory Model Based on Neural Networks Having Variable Interconnection Weights". Applied Optics, vol. 26, no. 5, 924-928, March 1, 1987.

44. T. Lu, S. Wu, X Xu, and F. Yu."Two Dimensional Programmable Optical Neural Network". Applied Optics, vol. 28, no. 22, 4908-4913, November 15, 1989.

45. M. Ishikawa, n. Mukohzaka, H. Toyoda, and Y. Suzuki." Optical Associatron: A Simple Model for Optical Associative Memory". Applied optics, vol. 28, no. 2, 291-301, January 15, 1989.

46. J. Jang, S. Shin, and S. Lee."Optical Neural-Net Analog-to-Digital Converter". Optics letters, vol. 14, no. 3, 159-161, February 1, 1989.

47. C. Peterson, S. Redfield, J. Keeler, and Eric Hartman." An Optoelectronic Architecure for Multilayer Learning in a Single Photorefractive Crystal". Neural Computation, vol. 2, 25-34, 1990.

48. M. Oita, J. Ohta, S. Tai, and K. Kyuma." Optical Implementation of Large Scale Neural Networks Using a Time-Division-Multiplexing Technique". Optics Letters, vol. 15, no. 4, 227-229, February 15, 1990.

49. M. Oita, M. Takahashi, S. Tai, and K. Kyuma." Character Recognition Using a Dynamic Optoelectronic Neural Network with Unipolar Binary Weights". Optics Letters, vol. 15, no. 21, 1227-1229, November 1, 1990.

50. J.S. Jang, S.W. Jung, S.Y. Lee, and S.Y. Shin." Optical Implementation of the Hopfield Model for Two Dimensional Associative Memory". Optics Letters, vol. 13, no.3 248-250, March 1988.

51. C. Mead and M. Ismail. Analog VLSI Implementation of Neural Systems. Kluwer Academic Publishers, 1989.

52. H. Graf, L. Jackel, and W. Hubbard. "VLSI Implementation of a Neural Network Model". IEEE Computer, 41-49, March 1988.

53. S. Yoo, L Anderson, and Y. Takefuji. " Analog Components for the VLSI of Neural Networks". IEEE Circuits and Devices Magazine,18-25, July 1990.

54. P. Hollis, and J. Paulos." Artificial Neural Networks Using MOS Analog Multipliers".IEEE Journal of Solid-State Circuits, vol. 25, no. 3, 849-855,June 1990.

55. F. Kub, K. Moon, I. Mack, and F. Long." Programmable Analog Vector-Multipliers". IEEE Journal of Solid-State circuits, vol. 25, no. 1, 207-214, February 1990.

56. A. Rao, M. Walker, L. Clark, L. Akers, and R. Grondin. "VLSI Implementation of Optical Classifiers". Neural Computation, vol. 2, 35-43, 1990.

57. H. Ozaktas, and J.W. Goodman."Lower Bound for the CommunicationVolume

Required for an Optically Interconnected Array of Points". J. Optical Society of America, vol. 7, no. 11, 2000-2106, November 1990.

58. D. McKnight, D. Vass, and R. Sillitto." Development of a Spatial Light Modulator: A Randomly Addressed Liquid-Crystal-Over-NMOS Array". Applied Optics, vol. 28, no. 22, 4757-4762, November 15, 1989.

59. J. Ohta, M. Takahashi, Y. Nitta, S. Tai, K. Mitsunaga, and K. Kyuma." GaAs/AlGaAS Optical Synaptic Interconnection Device for Neural Networks". Optics Letters, vol. 14, no. 16, 844-846, August 15, 1989.

60. I. De Rycke, A. Van Calster, J. Vanfleteren, J. De Baets, J. Doutreloigne, H. De Smet, and P. Vetter." 2-MHz Clocked LCD Drivers on Glass". IEEE Journal of Solid-State Circuits, vol. 25, no. 2, 531-537, April 1990.

61. W. Welford. Useful Optics. University of Chicago Press, 1991.

62. E. Hecht, and A. Zajac. Optics. Addison-Wesley, 1979.

63. A. McAulay. Optical Computer Architectures. John Wiley & Sons, Inc., 1991.

64. T. De Marco. Concise Notes On Software Engineering. Yourdon Press,1979.

65. A. S. Tanenbaum. Structured Computer Organization. Prentice Hall, 1984.

66. w. Press. Numerical Recipes in C. Cambridge University Press, 1988.

VITA

William (Will) M. Robinson was born in the Canal Zone, on July 5, 1964 to parents Laurence Robert Robinson and Elvira Theresa Robinson. William graduated from high school in 1983 in Frankfurt, Germany while his father was stationed there. In 1985 Will entered Louisiana Tech University to study engineering. He transferred to the University of Texas at San Antonio in 1986 when his family relocated to San Antonio. Will graduated with a BSEE from UTSA in 1989. Will has worked for the US Air force at Brooks AFB with Dr. John Taboada serving as his supervisor, friend, and mentor since 1987.Will is active in powerlifting, bodybuilding, and martial arts (2nd Dan Blackbelt) and shares these activities with his wife Becky. Will plans to continue his education in Electrical Engineering, study some Biomedical Engineering and pursue a Ph.D in Computer Science where his interest will include machine learning and perception.

Permanent Address: William M. Robinson
c/o Dave and Judy Bartine
117 Westcliff Drive
Harriman, TN 37748

This thesis was typed by William M. Robinson.

END
9-94