THE PROBLEM OF DETECTING SMALL TARGETS USING MICROWAVE RADAR: A NEURAL NETWORK SOLUTION

DONALD S. BRYANT

THE PROBLEM OF DETECTING SMALL TARGETS USING MICROWAVE RADAR: A NEURAL NETWORK SOLUTION

By

Donald S. Bryant, B.Sc. (Hons.)

A thesis submitted to the School of Graduate Studies in partial fulfilment of the requirements for the degree of Master of Engineering

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

May 31, 1994

St. John's Newfoundland Canada

ISBN 0-612-17574-X

Abstract

A neural network technique has been applied to the marine radar small target detection problem. It has been compared to the conventional processing method of scan to scan integration. The results of the analysis indicate that a neural network is capable of providing performance that is at least as good as, and if the scanning window is optimized for the pulse length being used, much better than the conventional processing technique for small targets embedded in sea clutter.

TABLE OF CONTENTS

.

LIST OF FIGURES

1.0 INTRODUCTION

The ability of radar to detect small targets at sea is typically limited by the presence of competing reflections from the ocean surface. A significant amount of research has been conducted over the years on techniques for improving radar performance, especially for the detection of small targets embedded in sea clutter. The importance of this problem as it pertains to collision avoidance and search and rescue has motivated researchers to investigate all aspects of radar design and radar signal processing. However, there remains a considerable void in the level of signal processing available in civilian radar systems. One of the reasons for this is that, until recently, it has not been possible to economically implement the desired processing in a real-time system.

The introduction of advanced single-chip signal processors and high speed memory has enabled the development of radar signal processors that are able to implement proven non-coherent processing techniques. These techniques, such as scan to scan integration, have been shown to be quite effective in improving radar performance in clutter. Scan to scan integration effectively smooths the sea clutter and noise background by summing consecutive radar scans making targets more visible on the radar display. This works best when the target is stationary during the processing period. However, when both the target and radar are moving the scan to scan integration process becomes very complex, potentially limiting integration to only a few scans. This will limit potential performance improvements for this type of processor.

In this thesis the neural network has been proposed as a potential processor for the radar target detection application. The neural network is modelled on the architecture of the brain and the process of training the network to recognize a target in a background of noise and clutter is similar to the process of teaching a student to recognize the letters of the alphabet. The training enables the student to read the writing of others, even when it is poorly written. Neural networks have been found to be very effective in character recognition applications, particularly when the characters are hand written and "noisy".

In order to design and train a neural network to detect radar targets, embedded in sea clutter and system receiver noise, it is necessary to isolate the unique characteristics or attributes of the target, sea clutter and noise signal. The sea clutter component of the radar signal is a phenomenon generated by the reflection of the radar signal from the ocean surface waves. Sea clutter is a function of the directional ocean wave spectrum. That is, the magnitude of the sea clutter is modulated by the sea state. As the sea state increases so does the magnitude of the sea clutter. The noise component of the radar signal is generated by the radar receiver and is a function of the hardware used in the design of this device.

A complete radar signal has three dimensions; range, bearing and time. The magnitude of the signal varies with these dimensions. The range characteristics of a target echo will depend on the target shape and size and the radar pulse length. The bearing characteristics of a target echo will depend on the target shape and size, the radar antenna beamwidth and the pulse to pulse variation in propagation

path and target radar cross-section. Observations of targets, sea clutter and noise indicate that discrimination of targets and clutter from noise would probably be possible as a function of bearing (pulse to pulse). However, discrimination between target and clutter as a function of bearing will be much more difficult. For the scanning radar situation the rotating antenna acquires new radar signals of the same area every 2 to 3 seconds. The scan to scan (or temporal) characteristics of the target echo may be sufficiently different from the temporal characteristics of the sea clutter to permit discrimination. This would be similar to the trained radar operator who often must observe the radar display for an extended period of time over multiple radar scans before deciding on the presence of a target.

This thesis postulates that a three dimensional neural network having spatial and temporal inputs could be trained to recognize a target signature even when both target and radar system are moving. The neural network would take advantage of the spatial (range and bearing) and temporal (scan to scan) behavior of target, clutter and noise to provide enhanced target detection.

This thesis represents the first phase of a three-phase development program which has been undertaken to assess the ability of neural networks to detect targets embedded in clutter and noise. The first phase of the development is designed to investigate the basic suitability of neural networks to the radar target detection problem. Subsequent phases call for full prototype implementation and commercialization.

Can a neural network provide radar target detection performance and, if so, how does its performance compare with that of

conventional signal processing techniques? In the first phase it is considered important to keep the analysis simple such that this basic question could be answered. The approach is to focus on the stationary target detection problem. This would simplify the structure or architecture of the network and provide a foundation for the design of a more advanced network for moving targets in a subsequent phase.

A high quality data set was required for use in training and testing the prototype network. During July of 1993 a two week radar data collection experiment was conducted at Cape Spear, Newfoundland using a mobile radar unit owned by the Canadian Coast Guard. Equipment for measuring wave height and surface weather were deployed along with two reference radar reflectors in a triangular pattern at a range of 2.5 to 3.0 nmi from Cape Spear. The reference radar reflectors were Lunenburg lens type having radar cross sections of 2 and 10 m^2. A radar data acquisition system was used to collect high fidelity radar data. During the period a reasonable range of environmental conditions were encountered from a low of 1 meter swell with virtually no wind up to 3.6 m significant wave height accompanied by a 25 to 30 knot wind. Foggy and heavy rain conditions occurred. Numerous vessels passed through the area ranging in size from small wooden open boats and tour boats up to container vessels. Overall, the 14 day field trial saw the collection of about 100 Gigabytes of radar data.

The neural network kernel has been implemented on an HP Apollo workstation and has been configured to accept radar data in spatial and temporal domains. A set of programs have been created to extract selected data from the raw data sets and to present the results

of the neural network in comparison with other processing techniques. The software has been designed so that it is possible to quantify the difference in performance between the neural network and scan to scan integration. The detection performance of each technique is directly measured using a common reference making it possible to compare the network performance with scan to scan integration in a statistically meaningful way.

In the sections to follow a survey of the current thinking in neural and radar signal processing technology is presented. A detailed description of the problem is included, together with the experimental plan and problem solution methodology. The results of the development are presented graphically. Some conclusions are drawn with respect to the future development of this work into a full prototype unit.

2.0 LITERATURE SURVEY

The basic concept of marine radar is a simple one. A marine radar functions by radiating electromagnetic energy and detecting the echo returned from reflecting objects. The range , or distance to the object is calculated by using the information found in the time it takes for the radiated energy to travel to the object and back to the radar antenna. The angular location of the object in the case of the scanning marine radar is found by using the angular position of the scanner. The marine radar is an active device. It uses a transmitter and does not depend on ambient radiation, as do most optical and infrared devices. Radar can detect relatively small objects at near or far distances and can measure their range with precision.

Radar (**ra**dio **d**etection **a**nd **r**anging) was originally developed to satisfy a military need. It has been used by the military for surveillance and weapon control. Military applications have funded most of the development of this technology. However, radar has been used extensively in civil applications for the safe travel of aircraft, ships and spacecraft.

Objects having high conductivity such as metal ships are very good reflectors of radio waves and as a result they provide very strong radar returns. Objects with low conductivity such as wooden boats, rubber life rafts and icebergs are all very poor reflectors of radar signals. The shape and surface conditions are also factors in determining the strength of the signal reflected from a target. The detection problem is compounded by the requirement to detect these weak targets in a background signal being reflected from the ocean itself. This signal is known as sea clutter. The ocean with its salt

content may provide stronger radar reflections than the small targets. Reflections from the ocean increase as a function of wind speed and wave height making small target detection even more difficult in high wind and wave conditions (Ryan 1992).

There are techniques which improve the detection of small targets embedded in sea clutter. One of the best techniques is scan to scan integration. This technique is based on the fact that the sea clutter relative to the target may be viewed as a non-coherent process (Ryan 1990). Scan to scan integration involves the adding together of successive scans of radar data in order to reduce the sea clutter component of the radar return. This processing scheme has been proven to improve the detection of small targets at sea (Ryan 1990). Techniques which are able to further absorb the nature of target and sea clutter may be even more effective in the detection application. This leads to an investigation of neural networks.

The motivation for neural network research is found in many disciplines. Neural nets have their genesis in biology and psychology. Indeed, in 1949, in his book *The Organization of Behavior*, a psychologist Hebb, postulates the following:

> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*

In this book Hebb goes on to say that changes in synaptic strengths between neurons are proportional to the activation level of the neurons. This is the formal basis for the creation of artificial neural networks with the ability to learn.

The theory of Hebbian learning describes a method for updating synapse strengths in neural networks enabling them to learn. This idea was incorporated into a two-layer network called a perceptron (Rosenblatt 1957-1958). The learning rule formulated by Rosenblatt states that the weights should be adjusted in proportion to the error between the output neurons and the required target values for these neurons. Rosenblatt later tried to formulate a three-layer version of his algorithm but failed as he was unable to deduce a sound method for training the weights associated with the hidden layer neurons (i.e., the layer between the input-layer and the output-layer). A device was developed called the ADALINE (adaptive linear combiner) together with a new learning rule which minimized the summed square error during training (Widrow 1962). The ADALINE proved useful in applications such as pattern recognition and classification.

Many problems could not be solved using the simple two-layer networks. Multi-layer nets had to be investigated, however, three key factors led to a decline in the research in this area and artificial neural networks in general during this period. First, the lack of a mathematical method for training multi-layer networks was a significant problem. Second, the relatively modest computational power available to train these neural networks during this period meant that only a few researchers had the ability to do work into the training and testing of networks. Third, a book, *Perceptrons*, was published by Minsky and Papert (1969). This book outlined in detail the limitations of the two-layer design. The authors speculated that while the multi-layer networks might be able to overcome most or all

of these difficulties the multi-layer architecture could not be trained and therefore was also a dead end.

Some research into the two-layer design continued. A two-layer net was used to build a content addressable memory (Kohonen 1984). A content addressable memory system uses the item to be stored as the index for its location in memory. Kohonen called this *associative memory*. Associative memory is based on an unsupervised learning method in which the weights are changed only on the basis of the training patterns presented without taking some desired result into the equations.

During the 1960's and continuing into the 1980's Stephen Grossberg had been developing models of the brain's function (Grossberg 1982). His research has resulted in several unique neural network models, which are able to do training on-line and have the capacity for self organization.

However, it was not until the discovery of backpropagation by Paul Werbos in 1974 that the field of neural networks experienced a resurgence of research activity (Werbos 1974). Backpropagation allows the training of multi-layer networks. Werbos discovered the algorithm while working on his doctoral thesis in statistics. At that time he called it the *dynamic feedback technique.*

One of the latest developments in artificial neural systems is the *Cascade-Correlation Learning Architecture* (Fahlman and Lebiere 1991). This is a new architecture for supervised learning in artificial neural networks. The method does more than modify the weight values of a fixed neural network structure it alters the topology of the net at the same time. Cascade-Correlation begins with a minimal network

topology, then automatically trains and adds new hidden layer neurons one at a time gradually creating a multi-layer structure. When a new hidden layer neuron has been added to the network, its input side weights are fixed. This unit then becomes a permanent feature-detector in the network, only available for producing outputs or for creating other feature-detectors. The real advantages of this method are: it learns very quickly relative to standard backpropagation, the network determines its own size, complexity and topology, it retains the structures it has built even if the training set changes, and it does not back-propagate error signals through the connections of the network.

3.0 THEORY

A definition of neural networks might be written as follows:

> *Artificial neural systems, or neural networks, are physical cellular systems which have the ability to acquire, store, and make use of experiential knowledge. (Zurada, 1992)*

The data or knowledge gained by a neural system is in the form of a system state (stable or otherwise), or a mapping embedded in the network itself. The information in whatever form may be recalled in response to a particular set of cues or a set of patterns presented to the system.

The current state of the art in artificial neural networks would indicate that it is possible to describe them as a mathematical attempt to model the way the brain functions, in order to harness its ability to infer from incomplete, or conflicting information.

Why is there such interest in neural networks? In trying to answer this question, consider how nature deals with the pattern recognition problem. Animals, in general, are much better and faster at recognizing images than are most digital computers. However, digital computers outperform biological and artificial neural systems for tasks based on precise arithmetic operations. Artificial neural systems represent a very promising class of information processors. Neural nets can add to the processing power of the von-Neumann digital computer with the ability to make decisions and to learn, in much the same way as animals, by ordinary experience.

Artificial neural networks have their foundations in biology and as such this discussion would not be complete without discussing briefly these biological neural systems. A biological neuron is composed of axons, dendrites, and synapses. These neurons undergo excitatory and inhibitory signals. One excitory signal on its own is usually too weak to trigger an action potential in the postsynaptic neuron. Its effect is said to be subliminal or below the threshold level. Many excitory signals may however be added together, a process called summation. Temporal summation occurs when repeated stimuli cause new excitory signals to form before the previous one has faded. In this way the neuron may be brought to firing level. Neural integration is the process of adding and subtracting incoming signals and processing to determine the correct response. Hundreds of stimuli may be absorbed before an impulse is actually transmitted. Each neuron acts as an integrator, sorting through the thousands of pieces of information continuously received. The artificial equivalent neuron, or processing element, simulates the axons and dendrites of its biological counterpart with electrical wires and models the synapses by using resistors with weighted values.

Neural network models consist of processing elements, interconnection topologies, and learning rules. The processing elements themselves consist of combinations of excitatory (generally positive) and inhibitory (generally negative) weights which act on the inputs in a summation function driven by an activation function which is based on the inputs to the processing element.

Each processing element may interact with the others in the network depending on how they are interconnected. In a fully

connected network the topology dictates that each node or processing element is connected to each other node in the net (see Figure 1). In practice the network is usually layered, that is, the network is stratified into sets of nodes which are not connected to any other node within its own set, but each node in the set is fully connected to the nodes in the adjacent layer (see Figure 2). One of the key elements required in setting up a neural net is the definition of the network topology. The architecture of the network is usually determined experimentally through a process of trial and error.

To talk about the concept of learning in neural networks requires a new set of terms and expressions. For example; a neural net is not programmed, it is taught. Consider the human cognitive process and the network training problem. The human brain takes no longer than a few milliseconds to complete most of its cognitive processing tasks. It is a fact that individual neurons in the human brain compute operations at a rate comparable to the time required to execute a single instruction in a digital computer. So, how does the brain accomplish these tasks in such a relatively short period of time? The answer lies in the brain's use of massive parallelism, that is, the brain makes use of as many as 10 billion neurons and, of course, more than 1000 times as many interconnections, depending on the task at hand, and the topology employed. To simulate this massive parallelism the artificial neural network sets up an interconnected array of processing

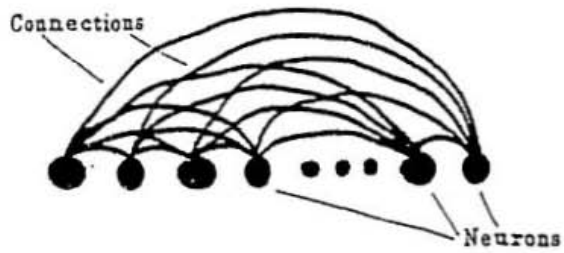


Figure 1. Fully Connected Neural Network

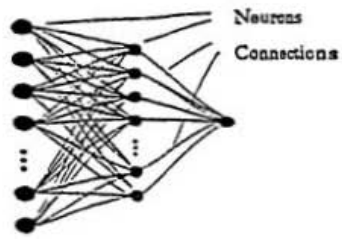


Figure 2. Layered Neural Network

elements. Each processing element has a number of inputs, together with a set of mathematical states and an output that is generally a non-linear sigmoidal function of the inputs. Each input to the processing element has a weight value associated with it which usually ranges from -1 to 1 although it may exist as any real number. When an element is activated it looks at all the inputs to it and then computes their respective weight values. If the calculated value is above some predetermined level the processing unit will generate an output value that is used as input by other processing elements. In most learning rules the only element to be adjusted during training of the neural network is the weight value. So, the training of a neural network is a matter of adjusting the weights, this may be done manually or automatically, using some set of mathematical or logical rules which will be discussed later. In terms of graph theory a neural net may be classified as a directed graph composed of a number of nodes which we call processing elements. Each of the nodes has only one output signal or value which is distributed to other processing nodes, while each of the nodes must process the incoming signal based on the values of the constants stored in it. The current neural net technology is based on the assumption that the update of any signal within a node is done discretely, rather than continuously or concurrently.

There are essentially three ways for neural net learning to take place:

1. Supervised

2. Unsupervised

3. Self-supervised

In the first case the neural net programmer must provide trial and error inputs to the network, thereby teaching the network the correct and the incorrect responses. With unsupervised learning the data is simply entered for access by the net without any programmer intervention. In general this process should lead to an internal data clustering which is the desired result.

Self-supervised learning occurs when the network is actively monitoring itself and dynamically correcting errors found in the interpretation of data, this is usually accomplished by feedback through the network. Training a neural system effectively synthesizes a set of underlying rules from a body of data or set of patterns. It is in the learning stage that the network encodes the required transformation, which maps a desired set of input features to a specific set of output features. In general, neural network topologies can be set up to generate arbitrarily complex decision regions for stimulus-response pairs. This inherent ability makes neural nets ideally suited for use as detectors or classifiers. One of the real advantages of neural networks lies in their parallel distributed processing structures. Neural nets do not execute a predetermined set of instructions, but evaluate many competing hypotheses simultaneously, thus increasing the processing speed. Neural nets provide other inherent benefits as well, for instance, neural net classifiers are non-parametric and make no assumptions about the probabilistic properties of the distribution of the training data.

The layered network is a feed forward network and as such provides for a reasonably fast supervised learning ability. The layered nets are also easily trainable, and can generate arbitrary mappings.

It has been shown that the three layer network can form arbitrary complex decision regions (Gibson and Cowan, 1990) that are not limited to convex shapes. Thus, it would seem that no more than three layers is required to solve arbitrarily complex classification mapping problems although experimentation has shown two may suffice.

The following is considered to be the general neural net learning rule (Amari, 1990):

- $W_i(t)$ The weight vector at time t. The members of $W_i(t)$, w_{ij} connect the j'th input value with the i'th neuron. The j'th input can be the output from another neuron or it can be an external input to the system.
- o(t) The output vector at time t.
- X(t) The input vector at time t.
- v(t) The learning vector at time t.
- $d_i(t)$ The teaching vector or desired response at time t.
- lrate The learning rate (usually less than or equal to unity).
- $net_i(t)$ The dot product of the weight vector and the vector X.
- f(arg) The network activation function.

The magnitude of the weight matrix increases proportional to the product of the input signal and the learning vector. The learning vector is a function of , X(t) and in some methods $d_i(t)$:

$$v = v(W_i(t),X(t),d_i(t)) \qquad (1)$$

The change in the weight matrix as dictated by the learning step at some time t according to the general learning rule is:

$$\Delta W_i(t) = \text{lrate}\ (v(W_i(t),X(t),d_i(t))\ X(t)) \qquad (2)$$

and,

$$W_i(t+1) = W_i(t) + \Delta W_i(t) \qquad (3)$$

The above holds for discrete-time learning, in the continuous case, in place of equation (2) we would have:

$$dW_i(t)/dt = \text{lrate}\ (v(W_i(t),X(t),d_i(t))\ X(t)) \qquad (4)$$

The following are various techniques that are used to put this general learning rule into practice:

Hebbian Learning Rule

In the Hebbian method the learning vector "v" is chosen to be equal to the neuron's output (Hebb, 1949);

$$v = f(W_i(t)\, X(t))$$

and,

$$\Delta W_i(t) = \text{lrate}\ f(W_i(t)\, X(t))\, X(t)$$

This method of teaching the network requires that the initial values of the weight matrix be set to small random non-negative values before starting the process.

Perceptron Learning Rule

In this case the learning vector is chosen as the difference between the desired and the actual neuron's response (Rosenblatt, 1958). This is an example of supervised learning and may be written as:

$$v = d_i(t) - o_i(t), \qquad o_i(t) = \text{sgn}(W_i(t)\, X(t))$$

and,

$$\Delta W_i(t) = \text{lrate}\ (v\, X(t))$$

so,

$$\Delta w_{ij} = \text{lrate}\ (v\, X(t))\, x_j(t) \quad \text{for } j = 1,2,3,\ldots, n$$

It should be noted that for this rule there exist some serious limitations on the neural response expected. The rule is only valid for binary neuron response and since then the desired response could only be 1 or -1, the change in the weight matrix may be reduced to,

$$\Delta W_i(t) = \mp 2.0 \text{ lrate } X(t)$$

the plus sign is applicable if $d_i(t) = +1$, and $\text{sgn}(W_i(t)\, X(t)) = -1$, and a minus sign when $d_i(t) = -1$, and $\text{sgn}(W_i(t)\, X(t)) = +1$. In this method the weights may be initially set arbitrarily.

Widrow-Hoff Learning Rule

This method (Widrow 1962) is used for supervised training of neural networks. It is independent of the activation function used since it minimizes the squared error between the desired output vector $d_i(t)$ and the neuron's activation value $net_i(t) = W_i(t)\, X(t)$. The learning vector for the rule is given as follows:

$$v = d_i(t) - W_i(t)\, X(t)$$

and,

$$\Delta W_i(t) = \text{lrate } (v\, X(t))$$

This rule is considered to be a special case of the *delta* learning rule. It is sometimes called the LMS (*least mean square*) learning rule. The weights may initially be set to any values.

Delta Learning Rule

This rule is valid for continuous activation functions, and may only be used in the supervised training mode. The learning vector for this method is called *delta* and is defined as,

$$v = [d_i(t) - f(W_i(t)\, X(t))]\, f'(W_i(t)\, X(t))$$

f' is the derivative of the network activation function $f(net_i(t))$ computed for $net_i(t) = W_i(t)\, X(t)$. This rule may be shown to be based on the method of least squared error between $d_i(t)$ and the output vector $o_i(t)$. This rule was introduced recently by (McClelland and Rumelhart 1986).

Correlation Learning Rule

By substituting $v = d_i(t)$ into the general learning rule it is possible to obtain the correlation learning rule. The change in the weight vector is defined as,

$$\Delta W_i(t) = lrate\, d_i(t)\, X(t)$$

This rule is a special case of Hebbian learning.It states that if $d_i(t)$ is the desired response due to $X(t)$ then the required weight change is proportional to their product. This method requires that the weights initially be set to zero.

The next set of learning techniques are best described in the context of a layered neural network topology.

Winner-Take-All Learning Rule

This rule is considered an example of competitive learning. It is used for unsupervised network training. Most often this method is used for learning statistical properties of the inputs (Hecht-Nielsen, 1987). Learning here is based on the principle that there will exist a neuron in the m'th layer having maximum response due to the presentation of X(t). This neuron would be declared the winner in this case. As a direct result of this, the weight vector $W_m(t)$, would be the only one adjusted for this step and its change would be given as,

$$\Delta W_m(t) = \text{lrate}\ (X(t) - W_m(t))$$

The selection of new winners is based on the following formula of maximum activation among all n neurons in the competition:

$$W_m(t)\ X(t) = \text{MAX}\ [W_i(t)\ X(t)]\ ,\ \text{for}\ i = 1,2,3,...,n$$

In this method the winning neuron is sometimes extended to the winning neighborhood of neurons. This has the effect of increasing the generalization of the final network. The weights are usually initially set to random values.

Outstar Learning Rule

This rule is designed to produce a desired response $d_i(t)$ in the layer n neurons. The rule is used to provide learning of repetitive and characteristic properties of stimulus-response relationships. The method is oriented towards supervised learning. It allows the network to extract statistical properties of the input and output vectors (Grossberg, 1982). The change in the weight matrix is given by,

$$\Delta W_i(t) = \text{lrate}\ (d_i(t) - W_i(t)), \text{ for } i = 1,2,3,...,n$$

The weights are usually initially set to random values.

Evolutionary Programming Technique

Another alternative learning method is called evolutionary programming (Grossberg, 1982). This technique is a simulation of natural evolution and as such is very similar in principle to the Winner-Take-All method discussed earlier. In this technique each weight vector (organism) is assigned a score based on how well it performs. Each "parent" vector is modified (mutated) at random in accordance with a Gaussian distribution having zero mean and a variance proportional to its error score. These mutations or "offspring" are then put in competition with the parents for survival to the next generation. As this process iterates, superior vectors should emerge from the evolution. Evolutionary programming can be directly applied

to the learning problem as it generates an optimal set of network weights in much the same way as standard back-propagation does.

Back-Propagation Learning

The training method most commonly used is back-propagation (Werbos, 1974). In this algorithm the weights of the network or strengths of the neural connections are modified numerically based upon error changes which are propagated backwards through the network. This optimization of the weight structure is essentially a "steepest decent" search of the network weight space and while it may be said that the technique itself is a numerically stable one, in that it will always find a local minimum, it may fail to reach the global minimum due to the inherently irregular shape of the search space. Any neural network must compute by a process of spreading activation. One way, to do this follows:

$$x_i(t) = X(t), \qquad 1 \le i \le m$$

$$net_i(t) = \sum_j W_i(t)\, x_i(t), \qquad m < i \le (N+n),\ 1 \le j \le (i-1)$$

$$x_i(t) = sig(net_i(t)), \qquad m < i \le (N+n)$$

$$Y^*_i(t) = x_{i+N}(t), \qquad 1 \le i \le n$$

$$Y_i(t) = \text{Target Outputs} \qquad 1 \le i \le n$$

where the function described by sig(arg) is usually a sigmoidal function such as:

$$sig(arg) = 1/(1 + exp(-arg))$$

and where N is a constant which can be any integer, in a fully connected network as long as it is no less than m. However in a layered network, N determines the number of neurons in the hidden layers. The value of (N+n) gives the total number of neurons in the system. The value of $net_i(t)$ represents the total level of voltage exciting neuron i, and $x_i(t)$ represents the intensity of the resulting output from neuron i. This output is sometimes referred to as the activation level of the neuron.

As can be seen, the real problem now in training the network is to correctly choose the weights $W_i(t)$ so as to suit the purpose. Back-propagation, determines the weights by minimizing the following error function:

$$\text{Error} = \sum_t E(t) = (1/2) \sum_t \sum_i (Y^*_i(t) - Y_i(t))^2$$

where, $1 \leq t \leq T$ and, $1 \leq i \leq n$.

This approach is shown in Figure 3.

Back-propagation Data Flow Diagram

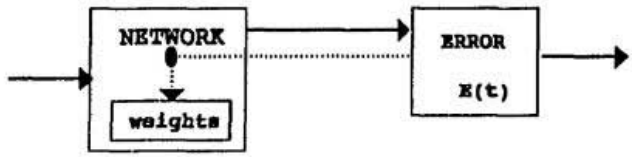


Figure 3

The weights are initially chosen as random numbers, but it may be better to estimate the weights, if any information about their values exist. The next step is to calculate $Y^*_i(t)$ and the errors E(t) for the particular set of weights. The derivatives are now calculated, that is, the partial derivatives of E with respect to each of the weights. The effect of dynamically modifying or changing any and all of the weights is now determined. A very simple approach is applied here. If increasing a given weight would lead to greater numerical error in E then that weight is adjusted downwards and visa versa. After adjusting all of the weights in the system the process restarts and iterates until some stopping criteria is reached. The stopping condition may rely on the numerical value of the error function E. One may choose a stopping condition based on satisfying the training data set. However experience has shown that this can lead to very poor generalization. It is better to rely on the underlying mathematical principles of optimization (i.e. $\partial E/\partial W_i(t) = 0$ or nearly so).

From the chain rule it follows that:

$$\partial^+ \text{Target}/\partial z(i) = \partial \text{Target}/\partial z(i) + \Sigma_j \, \partial^+ \text{Target}/\partial z(j) \, \partial z(j)/\partial z(i)$$

with,

$$j = i+1,...,N \text{ and } i = 1,...,N$$

where the derivatives with the + superscript represent the total derivatives, and the derivatives without it represent the ordinary partial derivatives. This result is valid only for systems where the values to be calculated can be obtained one by one in the order z(1), z(2), z(3), ...,z(n), Target. As an example, consider a simple system of two equations, in order:

$$z(2) = 6\ z(1)$$
$$z(3) = 2\ z(1) + 5\ z(2)$$

The partial derivative of z(3) with respect to z(1) is 2, to calculate this value we need only look at the equation which determines z(3) directly, however, the total derivative of z(3) with respect to z(1) is 32 because of the indirect impact added by z(2). The partial derivative measures what happens as we change z(1) and assume everything else remains constant. The total derivative measures what happens when z(1) is changed and also monitors all other changes related directly or indirectly to the system.

Define T_z(i) as the total derivative of the target with respect to z(i), which may be interpreted as the feedback in the system to z(i). In backpropagation the target is the error function E, already defined, so the required equation for the total derivative in this case is given by:

$$T\_Y^*_i(t) = \partial E/\partial Y^*_i(t) = Y^*_i(t) - Y_i(t),$$

which follows from the differentiation of the formula for the E function, so,

$$T\_x_i(t) = T\_Y^*_{i\text{-}N}(t) + \sum_j W_i(t)\, T\_net_i(t),$$

where, $j=i+1,\ldots,N+n$, and, $i=N+n,\ldots,m+1$

$$T\_net_i(t) = sig'(net_i(t))\, T\_x_i(t)\ ,\ i=N+n,\ldots,m+1$$

and,

$$T\_W_i = \sum_t T\_net_i(t)\, x_i(t),\ t=1,\ldots,T$$

where, $sig'(arg)$ is the derivative of $sig(arg)$

It can be shown that;

$$sig'(arg) = sig(arg)\,(1 - sig(arg)),$$

which can be beneficial during the implementation phase of the method. To derive the weights, the equation in backpropagation is:

$$W_i(t+1) = W_i(t) - lrate\ T\_W_i.$$

4.0 DESCRIPTION OF PROBLEM

The ability of marine radar to detect small targets at sea is typically limited by the presence of backscatter from the ocean surface coupled with receiver noise. This phenomenon of ocean radar backscatter is commonly known as sea clutter. The detection of small objects at sea by marine radars is of interest for a number of reasons.

One is the issue of ice-infestation of ocean waterways, which is a serious navigation problem. All ocean going vessels must rely on the ships radar as the primary sensor for navigation purposes. In this environment however, conventional marine radars do not perform to the satisfaction of the ships operator. It is known that as large icebergs melt they break into pieces called growlers and can weigh as much as 100 - 150 metric tonnes. Some of these pieces are still large enough to be considered very hazardous to shipping. Growlers are very hard to detect with marine radar as only 1 - 2 metres of the berg is actually above the water. So, even in a calm sea (i.e., 1 - 2 metres significant wave height) the radar return from a growler will clearly be difficult to detect over the competing sea clutter return. As the wave height increases the problem of growler detection becomes more severe.

The success of agencies responsible for search and rescue operations at sea is severely diminished by the inability of current marine radar technology to find small targets, such as liferafts, out of sea clutter. In a 1987 search and rescue (SAR) experiment conducted for the Canadian Coast Guard it was found that for four and six man life rafts with no radar enhancement, the search track sweep width is essentially zero (Dawe et al., 1987). This means that a search for objects such as these which are the most common size life rafts in use

today, has little chance of success. This study concluded that there is clearly a requirement for marine radars which are specifically designed to detect weak targets embedded in sea clutter. It went on to say that while optimizing the radar system parameters (i.e., scanner speed, transmit power, pulse scheme, receiver design, etc.), might improve the radar's response to weak targets in sea clutter, the most effective scheme to improve performance would be tc concentrate on signal processing techniques.

Traditionally, sea clutter has been modeled as a purely stochastic process. Non-coherent processing techniques such as scan to scan integration have been shown to improve radar performance in clutter. However, these techniques have limitations especially when integration is only carried out over a few scans. The scan to scan technique performs a numerical average of n scans of digital data. The decorrelation time of the sea clutter to be removed from the radar screen is a function of a number of environmental parameters. The dominating parameter at any instant in time however seems to be sea state. It is not entirely clear how many scans should be integrated under any particular set of environmental conditions in order to optimize small target detection in sea clutter. Also, it is clear that if the target in question is moving, then in order to integrate the digital scans of radar data they must first be registered spatially. This image registration itself will be difficult to accomplish. The implementation of these scan to scan techniques requires significant digital processing power.

Techniques that take advantage of the temporal behavior of target, clutter and noise, together with the spatial signature, may be

more efficient in this application. A neural network trained to extract the temporal and spatial characteristics of targets in clutter and noise may perform better than conventional techniques. Another potential advantage of the neural techniques is the fact that they can now be implemented directly in hardware. In fact, Intel Corporation has introduced the Electrically Trainable Analog Neural Network (ETANN 80170NX) chip.

5.0 EXPERIMENT

In order to test the hypothesis that a neural network technique applied to the radar small target detection problem might give better results than conventional techniques (i.e., scan to scan integration) digital radar data was gathered. Also, a suite of neural network software tools were designed and developed. The neural net tools include routines to train a network based on standard backpropagation as the learning method, as well as a Windows 3.1 based application for the manipulation of the digital radar data. The Windows 3.1 application also has the ability to do the required test, evaluation, and comparison, to conventional techniques of the trained networks, on the radar data.

5.1 DATA COLLECTION

The radar station was established at Cape Spear near the operational lighthouse. The orientation of the radar station relative to the topography at the cape placed the area of study in the coastal waters to the north east of Cape Spear. The digital radar data collection system consisted of an IBM compatible computer equipped with a Precision Digital Images (PDI) 15 AT video capture board. The PDI board will allow for a 1k x 1k x 8 bit single frame capture at a sample rate of up to 40 Mhz. The radar data collection device gathers digital data at 8 bits of resolution which translates into 256 digital levels. The transistor-transistor logic (TTL) signals from the radar itself are used as external syncs. The radar heading marker was used as the vertical sync signal or start of scan. The radar pulse trigger was used as the horizontal sync signal or start of scan line. For the

purposes of this study a subset of the total 120 gigabyte radar dataset was extracted. There were 6 datasets extracted each contained 100 scans of radar data. The first 3 radar datasets selected consisted of 100 scans of 100 lines by 100, 8 bit pixels the second 3 sets contained 100 scans of 100 lines by 200, 8 bit pixels. Each of the first 3 datasets where chosen with 3 targets available for detection and the second 3 sets with 2 targets available minimum. The first dataset may be characterized as a receiver noise dataset collected with the radar set to long pulse, having a nil sea clutter component. The second, was a medium clutter dataset with the radar set to long pulse, having a significant wave height of 2.5 metres and an average wind speed of less than 5 metres per second. The third dataset considered, with the radar set to long pulse, was a high clutter dataset having a significant wave height of 3.0 metres and an average wind speed of 13.5 metres per second. The fourth to sixth datasets all were collected under high sea clutter conditions, having a significant wave height of 3.6 metres and a mean wind speed of 14 metres per second. The fourth dataset was collected with the radar set to long pulse. The fifth dataset was collected with the radar set to medium pulse and the final dataset was collected with the radar set to short pulse.

Two calibrated radar reflectors having equivalent radar cross section of 2 square metres and 10 square metres, respectively, were mounted on Alberglen fiberglass spar buoys to act as known targets. The spars placed the reflectors about 3 metres above the sea surface. The mooring system (Figure 4) was attached on the side of the spar and buoyed at the surface with a large plastic fishing float in order to eliminate any vertical torque that could cause the spar to tilt. A 200

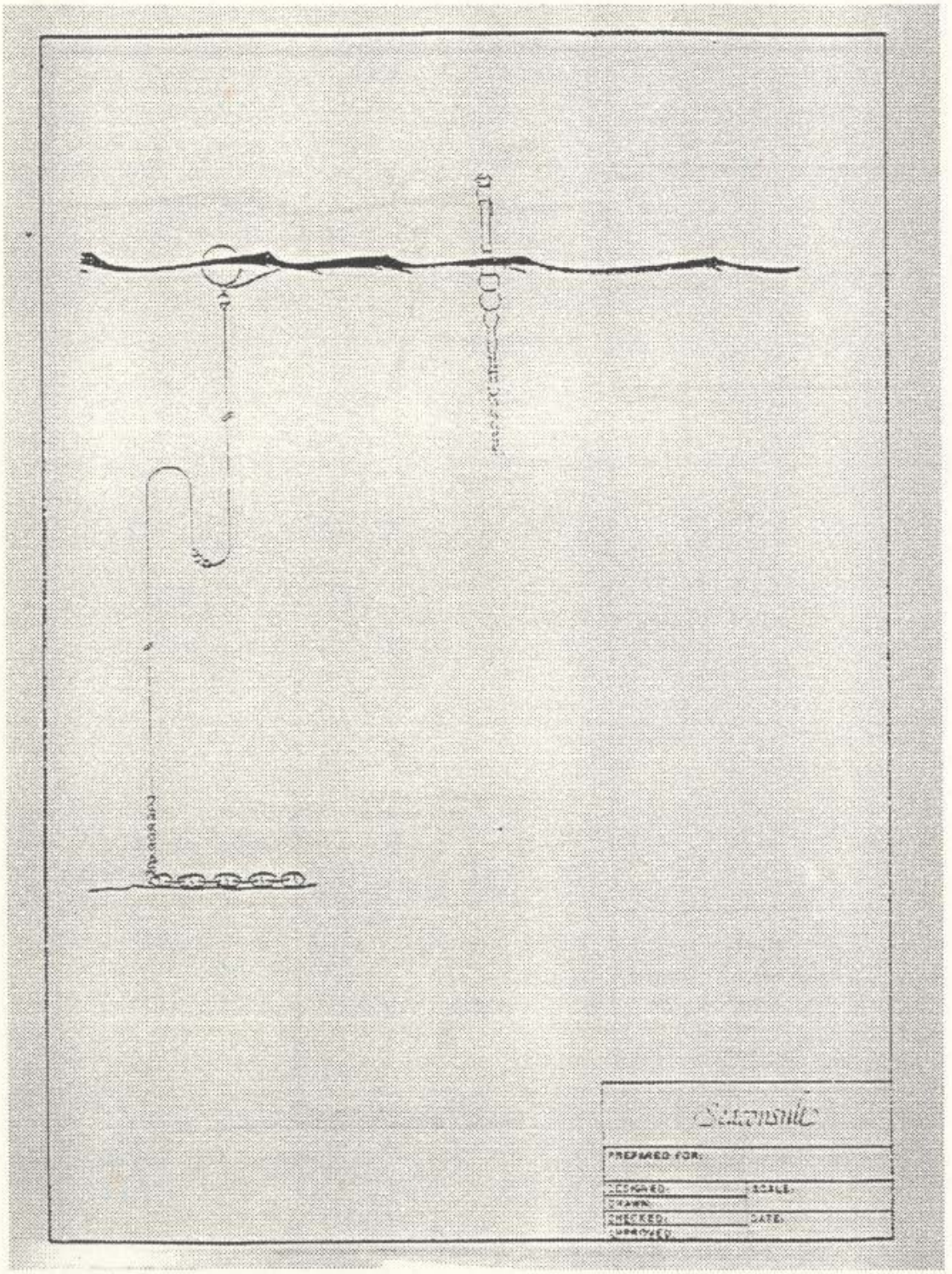


Figure 4. Spar buoy mooring used for radar reflectors and WEATHERPAK

kilogram external ballast weight was attached to the bottom of the spar to ensure that the attitude of the buoy would remain within 10 degrees of vertical.

Directional wave information was measured using a Datawell Directional Waverider. This is a spherical 90 cm diameter buoy which measures wave height, wave direction, and wave period. The buoy transmits the collected data together with some processed data to a shore-based computer receiving station. The complete wave monitoring system consisted of the directional wave buoy equipped with data telemetry transmitter, the data receiver, and an IBM compatible computer for data logging and system control. The directional waverider was moored at the site using a reliable mooring system whose design has evolved over several years of offshore use (Figure 5).

The directional waverider measures translations caused by wave motion. All calculations to determine the motions in fixed coordinate directions (north, west, and vertical) are done onboard the buoy. The determinations of spectral and directional data from the time history of the translational data are also computed onboard the waverider. Every 30 minutes, fast fourier transforms of 8 series of 256 data points (200 seconds) are added to give 16 degrees of freedom on 1600 seconds of data. Every 0.78 seconds (1.28 Hz), the three translational components and part of the most recent spectral data summary are transmitted by the buoy. Transmission of the complete spectral and directional data is completed in 250 seconds. During the 30 minutes between spectral analyses, translational data are stored to determine a new spectrum.

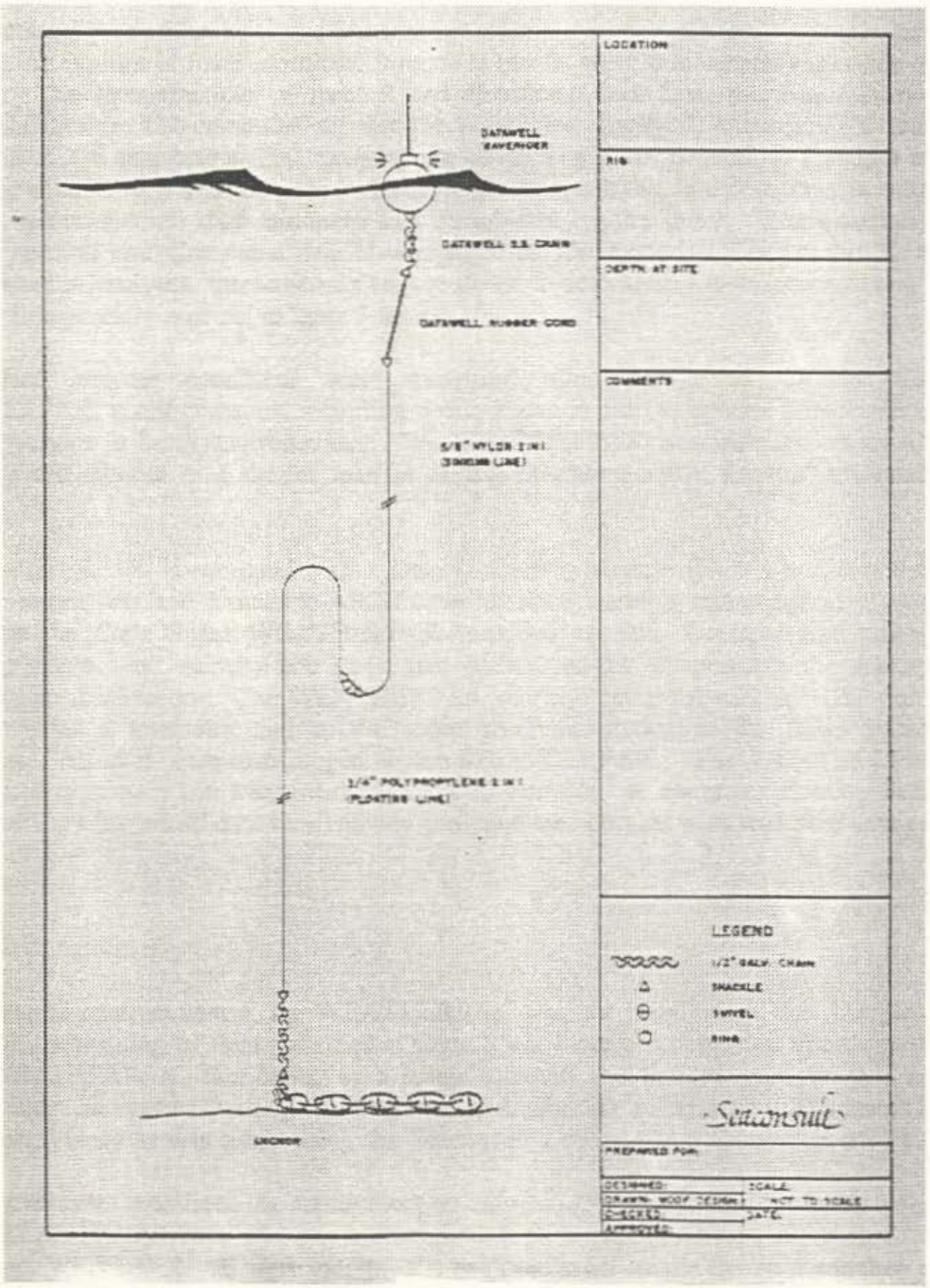


Figure 5. Directional Waverider Mooring

Sea surface weather conditions were measured using a Coastal Climate Company WEATHERPAK, a self-contained recording weather station built to operate independently over extended periods in harsh environments. The WEATHERPAK was installed on an Alberglen spar buoy and moored in a similar manner to that described above for the calibrated radar reflectors.

The WEATHERPAK is equipped with a data collection module (DCM) which collects and processes sensor data and formats to ASCII code for storage and/or transmission. The suite of sensors on the Cape Spear WEATHERPAK included an RM Young digital anemometer measuring mean wind velocity and peak gust speeds, an air temperature thermistor and a barometric pressure sensor. The WEATHERPAK was also equipped with a UHF transmitter which provided a telemetry link to the shore receiving station at the Cape Spear radar installation. All data were also logged in the WEATHERPAK itself. The WEATHERPAK was programmed to sample and transmit data every 15 minutes.

All equipment was mobilized for deployment on July 11, 1993 from the 20 metre steel fishing vessel ATLANTIC PRIZE. The day before field deployment, the Alberglen spars with radar reflectors and WEATHERPAK were individually ballasted in St. John's Harbour and then moored alongside ATLANTIC PRIZE. Prior to sailing on July 11, any remaining equipment was loaded, mooring systems were arranged on deck, and instrumentation was initialized and checked using the computerized receiving stations. Because of their awkward length and heavy ballast, the spars were towed to the location. The wave buoy was stored on the deck of the deployment vessel and lowered into the

water immediately prior to deployment. All moorings were installed by streaming the surface buoy and mooring line away from the vessel at the desired site and free-falling the anchor to the seabed. Weather at the time was very good and no difficulties were experienced in deployment. All buoys were positioned using the ship's Global Positioning System (GPS) and were very close to their planned locations. Following buoy deployment, the outputs from the directional waverider and the WEATHERPAK were monitored from the deployment vessel. The computer receiving stations were installed at the Cape Spear radar site that evening.

All moorings were recovered on July 24, 1993, using the 20-metre steel fishing vessel ATLANTIC SEA CLIPPER. Once again, the waverider was stowed on deck and the spars towed to the wharf. The receiving equipment and computers were removed from the Cape Spear radar facility on July 26, 1993.

During the field program it was observed that vessels did not always stay within the ranges suggested by St. John's VTS and that some vessels did indeed pass very close to the mooring area. To reduce the chances of collision, VTS did warn most vessels operating in the area of the buoy locations as part of routine management communications.

A summary of the wave and weather data collected near Cape Spear is presented here in Appendix I. The quality of data recovered from the directional waverider was very high.

The wave data received from the buoy consisted of raw three-dimensional accelerations along with directional spectrum and related

parameters computed on board the buoy. The data received at the shore station have been sorted into daily data files.

Hs	Significant Wave Height. Spectral approximation of Hs or $H_{1/3}$, the average of the highest 1/3 waves in a given sample. $H_{1/3}$ is intended to be the seastate that an experienced observer would report.
Tp	Peak Period. This is the period associated with the peak energy in the computed spectrum.
Tz	Mean Wave Period.
Dir (Tp)	The direction associated with the wave defined by the peak period.

Weather data from the WEATHERPAK buoy was received reliably until July 16 when the weather deteriorated and higher seastates developed. From July 16 until the completion of the field program only occasional weather data were received in real time via the UHF telemetry link at Cape Spear. On recovery, the WEATHERPAK was found to be completely operational with all data archived in the instrument's onboard memory.

5.2 NEURAL NETWORK DEVELOPMENT METHODOLOGY

The key to successful neural net development lies in the training of the network. The dataset used for training must be of very high quality and it must encompass the full range of scenarios the network will be expected to perform under.

The first step in the neural network development process was to design, develop, and test the standard backpropagation training method as a software package. This was done in C on an HP Apollo 730 Unix workstation. The next step, because of the unique nature of the radar data, was to design, develop, and test the required radar image processing software package. This was done in C on a 486 DX2 66 Mhz IBM compatible PC running the Windows 3.1 operating system. This platform was chosen for it's universal graphical user interface. For complete software listings, see Appendix II. Fundamental to the radar target identification application, the neural network is required to distinguish between sea clutter, receiver noise and the target itself. By using the radar image processing program, radar data samples were extracted into a format compatible with Mathcad 4.0 where they were graphically displayed and analyzed (see Figure 6). Based on the observed spatial nature of the radar target signature together with the desire to minimize the neural network complexity, the spatial size of the data sample window was chosen to be 2 to 3 pixels greater in each dimension (i.e., length & width) than the physical size of the radar target return. This window size varied with the radar pulse length as this radar system parameter changes the spatial size of the target signal, the longer the pulse, the longer the target will appear to be in the image sample. The temporal size of the

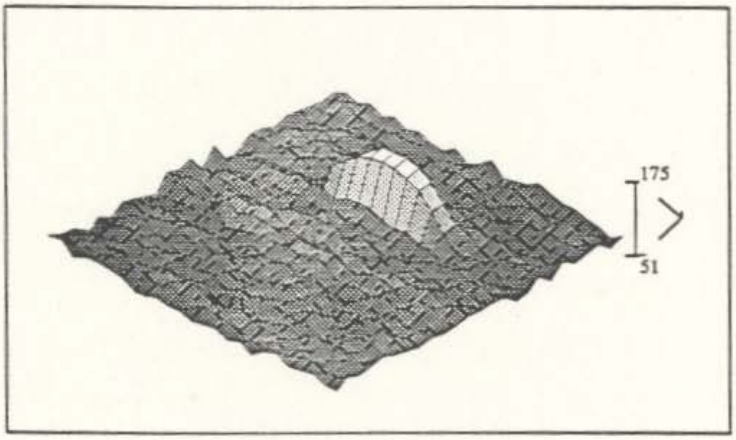


Figure 6. Radar Target Embedded in Sea Clutter

radar data sample was arrived at as a function of the coherent nature of the target, (i.e., the more scans the better), and the network complexity together with the real physical time constraint, (i.e., network complexity and time to get a processed image both increase as number of radar scans increase). The network was trained using 2, 3, and 5 scans of radar data. The objective of the neural network was to lock on to the spatial and temporal signature of the target embedded in sea clutter and receiver noise. This is possible since it is known that the radar target signature is statistically different from sea clutter and/or receiver noise on their own. In order to have the neural network absorb the physical nature of the radar target, a synthetic idealized radar target model was developed and used in the training phase. The synthetic target was constructed so that spatially it closely approximated what would be received by the radar system if it were to encounter a near perfect target return using a receiver with a zero noise figure and from a sea surface generating zero radar return (Figure 7). For each of the six datasets extracted the neural network was trained using a sample dataset consisting of 40 clutter plus noise samples, 5 receiver noise samples, and 1 synthetic idealized target model. The 40 clutter plus noise samples used in each case were taken from a dataset having a numerically similar mean wind speed and significant wave height as the datasets that would be used in the test and verification mode. The 5 receiver noise samples were taken from a dataset having a mean wind speed of zero metres per second and a significant wave height of zero metres. A three layer neural network architecture was used. The input layer size is determined by the size of the input data sample. The desired response in this case is a simple

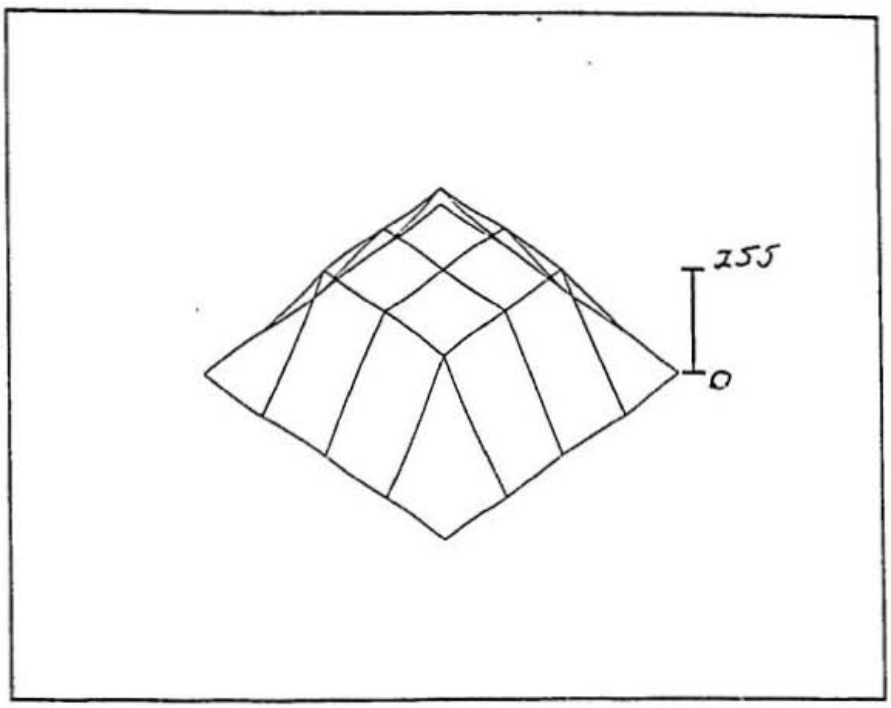


Figure 7. Synthetic Radar Target

detect/nodetect which translates directly into a single output neuron taking on values bounded between 0 - nodetect and 1- detect. This output value may be iterpreted as the probability that there exists target information in a sample being tested by the network. The middle or hidden layer size, is in fact, a more complex issue. It has been suggested that the hidden layer contain at least one neuron per training pattern (Reed, 1993). For this application it was found that a hidden layer containing between one and two neurons per training pattern yielded the highest degree of generalization. The actual optimum number of hidden layer nodes was arrived at by using a process of connection reduction or pruning. Each network was trained using a very large number of hidden layer nodes, relative to the number of training patterns, and then nodes were removed until the network performance started to degrade. At the degradation point the number of hidden layer nodes were increased until the networks performance stabilized.

All networks were trained with an optimization step size (learning-rate) of 0.55 and steepness coefficient of 0.10. The hidden layer size in all cases was found to be optimum at 4 neurons plus the number of training patterns (50 neurons). In the case of the long pulse data the optimum neural network had an input layer consisting of 300 neurons, a hidden layer of 50 neurons, and an output layer containing a single neuron. The training time in this case was 3 hours of processing time on a workstation capable of 80 million instructions per second.

6.0 RESULTS AND DISCUSSION

Six datasets were analyzed in the test and verification phase. The data analyzed in the testing phase were not used in the training phase although the trained networks were tested on data gathered during similar environmental conditions. In other words, a neural network was developed for a specific range of environmental conditions. It was found that if the network was trained on high sea state data that it performed well on data collected under similar conditions, plus, it performed well or generalized itself to data collected under much lower sea states.

The test and verification results have been summarized into a standard format used in the radar data analysis field. That is, the performance of the neural network together with that of the scan to scan technique has been plotted as "Targets Detected/Targets Available" .vs. "Average False Alarms". The y-axis is the number of targets successfully identified by the technique divided by the number of targets available for detection averaged over the number of scans of radar data analyzed. The x-axis is the number of false alarms per scan of radar data averaged over the number of scans of data analyzed. Thus, for example, one would expect the detection rate to improve with a greater number of false alarms. In this display format an ideally performing radar would show a single point in the top left hand corner of the plot indicating a probability of detection of one, while obtaining zero false alarms.

The noise data (dataset 1) when processed showed that the neural net technique was only marginally better than the standard scan to scan processing. However, both techniques performed very well

in this case. This is due largely to fact that the receiver noise is not strong enough to obscure the targets from detection, regardless of which processing technique is being used, as can be seen in Figure 8.

The performance summary for the second dataset , a medium clutter dataset, shows that at the network's best it outperforms the scan to scan processing. The performance gain is in the form of a reduction of average number of false alarms per scan. It can be seen from Figure 9 that at the peak of detection for the neural network that it is presenting a false alarm figure of about 0.30 on average per scan. By comparison, at the same detection level, the scan to scan technique presents a false alarm figure five times greater. The baseline curve indicating one scan of integrated data is the result for no processing at all, it is included as a reference curve.

The third dataset contains higher magnitude sea clutter with the significant wave height equal to 3.0 metres and the mean wind speed at 13.5 metres per second. It can be seen in Figure 10 that for both processors the ability to find targets in sea clutter is diminishing. However, the neural network demonstrates a significant improvement over the conventional processor. The neural net using 5 scans of data at the 50% detection level achieves a false alarm figure of approximately 8 false alarms per scan. The scan to scan technique using the same number of scans of data and at the same 50% detection level shows a false alarm figure greater than 18 false alarms on average per scan. In fact, the scan to scan processor only comes close to the performance of the neural net if it is allowed to operate on almost twice as long a time series of radar data. Again, the baseline curve

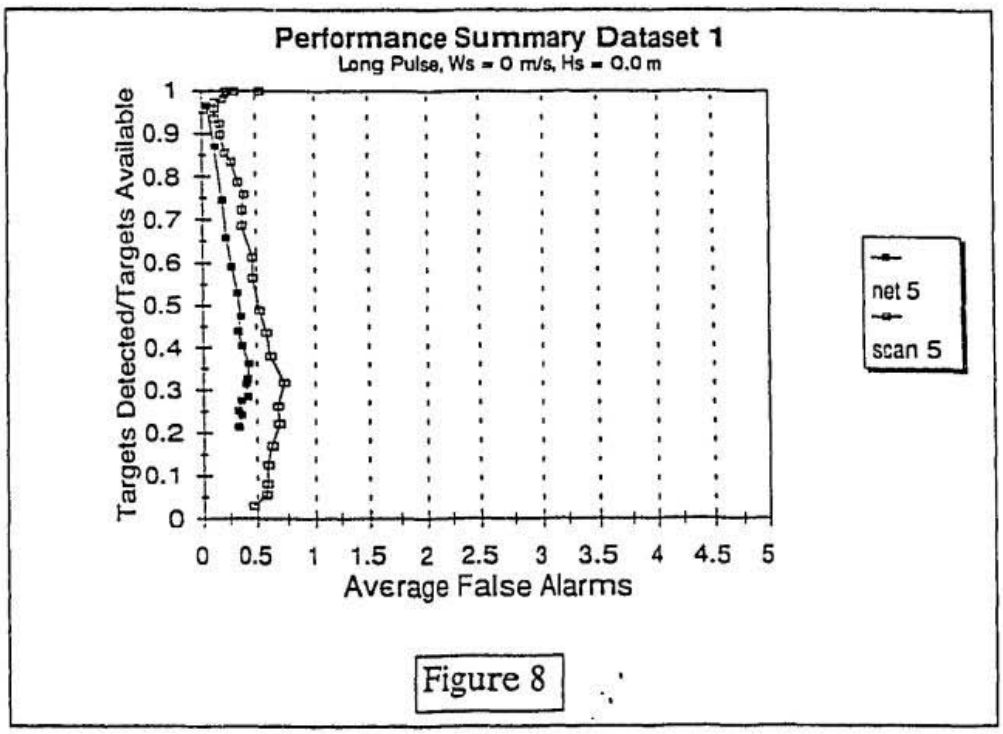

Performance Summary Dataset 1
Long Pulse, Ws = 0 m/s, Hs = 0.0 m
Targets Detected/Targets Available
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5
Average False Alarms
net 5
scan 5

Figure 8

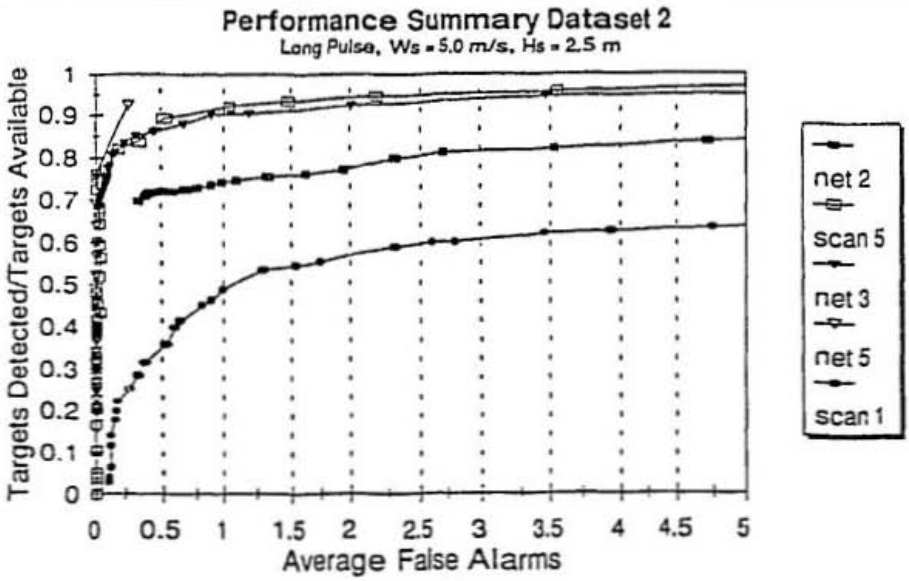

Performance Summary Dataset 2
Long Pulse, Ws = 5.0 m/s, Hs = 2.5 m
Targets Detected/Targets Available
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Average False Alarms
net 2
scan 5
net 3
net 5
scan 1

Figure 9

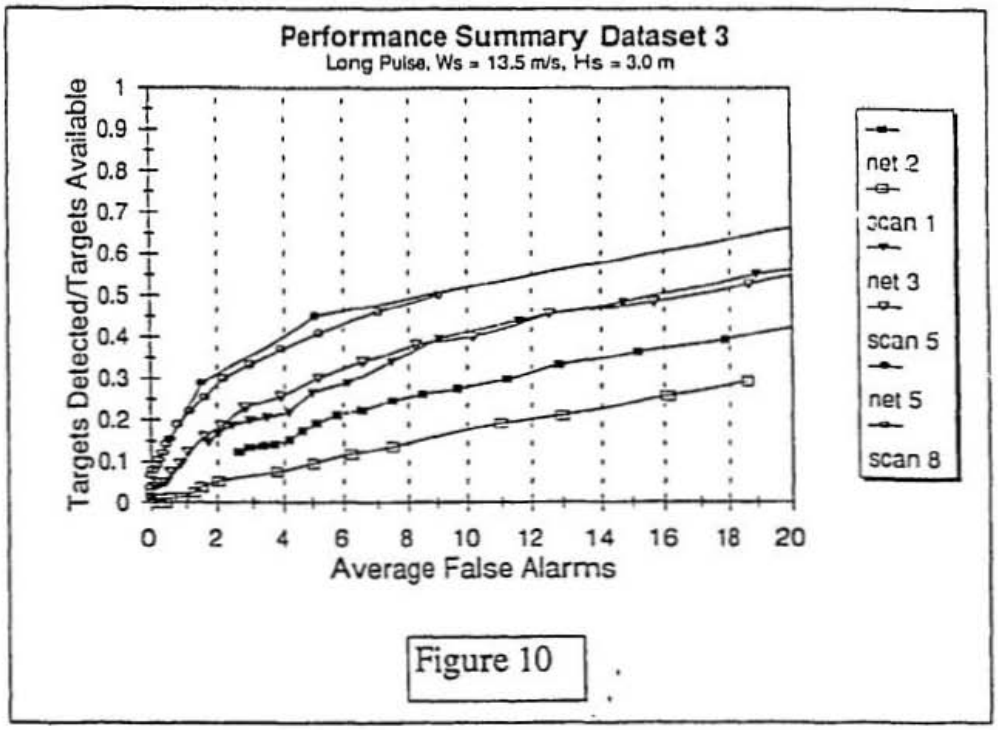

Performance Summary Dataset 3
Long Pulse, Ws = 13.5 m/s, Hs = 3.0 m
Targets Detected/Targets Available
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
16
18
20
Average False Alarms
net 2
scan 1
net 3
scan 5
net 5
scan 8

Figure 10

indicating one scan of integrated data is the result for no processing at all, it is included as a reference curve.

The fourth dataset represents the highest sea state for which data was collected with the radar set to long pulse. The significant wave height for this data was 3.6 metres and the mean wind speed was 14.0 metres per second. There is a further degradation of performance in both processors as would be expected. However, once again, the neural processor outperforms the conventional technique (see Figure 11). In this particular case the scan to scan processor using 16 scans of data can only equal the performance of the neural net using 2 scans of data. In fact, the networks performance using 5 scans of data is never approached by the conventional processor even if it is allowed to consume more than 3 times as much time series data.

The fifth dataset was collected during the same environmental conditions as the fourth except that the data were collected with the radar set to medium pulse. With the radar set to medium pulse it can be seen (Figure 12) that the overall ability to find target information out of the sea clutter is improved. In this case, the neural technique shows only very marginal improvement over the scan to scan processor. At the 50% detection level the neural processor presents a false alarm figure of 8 on average per scan while the scan to scan processor shows a figure of 9 on average per scan. This may be due to the fact that the synthetic target used in training for the medium pulse data was optimized for the long pulse setting of the radar. Time constraints prevented the optimization for the medium pulse setting.

The sixth dataset was collected during the same environmental conditions as the fourth and fifth except that the data were collected

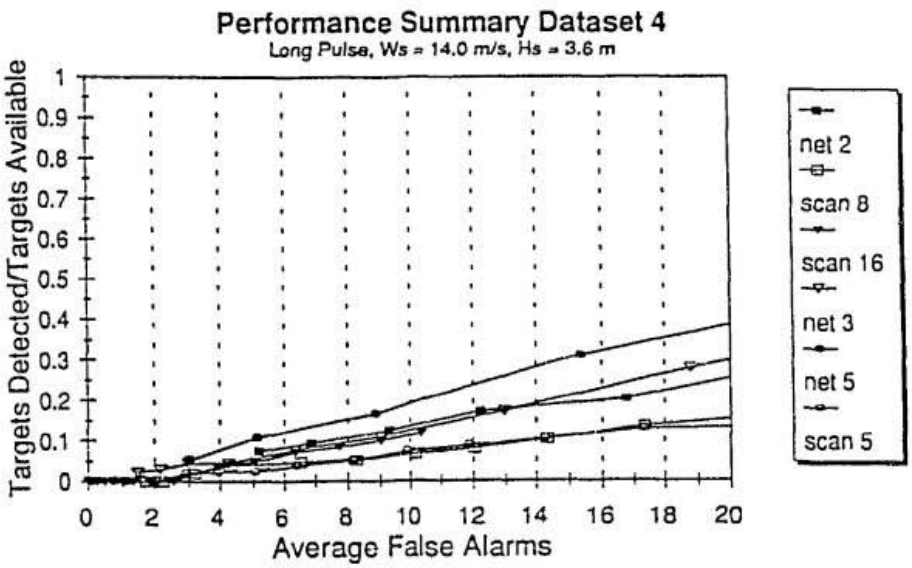

Performance Summary Dataset 4
Long Pulse, Ws = 14.0 m/s, Hs = 3.6 m
Targets Detected/Targets Available
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
16
18
20
Average False Alarms
net 2
scan 8
scan 16
net 3
net 5
scan 5

Figure 11

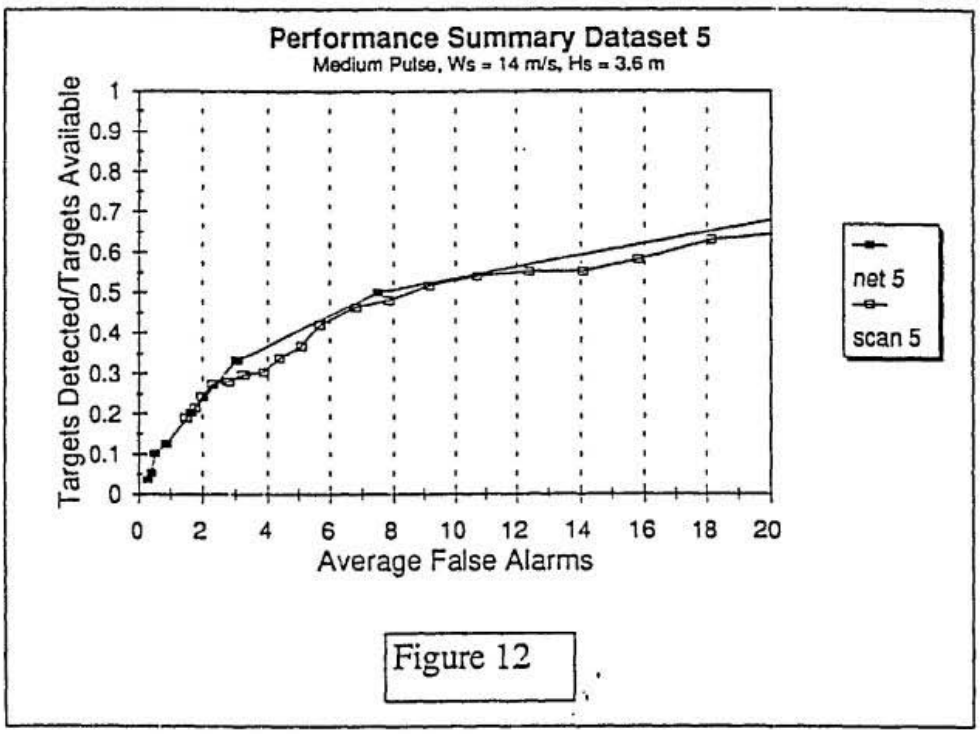

Performance Summary Dataset 5
Medium Pulse, Ws = 14 m/s, Hs = 3.6 m
Targets Detected/Targets Available
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
16
18
20
Average False Alarms
net 5
scan 5

Figure 12

with the radar set to short pulse. It can be seen (Figure 13) that the overall ability to extract target information out of the sea clutter is further improved, and once again the neural technique shows a significant improvement over the scan to scan processor. At the 50% detection level the neural processor presents a false alarm figure of 2.5 on average per scan while the scan to scan technique shows a false alarm figure which is more than double that of the network.

The only real limitation to the implementation of the artificial neural network technique is the processing power that it requires. A complete scan of radar data would occupy 1024 pixels by 1024 lines by 8 bits. In order to apply the neural network in its current form it must be scanned over the entire image stepping one pixel at a time. This means that for a neural net which uses the 6 pixel by 10 line by 5 scan data array, it would have to execute 1018 x 1014 times and each time it would have to process 15,050 connections. The total number of connections that would have to be made is 15,535,392,600 in order to process the entire 5 scan set of data for one scan of output. This huge number of required computations could not practically be implemented in real-time using an artificial neural network. However, it is possible using a real analog neural network. The Intel 80170NX (ETANN) is a silicon chip level implementation of a 64 neuron, 10240 synapse neural network (Figure 14). The chip has a data propagation delay of at most 3 microseconds. The chip can simultaneously compute the dot product of a 64 element analog input vector with a 64 by 64 synaptic array, which corresponds to a processing rate in excess of 1.3 billion interconnections per second. The ETANN can also be used in a multi-chip configuration. In fact, the ETANN is available in a board level

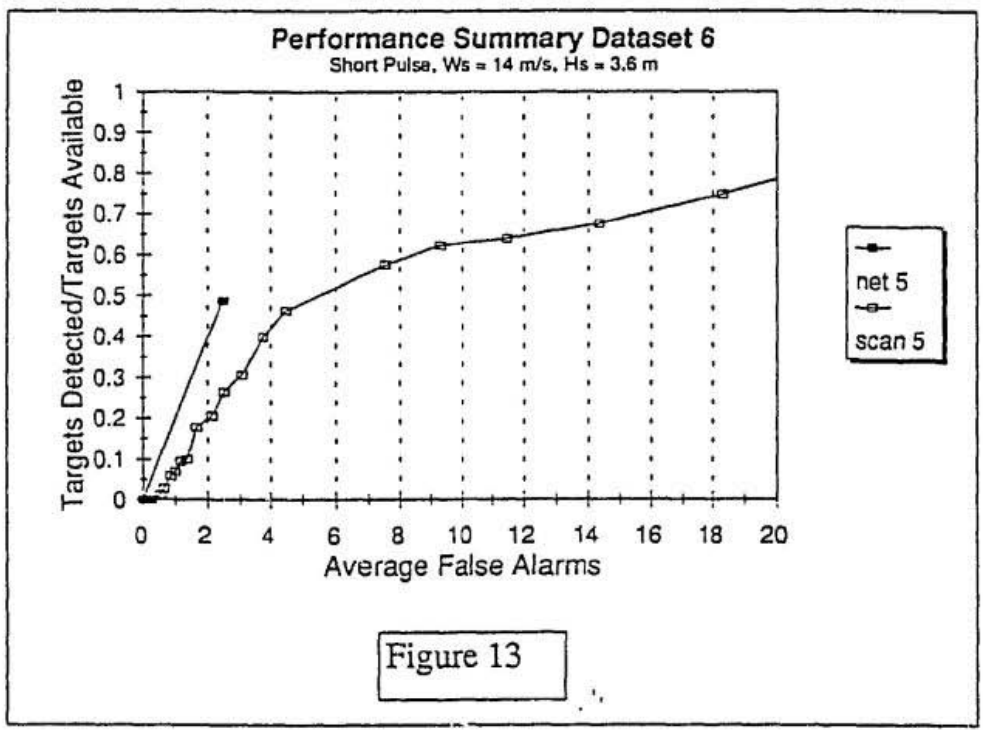

Performance Summary Dataset 6
Short Pulse, Ws = 14 m/s, Hs = 3.6 m
Targets Detected/Targets Available
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
16
18
20
Average False Alarms
net 5
scan 5

Figure 13

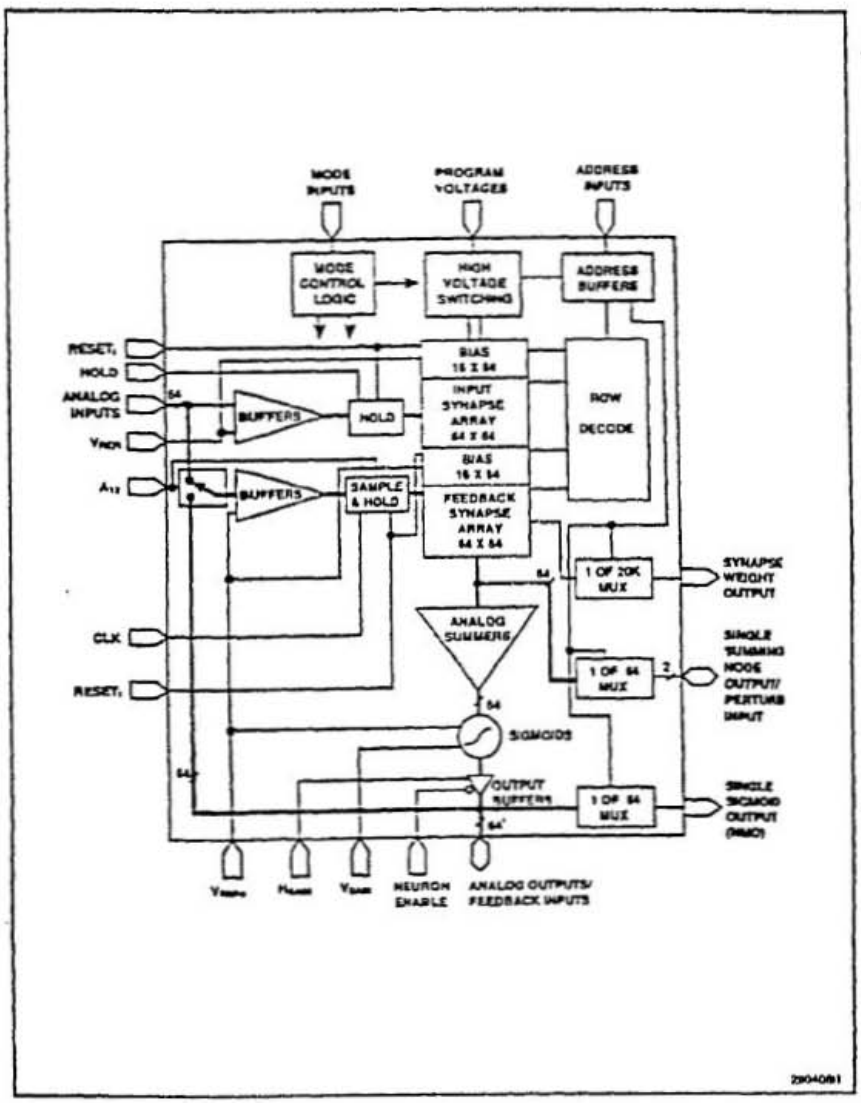


Figure 14. Chip Layout for Intel 80170NX Neural Processor

implementation from Intel. The Intel multi-chip prototyping board (EMB) is a hardware system designed for rapid prototyping of large high speed neural networks. The board may accommodate up to 8 ETANN chips and provides an IBM PC AT interface card. Similar to biological systems the analog ETANN chip suffers from component to component variations and relatively low precision. However, the chip can be trained successfully. The chip-in-loop (CIL) concept was developed and demonstrated on the ETANN chip (Tam, Gupta, Castro, and Hollar, 1990). Using this approach the ETANN outputs are loaded back to the training simulator to determine the optimum weight updates. The standard training method then integrates any imperfections or minor defects that may be present on the chip into the neural network weight architecture, which then becomes specific to that particular ETANN chip or set of chips. This concept has been extended to the multi-chip environment where differences from chip to chip may exist (Tam, Hollar, Brauch, Pine, Peterson, Anderson, and Deiss, 1992). In order to implement the neural network developed here, which is a 300-50-1 network, it is convenient to adopt an architecture which uses a single ETANN per window of radar data (i.e. 60 neurons per ETANN) together with another ETANN used to merge the neural signals (Figure 15). This 6 chip network would have a capacity of approximately 7.8 billion connections per second. The time required to obtain a new full scan of radar data is about 2.3 seconds, depending on the radar scanner speed. Therefore, this implementation of the network could easily accommodate the 15.5 billion connection real-time requirement. In fact, in the 2.3 seconds it takes the scanner

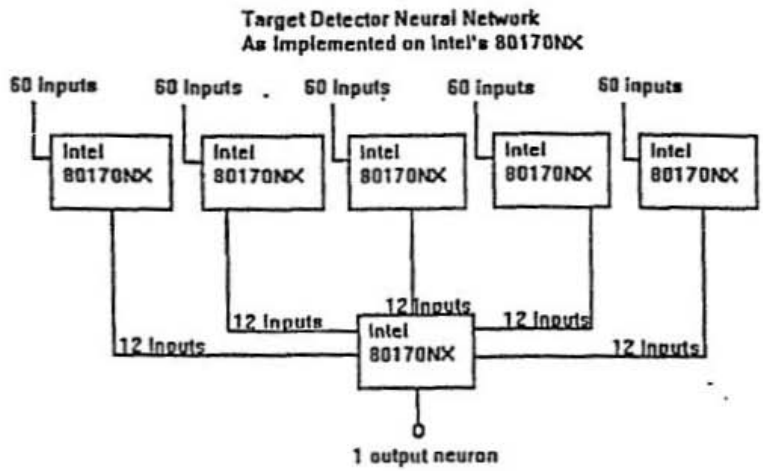

Target Detector Neural Network
As Implemented on Intel's 80170NX
60 Inputs
60 Inputs
60 Inputs
60 Inputs
60 Inputs
Intel 80170NX
Intel 80170NX
Intel 80170NX
Intel 80170NX
Intel 80170NX
12 Inputs
12 Inputs
12 Inputs
12 Inputs
12 Inputs
Intel 80170NX
1 output neuron

Figure 15

to complete a single rotation, this network has a capacity of almost 18 billion connections.

7.0 CONCLUSIONS

A set of neural network synaptic architectures have been developed and applied to the marine radar small target detection problem. The neural network technique has been compared to the conventional processing method of scan to scan integration with results which favor the neural processing. The neural net clearly equals or outperforms the conventional method on all datasets analyzed. A strategy for the realization of a neural radar, with superior detection ability in the ocean environment, has been presented.

Data from four different days have been used to train and test the neural network, covering the full range of wind and wave heights encountered. The results of the analysis indicate:

1. A neural network is capable of providing performance that is at least as good as, scan to scan integration, one of the more powerful conventional processing techniques. If the neural net scanning window is optimized for the pulse length being used the performance is much better.

2. It appears that when the network is trained on the higher sea state data it is capable of generalizing to lower sea states. The reverse is not the case. It is not known at this point if one network will be adequate for the full range of sea states to be encountered by a radar.

3. Comparison of results from different radar pulse lengths indicates that it may be necessary to train the network for a particular set of radar parameters.

A neural network shows great promise in the application of radar target detection. In order to proceed with further development it is necessary to review the potential of the neural network to provide some benefit over conventional processing techniques. These benefits may be in cost, performance or versatility. The results of this study indicate that performance and versatility may be key benefits of the neural network. The implementation cost of the present neural network today is in the order of $10,000 to $20,000. However, there is great potential for lower costs in the next couple of years. The performance improvements that have been demonstrated are significant in that these represent cases that may not yet be optimized for radar parameters and environmental condition. Even if the results presented are the best that may be achieved by the network it is important to note that it performs as good as or better than one of the most successful conventional techniques available, scan to scan integration. The added benefit of the network architecture chosen is that it may provide built-in capability to handle moving targets. This ability in itself could justify the use of the network over other techniques. Present signal processors must carefully align consecutive radar scans before performing scan to scan processing. When the radar is moving this requires that the radar data be converted to a cartesian grid and realigned using vessel position data. This process requires complex hardware and is subject to some losses. If the target

is moving then the number of scans that can be processed will be limited by target speed. On the other hand, the inherent parallel processing nature of the neural network may permit the use of the data without realignment and because the network essentially processes the data in all directions at once it should be able to handle moving targets.

A neural network processor for radar target detection may not fit into the traditional concept of the radar display, which is also required for navigation proposes. This may limit the neural processors market to applications that require improved target detection performance that complements the capabilities of the navigation or vessel traffic services radar. The neural processor embodies all the features required in a high performance tracking radar system and as a result it would be appropriate for vessels and systems requiring the ability to detect and track many targets.

In effect the neural radar concept would be to provide a target detection engine for use with more traditional radar display designs.

REFERENCES

Amari, S. I. 1990. "Mathematical Foundations of Neurocomputing," IEEE Proc. 78(9): 1443-1463.

Dawe, D. and D. Bryant, 1987. "Search and Rescue Detection Experiment". Prepared for Transport Canada.

Fahlman, S. E. and C. Lebiere, 1991. "The Cascade-Correlation Learning Architecture", in D. S. Touretzky (ed.), Advances in Neural Information Processing Systems 2, Morgan Kaufmann.

Gibson, G. J. and C. Cowan, 1990. "On the Decision Regions of Multilayer Perceptrons," IEEE Proc. 78(10): 1590-1594.

Grossberg, S. 1982. Studies of Mind and Brain: Neural Principles of Learning Perception, Development, Cognition, and Motor Control. Boston: Reidell Press.

Hebb, D. 0. 1949. The Organization of Behavior, a Neuropsychological Theory. New York: John Wiley.

Hecht-Nielsen, R. 1987. "Counterpropagation Networks," Appl. Opt. 26(23): 4979-4984.

Kohonen,T. 1984. Self-Organization and Associative Memory. Berlin: Springer Verlag.

McClelland, T. L., D. E. Rumelhart, and the PDP Research Group. 1986. Parallel Distributed Processing. Cambridge: The MIT Press.

Minsky, M. and S. Papert, 1969. Perceptrons. Cambridge, Mass.: MIT Press.

Reed, R., 1993. "Pruning Algorithms - A Survey," IEEE Trans. Neural Networks, 4(5): 740-747.

Rosenblatt, F. 1958. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," Psych. Rev. 65: 386-408.

Ryan, J., 1992. "Iceberg Detection Using Microwave Radar".

Ryan, J., 1990. "Modelling Radar Sea Clutter". Prepared for Defence Research Establishment Ottawa, December 1990, SSC Contract W7714-9-9257/01-ST.

Tam, S. and M. Hollar, Brauch, Pine, Peterson, Anderson, and Deiss, 1992. "A Reconfigurable Multi-Chip Analog Neural Network; Recognition and Back-Propagation Training", Intel Corporation.

Tam, S. and B. Gupta, H. Castro, M. Holler, 1990. "Learning on an Analog VLSI Neural Network Chip", Proceedings of the 1990 IEEE International Conference on Systems, Man & Cybernetics.

Werbos, P. J. 1974. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Doctoral Dissertation, Appl. Math., Harvard University, Mass.

Widrow, B. 1962. "Generalization and Information Storage in Networks of Adaline 'Neurons'," in Self-organizing Systems. M. C. Jovitz, G. T. Jacobi, and G. Goldstein. eds., Washington, D.C.: Spartan Books, 435-461.

Zurada, J. 1992. Artificial Neural Systems. St. Paul, MN: West Publishing Company.

BIBLIOGRAPHY

Burden, R. and J. D. Faires, A. C. Reynolds, 1981. Numerical Analysis. Boston Mass.: Prindle Weber and Schmidt.

Holler, M. and S. Tam, H. Castro, R. Benson, 1989. "An Electrically Trainable Artificial Neural Network (ETANN) with 10240 'Floating Gate' Synapses", International Joint Conference on Neural Networks, Washington, D.C.

Intel Corporation, June 1991.Experimental 80170NX data sheet.

Skolnik, M. 1990. Radar Handbook, Second Edition. McGraw Hill Publishing Company.

Appendix I

Environmental Data

The Directional Waverider Buoy record format is as follows:

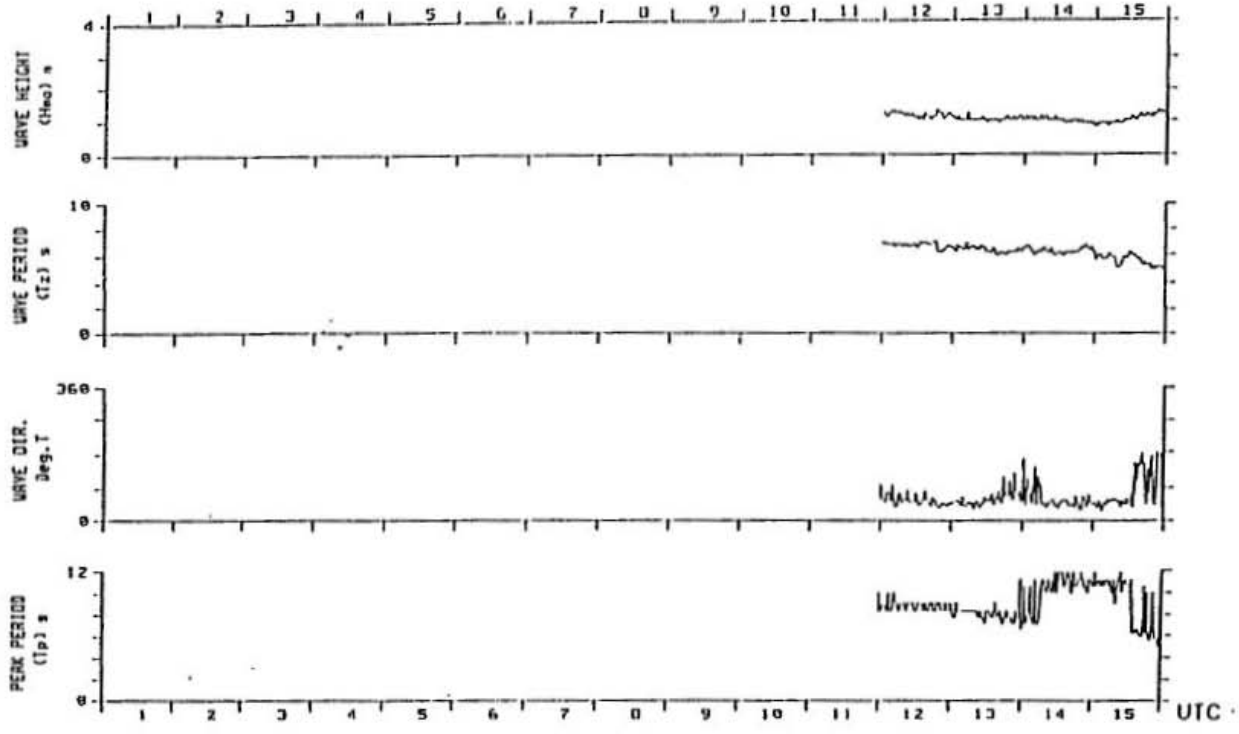
Record Position	Description
1 - 25	Unused.
26 - 33	Date of data sample collection, stored in YYYYMMDD format.
34 - 39	Time of day data sample collected, stored in HHMMSS format. All times expressed in UTC.
40 - 46	Wave Height - Hmo. (m)
47 - 53	Wave Period - Tz. (sec)
54 - 60	Direction Associated with the Peak Wave Period. (Degrees True)
61 - 67	Peak Wave Period - Tp. (sec)
68 - 132	Unused

SITE NAME: SIGMA RADAR GROUND TRUTHING
LATITUDE: 47° 32' 43" N
LONGITUDE: 52° 33' 31" W

CALIBRATED, QUALITY CONTROLLED

Prepared by Seaconsult Limited
9.36 AUG 6, 1993

INSTRUMENT ID: 30004/16 TYPE: DIRECTIONAL WAVERIDER BUOY
DELTA T: 30 min

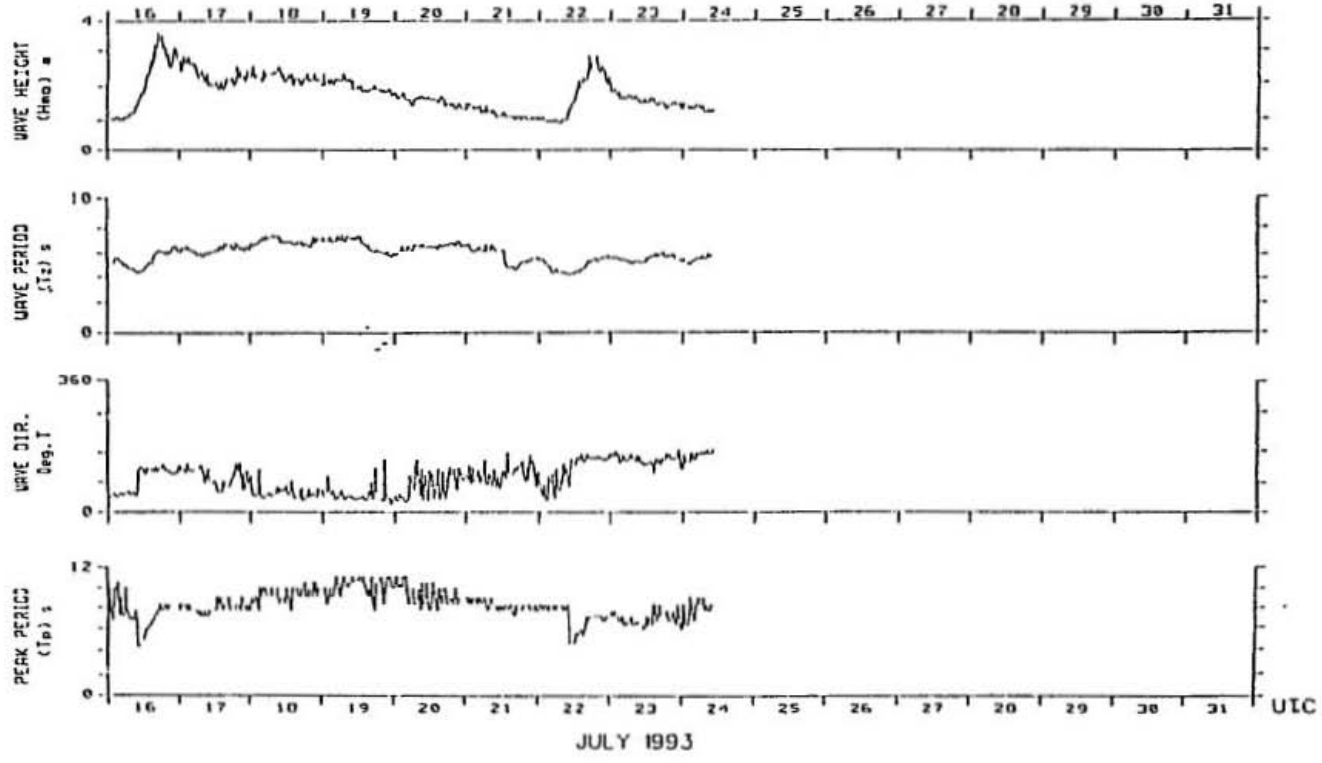


SITE NAME: SIGMA RADAR GROUND TRUTHING
LATITUDE: 47° 32' 43" N
LONGITUDE: 52° 33' 31" W

CALIBRATED, QUALITY CONTROLLED

Prepared by Seaconsult Limited
9:36 AUG 6, 1993

INSTRUMENT ID: 30004/16 TYPE: DIRECTIONAL WAVERIDER BUOY
DELTA T: 30 min



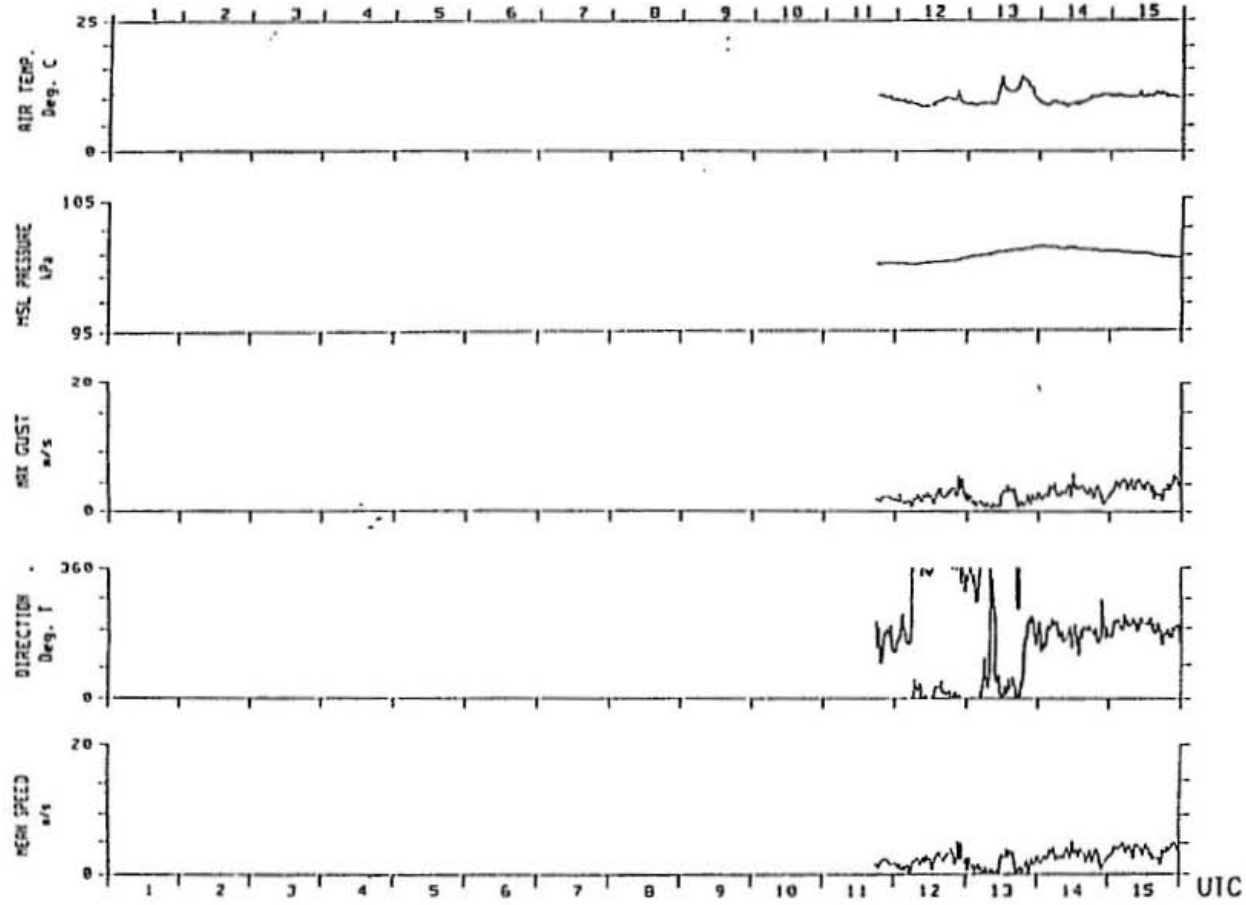
The WEATHERPAK 100 weather station record format is as follows:

Record Position	Description
1 - 25	Unused.
26 - 33	Date of data sample collection, stored in YYYYMMDD format.
34 - 39	Time of day data sample collected, stored in HHMMSS format. All times are UTC.
40 - 46	Air Temperature (°C).
47 - 53	Mean Sea Level Pressure (mbar).
54 - 60	Maximum Wind Speed, Gust. (m/s).
61 - 67	Mean Wind Direction. (Degrees True).
68 - 74	Mean Wind Speed. (m/s).
75 - 81	N-S component of the Mean wind speed and direction, (m/s).
82 - 88	E-W component of the Mean wind speed and direction, (m/s).
89 - 95	Sigma Theta, Standard deviation of the Wind Direction (°).
96 - 102	Weather station battery power (Volts).
103 - 109	Internal temperature (°C).
110 - 123	Unused.
124 - 132	Sequential number assigned to each sample collected during the weather station deployment.

SITE NAME: SIGMA RADAR GROUND TRUTHING
LATITUDE: 47° 32' 43" N
LONGITUDE: 52° 33' 31" W
HEIGHT ABOVE SEA LEVEL: 4 m
METER ID: 207/16 METER TYPE: WEATHERPAK-100
DELTA T: 15 min

CALIBRATED, QUALITY CONTROLLED

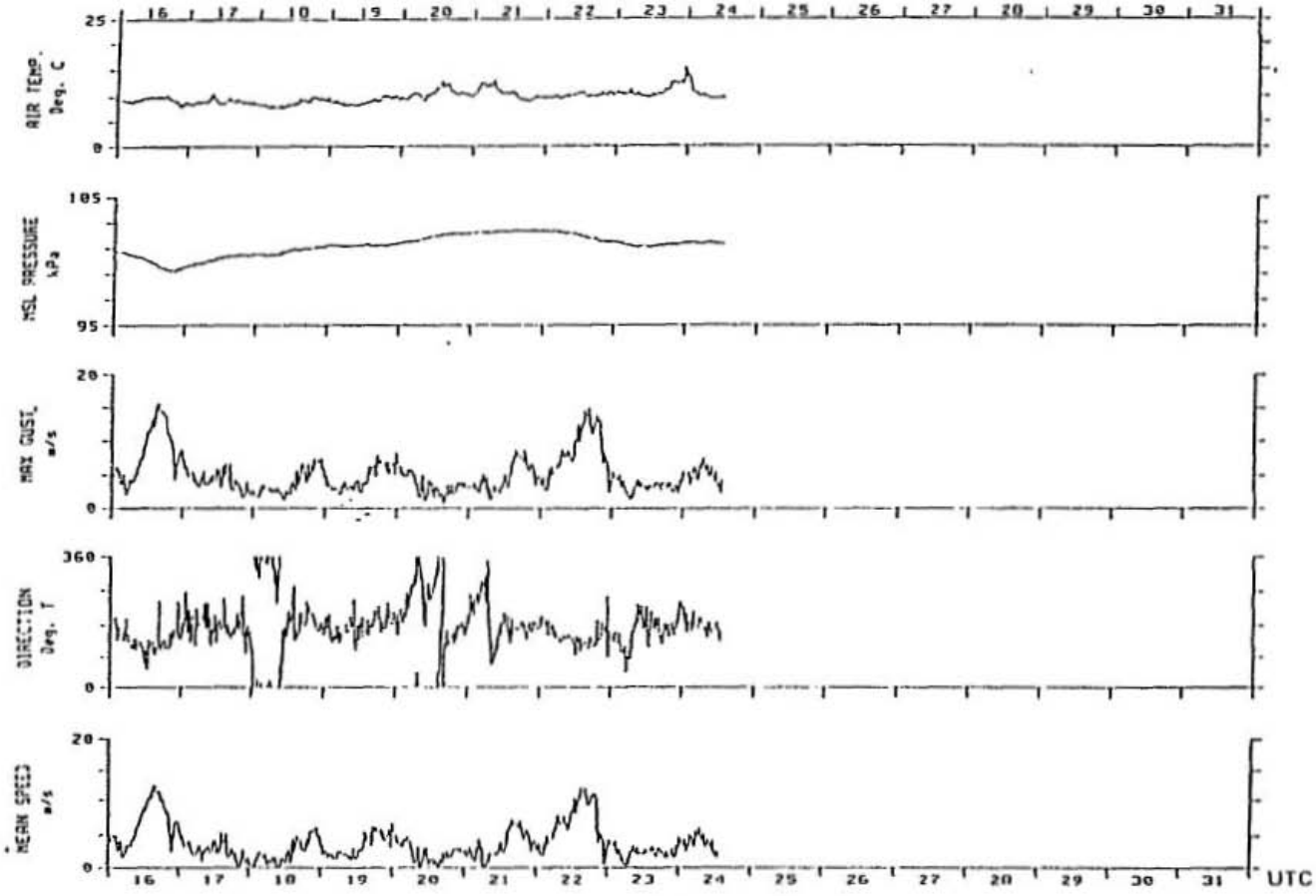
Prepared by Seaconsult Limited
15:34 AUG 5, 1993



SITE NAME: SIGMA RADAR GROUND TRUTHING
LATITUDE: 47° 32' 43" N
LONGITUDE: 52° 33' 31" W
HEIGHT ABOVE SEA LEVEL: 4 m
METER ID: 207/16 METER TYPE: WEATHERPAK-100
DELTA T: 15 min

CALIBRATED, QUALITY CONTROLLED

Prepared by Seaconsult Limited
15:34 AUG 5, 1993

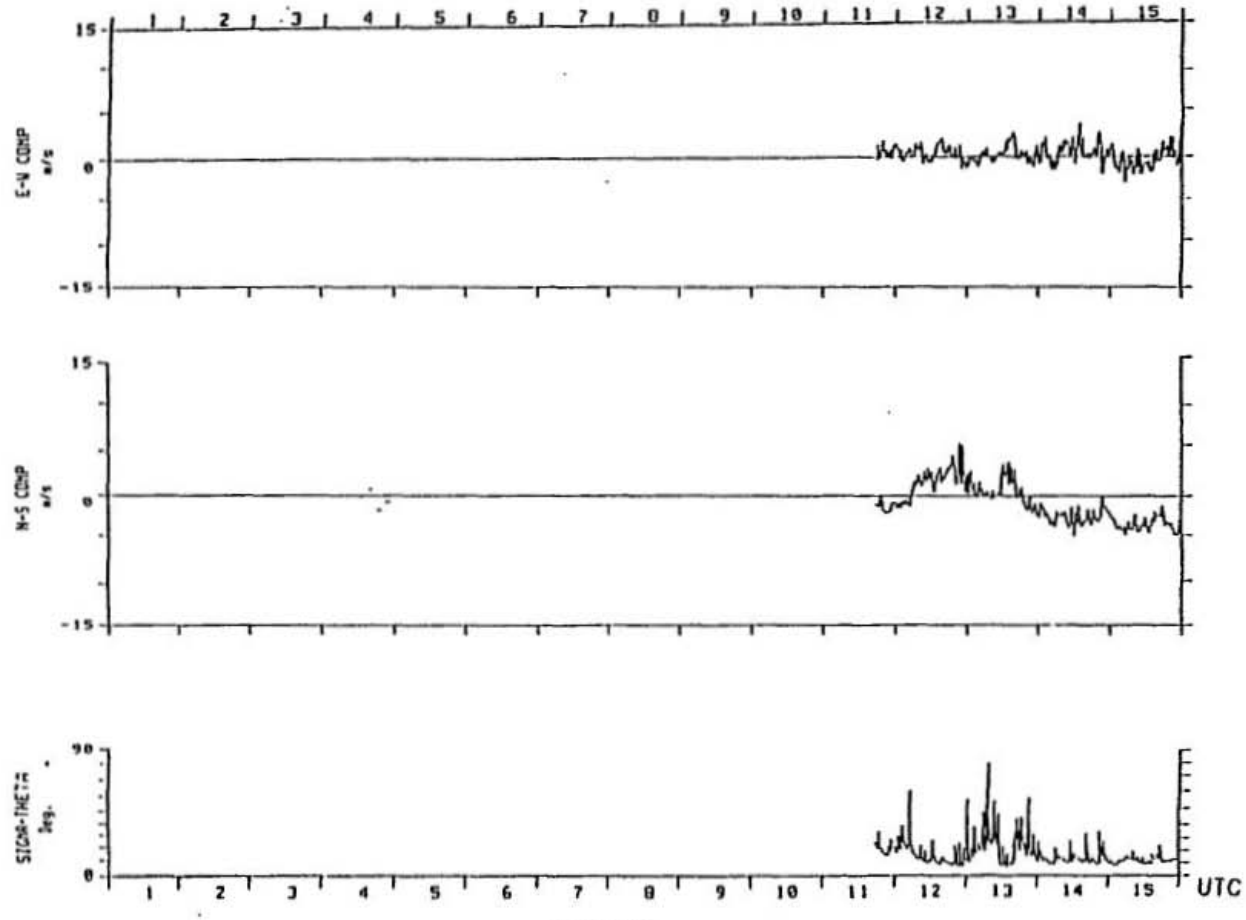


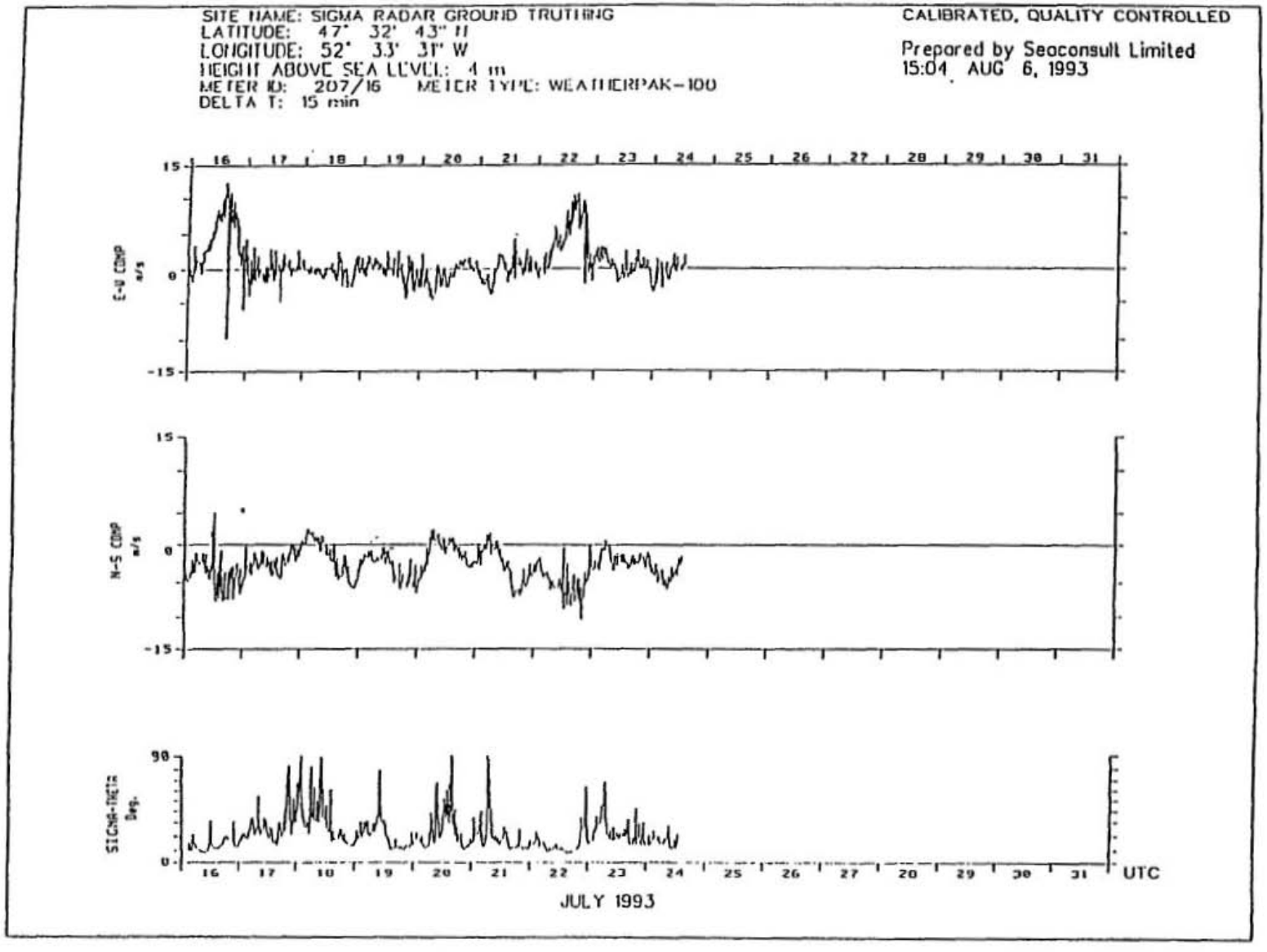
JULY 1993

SITE NAME: SIGMA RADAR GROUND TRUTHING
LATITUDE: 47° 32' 43" N
LONGITUDE: 52° 33' 31" W
HEIGHT ABOVE SEA LEVEL: 4 m
METER ID: 207/16 METER TYPE: WEATHERPAK-100
DELTA T: 15 min

CALIBRATED, QUALITY CONTROLLED

Prepared by Seaconsult Limited
15:04 AUG 6, 1993




SITE NAME: SIGMA RADAR GROUND TRUTHING
LATITUDE: 47° 32' 43" N
LONGITUDE: 52° 33' 31" W
HEIGHT ABOVE SEA LEVEL: 4 m
METER ID: 207/16 METER TYPE: WEATHERPAK-100
DELTA T: 15 min
CALIBRATED, QUALITY CONTROLLED
Prepared by Seaconsult Limited
15:04 AUG 6, 1993
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
15
0
-15
E-W COMP m/s
N-S COMP m/s
SIGMA-THETA Deg.
90
0
UTC
JULY 1993

Appendix II

Software Listings

```
/*Program Neural Engine*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <float.h>
#define BYTE unsigned char
#define gl_ndim 1
#define gl_isign 1
#define Num_waves 4
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
#include "netparm.000"

float         ww[(NN-M)+N+1][NN+1];
float         fxww[(NN-M)+N+1][NN+1];
float         x[LSIZ],complex_data[2*(M+1)];
float         yhat[N+1],myhat[TVART+1][N+1];
float         data[TVART+1][M+1],maxval;
float         desiro[TVART+1][N+1];
float         normax[M+1],normax_result[N+1];
unsigned int    ythr[N+1],nnd[gl_ndim+1];

//wavelet transform

typedef struct {
        int ncof,ioff,joff;
        float *cc,*cr;
} wavefilt;

wavefilt wfilt;

#define NRANSI
#include "nrutil.h"

void pwt(float a[], unsigned long n, int isign)
{
        float ai,ai1,wksp[M];
        unsigned long i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;

        if (n < 4) return;

        nmod=wfilt.ncof*n;
        n1=n-1;
        nh=n >> 1;
        for (j=1;j<=n;j++) wksp[j]=0.0;
        if (isign >= 0) {
                for (ii=1,i=1;i<=n;i+=2,ii++) {
                        ni=i+nmod+wfilt.ioff;
                        nj=i+nmod+wfilt.joff;
                        for (k=1;k<=wfilt.ncof;k++) {
                                jf=n1 & (ni+k);
                                jr=n1 & (nj+k);
                                wksp[ii] += wfilt.cc[k]*a[jf+1];
```

```
                        wksp[ii+nh] += wfilt.cr[k]*a[jr+1];
                    }
                }
        } else {
                for (ii=1,i=1;i<=n;i+=2,ii++) {
                    ai=a[ii];
                    ai1=a[ii+nh];
                    ni=i+nmod+wfilt.ioff;
                    nj=i+nmod+wfilt.joff;
                    for (k=1;k<=wfilt.ncof;k++) {
                        jf=(n1 & (ni+k))+1;
                        jr=(n1 & (nj+k))+1;
                        wksp[jf] += wfilt.cc[k]*ai;
                        wksp[jr] += wfilt.cr[k]*ai1;
                    }
                }
        }
        for (j=1;j<=n;j++) a[j]=wksp[j];
}
#undef NRANSI

void wt1(float a[], unsigned long n, int isign,
        void (*wtx)(float [], unsigned long, int))
{
        unsigned long nn;

        if (n < 4) return;
        if (isign >= 0) {
                for (nn=n;nn>=4;nn>>=1) (*wtx)(a,nn,isign);
        } else {
                for (nn=4;nn<=n;nn<<=1) (*wtx)(a,nn,isign);
        }
}
//end wavelet

void Load_Complex_Data(int t)
{
        int    index=1,i;


        if (FFTflag == 1){
                for (i=1;i<=M;i++){
                        complex_data[index] = data[t][i];
                        complex_data[index+1] = 0.0;
                        index = index + 2;
                }
        }
        if (FFTflag == 2){
                for (i=1;i<=M;i++){
                        complex_data[index] = data[t][i];
                        index = index + 1;
                }
        }
```

```
}

float   sq(float z)
{
        return (z*z);
}

void complex_abs_val(int t)
{
        int    j,i=1;

        if (FFTflag == 1){
                for (j=1;j<=(2*M);j=j+2)
                data[t][i++] = sqrt(sq(complex_data[j]) + sq(complex_data[j+1]));
        }
        if (FFTflag == 2){
                for (j=1;j<=M;j++)
                        data[t][j] = abs(complex_data[j]);
        }

}



void fftn(float datafft[],int nn[],int ndim,int isign)
{
        int         i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
        int         ibit,idim,k1,k2,n,nprev,nrem,ntot;
        float       tempi,tempr;
        double      theta,wxi;
        double      wpi,wpr,wr,wtemp;

        ntot = 1;
        for (idim=1;idim<=ndim;idim++)
                ntot *= nn[idim];

        nprev = 1;
        for (idim=ndim;idim>=1;idim--){
                n = nn[idim];
                nrem = ntot/(n*nprev);
                ip1 = nprev << 1;
                ip2 = ip1*n;
                ip3 = ip2*nrem;
                i2rev = 1;
                for (i2=1;i2<=ip2;i2+=ip1){
                        if (i2 < i2rev){
                                for (i1=i2;i1<=i2+ip1-2;i1+=2){
                                        for (i3=i1;i3<=ip3;i3+=ip2){
                                                i3rev = i2rev + i3 - i2;
                                                SWAP(datafft[i3],datafft[i3rev]);
                                                SWAP(datafft[i3+1],datafft[i3rev+1]);
                                        }
```

```
                        }
                    }
                    ibit = ip2 >> 1;
                    while (ibit >= ip1 && i2rev > ibit){
                        i2rev -= ibit;
                        ibit >>= 1;
                    }
                    i2rev += ibit;
                }
                ifp1 = ip1;
                while (ifp1 < ip2){
                    ifp2 = ifp1 << 1;
                    theta = isign * 6.28318530717959/(ifp2/ip1);
                    wtemp = sin(0.5*theta);
                    wpr = -2.0*wtemp*wtemp;
                    wpi = sin(theta);
                    wr = 1.0;
                    wxi = 0.0;
                    for (i3=1;i3<=ifp1;i3+=ip1){
                        for (i1=i3;i1<=i3+ip1-2;i1+=2){
                            for (i2=i1;i2<=ip3;i2+=ifp2){
                                k1 = i2;
                                k2 = k1 + ifp1;
                        tempr = wr * datafft[k2] - wxi * datafft[k2+1];
                        tempi = wr * datafft[k2+1] + wxi * datafft[k2];
                                datafft[k2] = datafft[k1] - tempr;
                                datafft[k2+1] = datafft[k1+1] - tempi;
                                datafft[k1] += tempr;
                                datafft[k1+1] += tempi;
                            }
                        }
                        wr=(wtemp=wr)*wpr-wxi*wpi+wr;
                        wxi=wxi*wpr+wtemp*wpi+wxi;
                    }
                    ifp1 = ifp2;
                }
                nprev *= n;
        }
}

void readdata()
{
char    fname[TVART+1][80];
FILE    *f1,*f2,*f3,*fe;
int     Pixel,Line,i,j,t;
unsigned long int     len,count;
char    file[80],date[80],time[80],srate[80],ftype[80],temp[80];
char    infil[70];

f2 = fopen("result.dat","r");
f3 = fopen("train.fil","r");

for (t=1;t<=TVART;t++)
{
```

```
        fscanf(f3,"%s",infil);
        f1 = fopen(infil,"r");

        /*read the header*/

        fgets(file,65,f1);
        fgets(date,65,f1);
        fgets(time,65,f1);
        fgets(srate,65,f1);
        fscanf(f1,"%4d",&Pixel);
        fgets(t1,61,f1);
        fscanf(f1,"%4d",&Line);
        fgets(t2,61,f1);
        fgets(ftype,65,f1);
        fgets(temp,65,f1);

        printf("\n Reading> %s\n",infil);
        printf("\n Pixels > %d",Pixel);
        printf("\n Lines  > %d\n",Line);

        count = 0;
        for (i=1;i<=M;i++)
        {
                fscanf(f1,"%c",&data[t][i]);
                count++;
        }
        printf(" Bytes Read into Buffer> %d\n",count);
        for (i=1;i<=N;i++)
        {
                fscanf(f2,"%f",&desiro[t][i]);
        }

        fclose(f1);
}

fclose(f2);
fclose(f3);

}

void readsignal()
{
char            fname[TVART+1][80];
FILE            *f1,*f2,*f3,*fe;
int             Pixel,Line,i,j,t;
unsigned long int    len,count;
char            infil[70];

f2 = fopen("result.dat","r");
f3 = fopen("train.fil","r");

for (t=1;t<=TVART;t++)
{
        fscanf(f3,"%s",infil);
```

```
        f1 = fopen(infil,"r");

        /*read the header information*/

        printf("\n Reading> %s\n",infil);
        printf("\n Bytes > %d\n",M);
        count = 0;

        /* read the training pattern data */

        for (i=1;i<=M;i++)
        {
                fscanf(f1,"%f",&data[t][i]);
                count++;
        }
        printf("\n Bytes Read into Buffer> %d\n",count);


        /* get the result for the pattern just read in */

        for (i=1;i<=N;i++)
        {
                fscanf(f2,"%f ",&desiro[t][i]);
        }

        fclose(f1);
}

fclose(f2);
fclose(f3);

}

int ii(int imod)
{
        return(imod-M);
}

void fxnetwork(fxyhat)

        float  fxyhat[N+1];
        {

        int     i,j,n,nn,m;
        float   fxnet[TN+1];
        float   fxx[TN+1];

        n = N;
        nn = NN;
        m = M;

        for (i=1;i<=nn;i++) fxx[i] = 0.0;

        for (i=1;i<=n;i++) fxx[i+nn] = fxyhat[i];
```

```
/* for i running backwards now (backpropagation) */

        for (i=nn+n;i>=nn+1;i--)
        {
                fxnet[i] = fxx[i]*x[i]*(1.0-x[i]);

                for (j=m+1;j<=nn;j++)
                {
                        fxww[ii(i)][j] = fxnet[i]*x[j];
                }

        }
        for (i=nn;i>=m+1;i--)
        {
                for (j=nn+1;j<=nn+n;j++)
                {
                        fxx[i] = fxx[i] + ww[ii(j)][i]*fxnet[j];
                }

                fxnet[i] = fxx[i]*x[i]*(1.0-x[i]);

                for (j=1;j<=m;j++)
                {
                        fxww[ii(i)][j] = fxnet[i]*x[j];
                }

        }

        }

        void network(ax,t)

        int             t;
        float   ax[TVART+1][M+1];
        {

        unsigned int    i,j,n,nn,m;
        float           net;

        n = N;
        nn = NN;
        m = M;

        /*insert inputs*/
        for (i=1;i<=m;i++)
        {
                x[i] = ax[t][i];
        }

        for (i=m+1;i<=nn;i++)
        {
                net = 0.0;
                for (j=1;j<=m;j++)
```

```
        {
                net = net + ww[ii(i)][j]*x[j];
        }

        x[i] = 1.0/(1.0+exp(-net*LAMDA));
}

for (i=nn+1;i<=nn+n;i++)
{
        net = 0.0;
        for (j=m+1;j<=nn;j++)
        {
                net = net + ww[ii(i)][j]*x[j];
        }

        x[i] = 1.0/(1.0+exp(-net*LAMDA));
}

for (i=1;i<=n;i++)
{
        yhat[i] = x[i+nn];
}

}

int conver(ax,y,l)

int    l,y[TVART+1][N+1];
float  ax[TVART+1][M+1];
{
int    i,j,k,t,n,nn,m,tvart,conflg,check;
float  limit,ub,lb;


limit = 0.10;
ub = 0.90;
lb = 0.10;
k = 1;
n = N;
nn = NN;
m = M;
tvart = TVART;

for (i=m+1;i<=nn+n;i++)
{
        for (j=1;j<=i-1;j++)
        {
                if (fxww[ii(i)][j] > limit)
                {
                        k = 0;
                }
        }
}
```

```
        for (t=1;t<=tvart;t++)
        {
         network(ax,t);
         check = 0;
         for (i=1;i<=n;i++)
         {
                if (yhat[i] > ub)
                {
                        ythr[i] = 1;
                }
                else
                {
                 if (yhat[i] < lb)
                 {
                        ythr[i] = 0;
                 }
                 else
                 {
                        ythr[i] = 2;
                 }

                 if(ythr[i] != y[t][i])
                 {
                        check=1;
                        goto j99;
                 }
                }
                }
        }
j99:    if ((k == 1) || (check == 0))
        {
                conflg = 1;
        }
        else
        {
                conflg = 0;
        }
return(conflg);
}

void normsignal()
{
        unsigned int    t,i;

        if (Norm_Var_by_Var == 1)
        {

                for (i=1;i<=M;i++) normax[i] = -999999.99;
                for (i=1;i<=N;i++) normax_result[i] = -999999.99;
                for (t=1;t<=TVART;t++)
                {
                        for (i=1;i<=M;i++)
                        {
                                normax[i] = max(normax[i],data[t][i]);
```

```
                }
                for (i=1;i<=N;i++)
                {
                normax_result[i] = max(normax_result[i],desiro[t][i]);
                }
        }
        }
        else
        {
                maxval = -999999.99;
                for (i=1;i<=N;i++) normax_result[i] = -999999.99;
                for (t=1;t<=TVART;t++)
                {
                        for (i=1;i<=M;i++)
                        {
                                maxval = max(maxval,data[t][i]);
                        }
                        for (i=1;i<=N;i++)
                        {
                        normax_result[i] = max(normax_result[i],desiro[t][i]);
                        }
                }
                for (i=1;i<=M;i++) normax[i] = maxval;
                printf("\n Signal MAX: %f\n",maxval);
        }
}


main()

{
char        fn1[25],fn2[25],datebuf[9],timebuf[9];
int         l,conflg,setfl,z,wc;
float       y[TVART+1][N+1];
float       ax[TVART+1][M+1],fxyhat[N+1],lxrate;
unsigned int   n=N,nn=NN,m=M,tvart=TVART,tn=TN;
unsigned int   i,j,maxpas,passnm,t,tend;
time_t      t5,t6;
FILE        *fx,*fin1,*fin2,*fin3,*fin4;


        printf("\n");
        printf(" <Optimizing neural connections please standby...> \n");
        printf("\n");

        fin1 = fopen("neuron.dat","r");
        fscanf(fin1,"%f %d",&lxrate,&maxpas);
        fclose(fin1);
        fin4 = fopen("norm.000","w");

        /* read the training data into memory*/
        if (BinaryData == 1)
        {
                readdata();
```

```
}
else
{
        readsignal();
}

/* if FFTflag = 1 do an FFT on the inputs*/
if (FFTflag == 1){
        //setup for FFT if required
        nnd[1] = M;
        for (t=1;t<=TVART;t++){

                Load_Complex_Data(t);
                //calculate fft

                fftn(complex_data,nnd,gl_ndim,gl_isign);
                //calculate power spectrum

                complex_abs_val(t);
        }
}

/* if FFTflag = 2 do an WAVELET TRANSFORM on the inputs*/
if (FFTflag == 2){
        //setup for WAVELET TRANSFORM
        pwtset(Num_waves);

        for (t=1;t<=TVART;t++){
                //calculate WAVELET TRANSFORM
                Load_Complex_Data(t);

                wt1(complex_data,M,gl_isign,pwt);

                complex_abs_val(t);
        }
}


/* perform normalization*/
normsignal();

for (t=1;t<=TVART;t++)
{
        for (i=1;i<=m;i++)
        {
                if (normax[i] != 0.0){
                        ax[t][i] = data[t][i]/normax[i];
                }else{
                        ax[t][i] = 0.0;
                }
                if (t == 1) fprintf(fin4,"%f ",normax[i]);
                /* must normalize the data to be between 0.0 and 1.0 */
        }
        for (i=1;i<=n;i++)
```

```
        {
                y[t][i] = desiro[t][i]/normax_result[i];
                if (t == 1) fprintf(fin4,"%f ",normax_result[i]);
        }
}
fclose(fin4);

srand((unsigned) time(&t5));
l = 0;

for (i=m+1;i<=nn;i++)
{
        for (j=1;j<=m;j++)
        {
                ww[ii(i)][j] = 2.0*rand()/32767.0-1.0;
                l = l + 1;
        }
}

for (i=nn+1;i<=nn+n;i++)
{
        for (j=m+1;j<=nn;j++)
        {
                ww[ii(i)][j] = 2.0*rand()/32767.0-1.0;
                l = l + 1;
        }
}

conflg = 0;
tend = TVART;
setfl = 0;

for (passnm=1;passnm<=maxpas;passnm++)
{

        fin1 = fopen("neuron.dat","r");
        fscanf(fin1,"%f %d",&lxrate,&maxpas);
        fclose(fin1);


        for (t=1;t<=tend;t++)
        {

                network(ax,t);

                for (i=1;i<=n;i++) myhat[t][i] = yhat[i];
                for (j=1;j<=n;j++)
                {
                        fxyhat[j] = yhat[j] - y[t][j];
                }

                fxnetwork(fxyhat);

                for (i=m+1;i<=nn+n;i++)
```

```
                {

                        for (j=1;j<=i-1;j++)
                        {
                                z = ii(i);
                                ww[z][j] = ww[z][j]-lxrate*fxww[z][j];
                        }
                }
        }

        /*enables remote monitoring of process*/

        fx = fopen("monitor","w");
        fprintf(fx,"\n PASS > %d ",passnm);

        for (t=1;t<=tend;t++)
        {
                fprintf(fx,"\n PATTERN> %d",t);
                fprintf(fx,"\n");
        for (i=1;i<=n;i++) fprintf(fx," %f ",myhat[t][i]*normax_result[i]);
                                fprintf(fx,"\n");
        for (i=1;i<=n;i++) fprintf(fx," %f ",y[t][i]*normax_result[i]);
        }
        fprintf(fx,"\n\n");
        fclose(fx);


}

/* write out current weight matrix "ww[i][j]" disk file */
fn2[0] = 'w';
fn2[1] = '.';
fn2[2] = '0';
fn2[3] = '0';
fn2[4] = '0';
fn2[5] = NULL;

wc = 0;
printf("\n The neural weight matrix has been created as> %s\n",fn2);
fin3 = fopen(fn2,"w");

for (i=m+1;i<=nn;i++)
{
        for (j=1;j<=m;j++)
        {
                wc++;
                fprintf(fin3,"%f ",ww[ii(i)][j]);
        }
}

for (i=nn+1;i<=nn+n;i++)
{
        for (j=m+1;j<=nn;j++)
        {
```

```
                wc++;
                fprintf(fin3,"%f ",ww[ii(i)][j]);
        }
}

printf("\n Total Number of Weights is %d",wc);

fclose(fin3);

} /* END */
```

```
/*radarimg.c*/

#include "RADARVAR.h"
#include "RADARIMG.h"

#define HB 80
#define BYTE unsigned char
#define LAMDA 0.10
#define NORM 255.0
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

// defines for directional ocean wave spectra
#define xscale 1
#define yscale 1
#define tscale 1
#define Sline 64
#define Spixel 64
#define Sscans 1
#define SpecScan 1
#define MAXS 128
#define mdo 4000
#define size mdo
#define xoffset 235
#define yoffset 285
#define gl_ndim 2
#define gl_isign 1
#define INIT Spixel*Sline*Sscans*gl_ndim

float           huge o[5][mdo+1];
BYTE            huge oflag[mdo+1];

float     sx[MAXS];
float           sy[MAXS];
float           sz[MAXS];
float           huge afft[INIT];
float           huge complex_data[INIT+4];
float           huge plt_data[Spixel][Sline];
float           huge afft_2d[Spixel][Sline];
float           k_inside,k_outside,kx,ky,xstart,ystart;
float           pie, maxt, mint, tempsz;
int             nm1, index = 1, index1 = 1, test_size, nnd[gl_ndim], xxsiz, yysiz;

//for the scan-to-scan average routine

BYTE huge           *pReadAvgBuffer;
HANDLE                    hReadAvgBuffer;
BYTE huge           *pToReadAvgBuffer;
HANDLE                    hImageAvgBuffer;
float huge          *pImageAvgBuffer;
float huge          *pToImageAvgBuffer;
unsigned int  sysres, gdires, useres, memarg;
long int                  memint;
DWORD                           memres;
```

```
HRGN                    hRgn;
char                    sstr[20];
int                     GetMemFlag = 0;
int                     GetScanMemFlag = 0, TextY;
long int                sdismemx,sdismemy;
/* Series evaluation */

int ImageMax, ImageMin;
int GraphicMax, GraphicMin;
float GraphicStep, ImageStep;
float EvalStep;
BOOL EvalSeriesFlag = FALSE;
BOOL labelflag = FALSE;
char GraphFileTemp[128];

/* statistics */
#define TargetNumber 200
#define PixelNumber 400
#define MarkTargetNumber 6
#define MarkPixelNumber 400

int     k;
int     CheckCoord;
int huge        *pPixelRecord;
HANDLE              hPixelRecord;
int huge        *pPixelRecord2;
HANDLE              hPixelRecord2;

// palette stuff
HANDLE      hPalData;
BOOL        PalFlag = FALSE;
HPALETTE hpalImg, hpalOld;
LPLOGPALETTE    lppalData;
LPBITMAPINFO    lpbmInfo;

/* correlation */

int                 Fir1, Fir2;
int huge        *pAddFirRecord1;
HANDLE              hAddFirRecord1;
int huge        *pAddFirRecord2;
HANDLE              hAddFirRecord2;
int huge        *pCoorelate;
HANDLE              hCoorelate;

int     LinesPrinted = 0, Page = 1, YLine;
long int        targetsG, targetsI, temptar;
long int        targetpixels;
int     nFalseAlarmI;
int     nFalseAlarmG;
int     TargetFoundI;
int     TargetFoundG;
float targetsGAvg, targetsIAvg;
float nFalseAlarmIAvg;
```

```
float nFalseAlarmGAvg;
float TargetFoundIAvg;
float TargetFoundGAvg;
char    Ntarstr[50];
char    Ntottar[10];
char    Nfalsealarm[50];
FILE    *don,*sp,*fp3,*fp4;
FILE    *Targetdat;
int     ScanPos;
int     MarkCount = 0;
int     TargetNo=0;
int     arraysize[6];   /* sizes of target position arrays */
int     TargetWin[MarkTargetNumber][MarkPixelNumber];
int     TargetPos[MarkTargetNumber][MarkPixelNumber];
int     PosSubX, PosSubY, CheckPos;
int     CurrPixel;
WORD            MarkX, MarkY, LowLeftX, LowLeftY;
BOOL  AverageEvalFlag = FALSE;
BOOL  LastBatchFlag = FALSE;
BOOL  StatsHeaderFlag = FALSE;
BOOL  ULflag = FALSE;
BOOL  ULflagImage = FALSE;
BOOL  ThresholdAvgFlag = FALSE;
BOOL  CorrelFlag = FALSE;
BOOL  SpectraFlag = FALSE;
int     ImageTar = 0, ImageTarIndex = 0;
/* End of the stats */

char    Decimal[20];
float huge  *lptemp;
float huge  *pScanWinNorm;
HANDLE  hScanWinNorm;
unsigned int huge *pScanWinCopy;
HANDLE hScanWinCopy;
long int neuron,neuronstotal,memory1,memory3,mtarpix;
int  testdon;

char    herex[10];
char    herey[10];
char    FileName[128];
char    scanlabel[128];
char    TempFile[128];
char    TempFile2[128];
char    NextFile[128];
char    PathName[128];
char    OldName[128];
char    OpenName[128];
char    DefPath[128];
char    DefSpec[13];
char    DefExt[4];
char    str[255];
int     ExtNum;
int     TextPosX;
int     TextPosY;
```

```
int     XRgn;
int     YRgn;
int     xhi,yhi,ylo;
int     bytesread;
char    string[30];

/* Global variables */

char    SSize[4];
char    NumberScans[4];
char    NumberLayers[5];
char    NumOutstr[5];
char    evaluation[25];
char    amtnoise[20];
char    amtclutter[20];
char    amttarget[20];
char    Noxx[5],Lcxx[5],Mcxx[5],Hcxx[5],Stxx[5],Ltxx[5],Htxx[5];
char    WindowName[128];
int     count = 0;
char    classstr[20];
int     pernoise,perclutter,pertarget;
int     pernoise1,perclutter1,pertarget1;
int     pernoise2,perclutter2,pertarget2;
int     GraphicSize;
int     GraphicNumber, FileNumber;
long int  memory2;

HWND hWndTarThres,hWndTarLabel,hWndTarVal;
HWND hWndAvgThres,hWndAvgLabel,hWndAvgVal;
int  ThresholdVal = 180;
int  TarVal=305;
float TarValFloat;
float    ThresholdValFloat;
FARPROC lpfnAvgThresInfo;
FARPROC lpfnTarThresInfo;
BOOL AvgDraw = FALSE;
char avgtbuffer[20];
HANDLE hHourGlass;              /* handle to hourglass cursor    */
HANDLE hSaveCursor;             /* current cursor handle         */
static HCURSOR hDonnieCur;
HCURSOR hArrow;
int hFile;                      /* file handle                   */
OFSTRUCT OfStruct;              /* information from OpenFile()   */
OFSTRUCT OfStruct2;
struct stat FileStatus;         /* information from fstat()      */
BYTE huge *pTemp[25];
BYTE huge *pScanBuffer[25];   /* Buffer to store each scan individually */
HANDLE hScanBuffer[25];         /* handle to editing buffer      */
HANDLE hRepBuffer;
BOOL ShowAllFlag = FALSE; /* When flag is false only target bit map is drawn */
BOOL bChanges = FALSE;
BOOL bSaveEnabled = FALSE;
BOOL bNew = TRUE;
BOOL ScanWinAlloc = FALSE;
```

```
BOOL GraphicFlag = FALSE;
BOOL ImageFlag = FALSE;
BOOL ImageDraw = FALSE;
BOOL Batch = FALSE;
BOOL First = FALSE;
BOOL BitMap0 = FALSE;
BOOL ClearAll;
BOOL ShowFileName = TRUE;
BOOL UpdateImage;
BOOL AvgImageFlag;
BOOL NetLoadedFlag;
BOOL Threshold;
BOOL EvalAvg = FALSE;
BOOL MarkTarget, VideoFlag=FALSE; //flag to toggle video on or off
int  BitmapCount=0;

HANDLE hDisplayBuffer[25];
BYTE huge *pDisplayBuffer[25];
BYTE huge *pToDisplay[25];
BYTE huge *ImageBuffer;
int fileoffset;

HANDLE hAverageBuffer;
float huge *pAverageBuffer;
int huge *pToAverage;
BYTE huge *pAverageByte;
HANDLE hAverageByte;
BYTE huge *pToAverageByte;

DWORD available;

BYTE huge *pImageAvgByte;
BYTE huge *pToImageAvgByte;
BYTE huge *pSpecImage;
BYTE huge *pToSpecImage;
HANDLE      hImageAvgByte;
HANDLE      hSpecImage;
int huge *pToImageAvg;
BYTE huge *pToImageByte;

HBITMAP hBitmap[25], hOldBitmap[25], hBitmapGraphic;
HBITMAP hRepaintBitmap, hOldRepaintBitmap, hBitmapImage,
hOldBitmapImage;
HDC hDC, hMemoryDC, StretchhDC;    /* handle for the display device     */
HANDLE hBitInfo;
LPBITMAPINFO pBitInfo;

BOOL bTrack = FALSE;               /* TRUE if left button clicked  */
int OrgX = 0, OrgY = 0;            /* original cursor position     */
int PrevX = 0, PrevY = 0;          /* current cursor position      */

WORD X = 0, Y = 0;                 /* last cursor position         */
RECT Rect;                         /* selection rectangle          */
POINT ptCursor;                    /* x and y coordinates of cursor */
```

```
int repeat = 1;                    /* repeat count of keystroke    */

int VectorSize, VectorDim = 1;
POINT ptPSize;

int ScrX = 0, ScrY=0;    /* source of bitmap rectangle */
int nScrX = 0, nScrY = 0;
int nScanCount;
BOOL EndRow,EndFlag;
RECT test;
int Temp;
int BatchNumber;
int WinInc=1;
int PosDecrement;

/* OrgX and OrgY holds the pixel position to copy. Copy data according to
pixel size and number of scan lines */

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
lpszCmdLine, int nCmdShow)
{
/*****************************************************************/
/* HANDLE hInstance;     handle for this instance                 */
/* HANDLE hPrevInstance;  handle for possible previous instances  */
/* LPSTR  lpszCmdLine;    long pointer to exec command line       */
/* int    nCmdShow;       Show code for main window display       */
/*****************************************************************/

MSG msg;        /* MSG structure to store your messages      */
int nRc;        /* return value from Register Classes        */

strcpy(WindowName,"Smart Radar Display");
strcpy(szAppName, "RADARIMG");
hInst = hInstance;
if(!hPrevInstance){
        /* register window classes if first instance of application    */
        if ((nRc = nCwRegisterClasses()) == -1){
                /* registering one of the windows failed               */
                LoadString(hInst, IDS_ERR_REGISTER_CLASS, szString,
sizeof(szString));
                MessageBox(NULL, szString, NULL, MB_ICONEXCLAMATION);
                return nRc;
                }
        }

/* create application's Main window                                   */
hWndMain = CreateWindow(
                                        szAppName,          /* Window
class name      */
                                        WindowName,         /* Window's
title  */
                                        WS_CAPTION    |     /* Title and
Min/Max     */
```

```
                                        WS_SYSMENU    |     /* Add system
menu box      */
                                        WS_MINIMIZEBOX |    /* Add minimize
box           */
                                        WS_MAXIMIZEBOX |    /* Add maximize
box           */
                                        WS_THICKFRAME  |    /* thick sizeable
frame         */
                                        WS_CLIPCHILDREN |   /* don't draw in
child windows areas */
                                        WS_OVERLAPPED  |
                                        WS_MAXIMIZE,
                                        CW_USEDEFAULT, 0,   /* Use default X,
Y             */
                                        CW_USEDEFAULT, 0,   /* Use default X,
Y             */
                                        NULL,               /* Parent window's
handle        */
                                        NULL,               /* Default to Class Menu
*/
                                        hInst,              /* Instance of window
*/
                                        NULL);              /* Create struct for
WM_CREATE */

 if(hWndMain == NULL){
        LoadString(hInst, IDS_ERR_CREATE_WINDOW, szString,
sizeof(szString));
        MessageBox(NULL, szString, NULL, MB_ICONEXCLAMATION);
        return IDS_ERR_CREATE_WINDOW;
       }

 /**** Threshold Bar for the GRAPHIC window ****/

 hWndTarThres = CreateWindow ("scrollbar", NULL,
                                                    WS_CHILD |
WS_VISIBLE | WS_TABSTOP | SBS_VERT,
GetSystemMetrics(SM_CXSCREEN)-75, 65,
GetSystemMetrics(SM_CXVSCROLL), 300,
                                                    hWndMain, 501,
hInst, NULL) ;

 hWndTarLabel = CreateWindow ("static", "Neural Processor",
                                                    WS_CHILD |
WS_VISIBLE | SS_CENTER,
GetSystemMetrics(SM_CXSCREEN)-105, 20,
                                                    85, 45,
                                                    hWndMain, 502,
hInst, NULL) ;
```

```
hWndTarVal = CreateWindow ("static", "305",
                                                    WS_CHILD |
WS_VISIBLE | SS_CENTER,

GetSystemMetrics(SM_CXSCREEN)-81, 380,
                                                    30, 20,
                                                    hWndMain, 503,
hInst, NULL) ;

lpfnTarThresInfo = (FARPROC) GetWindowLong (hWndTarThres,
GWL_WNDPROC) ;

SetScrollRange (hWndTarThres, SB_CTL, 0, 400, FALSE) ;
SetScrollPos   (hWndTarThres, SB_CTL, TarVal, FALSE) ;

/**** Threshold Bar for the AVERAGED window ****/

hWndAvgThres = CreateWindow ("scrollbar", NULL,
                                                    WS_CHILD |
WS_VISIBLE | WS_TABSTOP | SBS_VERT,

GetSystemMetrics(SM_CXSCREEN)-155, 65,

GetSystemMetrics(SM_CXVSCROLL), 300,
                                                    hWndMain, 504,
hInst, NULL) ;

hWndAvgLabel = CreateWindow ("static", "Scan-To-Scan Integration",
                                                    WS_CHILD |
WS_VISIBLE | SS_CENTER,

GetSystemMetrics(SM_CXSCREEN)-192, 20,
                                                    90, 45,
                                                    hWndMain, 505,
hInst, NULL) ;

hWndAvgVal = CreateWindow ("static", "180",
                                                    WS_CHILD |
WS_VISIBLE | SS_CENTER,

GetSystemMetrics(SM_CXSCREEN)-161, 380,
                                                    30, 20,
                                                    hWndMain, 506,
hInst, NULL) ;

lpfnAvgThresInfo = (FARPROC) GetWindowLong (hWndAvgThres,
GWL_WNDPROC) ;

SetScrollRange (hWndAvgThres, SB_CTL, 0, 255, FALSE) ;
SetScrollPos   (hWndAvgThres, SB_CTL, ThresholdVal, FALSE) ;


hHourGlass = LoadCursor(NULL, IDC_WAIT);
hArrow = LoadCursor(NULL, IDC_ARROW);
```

```
hDonnieCur = LoadCursor(hInst,(LPSTR)"DONNIE");
ShowWindow(hWndMain, SW_SHOWMAXIMIZED);        /* display main window
*/
UpdateWindow(hWndMain);


hAccel = LoadAccelerators(hInst, szAppName);

while(GetMessage(&msg, NULL, 0, 0))  /* Until WM_Quit message   */
        {
        //check system resources & memory


                sysres = GetFreeSystemResources(0x0000);
                gdires = GetFreeSystemResources(0x0001);
                useres = GetFreeSystemResources(0x0002);
                memres = GetFreeSpace(memarg);

                memint = memres/1024;

                hDC = GetDC(hWndMain);
                itoa(sysres, sstr, 10);
                TextOut(hDC, 20, 425, sstr, strlen(sstr));
                itoa(gdires, sstr, 10);
                TextOut(hDC, 50, 425, sstr, strlen(sstr));
                itoa(useres, sstr, 10);
                TextOut(hDC, 80, 425, sstr, strlen(sstr));
                itoa((int)memint, sstr, 10);
                TextOut(hDC, 110, 425, sstr, strlen(sstr));
                TextOut(hDC, 465,400,"<Sensitivity Controls>",22);

                ReleaseDC(hWndMain, hDC);

                if (ImageDraw){
                                GetGraphic(hWndMain);

                                hDC = GetDC(hWndMain);
                                TextOut(hDC, 8,5,FileName,strlen(FileName));
                                TextOut(hDC, 125,5,"Scan-to-Scan",12);
                                TextY = (Line + 19);
                                TextOut(hDC, 3,TextY,"Neural Processor",16);
                                ReleaseDC(hWndMain, hDC);

                        ImageDraw = FALSE;
                }



                if (AvgDraw){
                        SetCursor(hHourGlass);

if(DoTheImageAverageEvaluationForTheSmartRadarProject(hWndMain)){
                                        AvgImageFlag = TRUE;
                                        AvgDraw = FALSE;
```

```
                    }
                    else
                                AvgImageFlag = FALSE;
                    SetCursor(hArrow);
            }

            if(!TranslateAccelerator(hWndMain, hAccel, &msg))
            {   TranslateMessage(&msg);
                                DispatchMessage(&msg);
            }
        }

        /* Do clean up before exiting from the application*/

        CwUnRegisterClasses();
        return msg.wParam;
} /* End of WinMain                                    */

/*****************************************************************/
/*                                                   */
/* Main Window Procedure                                     */
/*                                                   */
/* This procedure provides service routines for the Windows events    */
/* (messages) that Windows sends to the window, as well as the user   */
/* initiated events (messages) that are generated when the user selects */
/* the action bar and pulldown menu controls or the corresponding     */
/* keyboard accelerators.                                    */
/*****************************************************************/

LONG FAR PASCAL WndProc(HWND hWnd, WORD Message, WORD wParam,
LONG lParam)
{
HMENU    hMenu=0;       /* handle for the menu            */
int      nRc=0;         /* return code                    */

FARPROC lpProcAbout, lpOpenDlg, lpSaveAsDlg;

int      Success;       /* return value from SaveAsDlg() */
int      i, j, n;
int      nSIZE;
long int  memory;
char     szbuffer[10];

 switch (Message){
   case WM_COMMAND:

       switch (wParam)
                                {
                                        case IDM_O_SPECTRA_ON:

                                                hMenu = GetMenu(hWnd);

                                                if
(CheckMenuItem(hMenu,IDM_O_SPECTRA_ON,MF_UNCHECKED)
```

```
                    == MF_UNCHECKED)|

CheckMenuItem(hMenu,IDM_O_SPECTRA_ON,MF_CHECKED);
                                                    SpectraFlag = TRUE;
                                                }
                                                else
                                                    SpectraFlag = FALSE;


                                    break;

                                    case IDM_O_REPLAY_DATA:
                        //must replay radar data scans and pass each scanline on
                        //to the FFT function-> then the FFT result must be
displayed
                                    //Call OpenDlg() to get the filename

                                        labelflag = FALSE;
                                        strcpy(DefSpec, "*.*");
                        lpOpenDlg = MakeProcInstance((FARPROC) OpenDlg,
hInst);
                        hFile=DialogBox(hInst, "Open", hWnd, lpOpenDlg);
                                        FreeProcInstance(lpOpenDlg);

                                        if(!hFile)
                                            return (NULL);

                                    Temp = WinInc;
                                        {
                                            FARPROC
lpOverlapDlg;
                lpOverlapDlg = MakeProcInstance((FARPROC)OverlapDlg, hInst);
                nRc = DialogBox(hInst, MAKEINTRESOURCE(820),
hWnd,lpOverlapDlg);

        FreeProcInstance(lpOverlapDlg);
                                        }
                                        if (!nRc)
                                            break;
                                        FileNumber = WinInc;
                                        WinInc = Temp;

                                        Replay_data(hWnd);

                                break;

                                case IDM_F_OPEN:

                                    /* Call OpenDlg() to get the filename
*/
            labelflag = FALSE;
                                        strcpy(DefSpec, "*.*");
            lpOpenDlg = MakeProcInstance((FARPROC) OpenDlg, hInst);
```

```
            hFile=DialogBox(hInst, "Open", hWnd, lpOpenDlg);
            FreeProcInstance(lpOpenDlg);
            if(!hFile)
                                                    return (NULL);
            if (GetFile(hWnd))
               ImageFlag = TRUE;

          break;


        case IDM_T_CAPTURE:

            {
            FARPROC lpCaptureDlg;
            lpCaptureDlg = MakeProcInstance((FARPROC)CaptureDlg, hInst);
                nRc = DialogBox(hInst, MAKEINTRESOURCE(300),
hWnd,lpCaptureDlg);
                                        FreeProcInstance(lpCaptureDlg);
            }
            if (!nRc)
                                                    break;
            hDC = GetDC(hWnd);
            ptPSize.x = ScanSizeX;
            ptPSize.y = ScanSizeY;
                                        ReleaseDC(hWnd,hDC);

                        break;

        case IDM_T_REPAINT:

            ScrX = 0;
            InvalidateRect (hWnd, NULL, TRUE);
            UpdateWindow(hWnd);
        break;


                        case IDM_T_CLEARALL:

                            ClearAll = TRUE;
                            ScrX = 0;
                            nSIZE=((nScanCount)*(Pixel+4));

                            hDC=GetDC(hWnd);
                            hMemoryDC = CreateCompatibleDC(hDC);
                    hRepaintBitmap=CreateCompatibleBitmap(hDC,
nSIZE,nSIZE);
                    hOldRepaintBitmap =
SelectObject(hMemoryDC,hRepaintBitmap);

                    PatBlt(hMemoryDC, 0, 0, nSIZE, nSIZE, WHITENESS);
                    BitBlt(hDC,ScrX,ScrY, Pixel, Line, hMemoryDC, 0,
0,SRCCOPY);
```

```
                    hRepaintBitmap=SelectObject(hMemoryDC,
hOldRepaintBitmap);
                                        DeleteObject(hRepaintBitmap);
                                        DeleteDC(hMemoryDC);
                                        ReleaseDC(hWnd,hDC);

                                        SetCursor(hSaveCursor);
                                        FreeMemory(hWnd);
                                        ClearAll = FALSE;

                    break;

                    case IDM_T_SHOWALL:

        hMenu = GetMenu(hWnd);
              if (CheckMenuItem(hMenu,IDM_T_SHOWALL,MF_UNCHECKED)
                    == MF_UNCHECKED){
          CheckMenuItem(hMenu,IDM_T_SHOWALL,MF_CHECKED);
          ShowAllFlag = TRUE;
        }
        else
                                              ShowAllFlag = FALSE;
        break;

    case IDM_T_AVERAGE:

        SetCursor(hHourGlass);
        if(DoTheImageAverageEvaluationForTheSmartRadarProject(hWnd))
          AvgImageFlag = TRUE;
        else
          AvgImageFlag = FALSE;
        SetCursor(hArrow);

        break;

  case IDM_T_THRESHOLD:

                                    hMenu = GetMenu(hWnd);

          if (CheckMenuItem(hMenu,IDM_T_THRESHOLD,MF_UNCHECKED)
                        == MF_UNCHECKED){
            CheckMenuItem(hMenu,IDM_T_THRESHOLD,MF_CHECKED);
            Threshold = TRUE;
                                              }
          else
            Threshold = FALSE;

                                              break;

            case IDM_T_CORRELATE:

                                              hMenu = GetMenu(hWnd);
```

```
                if
(CheckMenuItem(hMenu,IDM_T_CORRELATE,MF_UNCHECKED)

                        == MF_UNCHECKED){

CheckMenuItem(hMenu,IDM_T_CORRELATE,MF_CHECKED);
                                                CorrelFlng = TRUE;
                                        }
                                        else
                                                CorrelFlag = FALSE;

                break;


                case IDM_F_SAVE:
            /* If there is no filename, use the saveas command to get
                one.  Otherwise, save the file using the current
                filename. */

             if (bNew)
                goto saveas;
             if (bChanges)
                SaveFile(hWnd);
              break;

             case IDM_F_SAVEAS:

           saveas:

                strcpy(TempFile,FileName);
                lpSaveAsDlg = MakeProcInstance(SaveAsDlg, hInst);
            /* Call the SaveAsDlg() function to get the new filename */
                Success = DialogBox(hInst, "SaveAs", hWnd, lpSaveAsDlg);
                FreeProcInstance(lpSaveAsDlg);


                if (Success == IDOK)
                    SaveFile(hWnd);
                strcpy(FileName,TempFile);

           break;

                        case IDM_F_EXIT:
                                FreeMemory(hWnd);

           break;

          case IDM_F_ABOUTRID:

             lpProcAbout = MakeProcInstance(About, hInst);
                                DialogBox(hInst, "AboutBox", hWnd,
lpProcAbout);
             FreeProcInstance(lpProcAbout);
           break;
```

```
case IDC_EDIT:

        if (HIWORD (lParam) == EN_CHANGE)
            bChanges = TRUE;
        return (NULL);


      case IDM_N_CLICK:

          hMenu = GetMenu(hWnd);

       if (CheckMenuItem(hMenu,IDM_N_CLICK,MF_UNCHECKED)

                          == MF_UNCHECKED){

CheckMenuItem(hMenu,IDM_N_CLICK,MF_CHECKED);
                                                    Click = TRUE;
                                                }
                                                else
                                                    Click = FALSE;

                                                break;


                        case IDM_N_NETLOAD:

                                                strcpy(str,"*.*");
                                                {
                                                FARPROC lpfnNETLOADMsgProc;
lpfnNETLOADMsgProc =
MakeProcInstance((FARPROC)NETLOADMsgProc,hInst);
nRc = DialogBox(hInst, MAKEINTRESOURCE(100),
hWnd,lpfnNETLOADMsgProc);
                                                FreeProcInstance(lpfnNETLOADMsgProc);
                                                }
                                                if (nRc == FALSE)
                                                        break;
                                                GraphicFlag = TRUE;
                                                ptPSize.x = ScanSizeX;
                                                ptPSize.y = ScanSizeY;


                                                if (pScanWinData != NULL){
                                                        GlobalUnlock(hScanWinData);
                                    pScanWinData = (BYTE huge
*)GlobalFree(hScanWinData);
                                                        ScanWinAlloc = FALSE;
                                                }

                                                if (NumScans > ScanScans){
                                            memory = (long int)
ScanSizeX*ScanSizeY*(NumScans+1);
                                                }else{
```

```
                                memory = (long int)
ScanSizeX*ScanSizeY*(ScanScans+1);
                                }

        if ((hScanWinData = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                                                        memory)) == NULL){
                                MessageBox(hWnd, "Memory Allocation Error.",
                                "Error 8.1", MB_OK | MB_ICONHAND);
                                        break;
                                }

                if ((pScanWinData = (BYTE huge *)GlobalLock(hScanWinData)) ==
NULL){
        MessageBox(hWnd, "Global Lock Failed.", "Error 9", MB_OK |
MB_ICONHAND);
                                        break;
                                }
                                else
                                        ScanWinAlloc = TRUE;

                                SetCursor(hDonnieCur);
                                nettst(hWnd);

                                break;

                                case IDM_N_EVALUATEIMAGE:

                                        First = TRUE;
                                        Evaluate(hWnd);
                                        First = FALSE;
                                 hRgn =
CreateRectRgn(0,20,Pixel+1,Pixel+21);
                                        InvalidateRgn(hWnd,hRgn,TRUE);
                                        DeleteObject(hRgn);

                                break;

                                case IDM_N_EVALAVG:

                                        hMenu = GetMenu(hWnd);

                        if
(CheckMenuItem(hMenu,IDM_N_EVALAVG,MF_UNCHECKED)

                                == MF_UNCHECKED){

CheckMenuItem(hMenu,IDM_N_EVALAVG,MF_CHECKED);
                                                        EvalAvg = TRUE;
                                                }
                                                else
                                                        EvalAvg = FALSE;
```

```
                                        break;

                                        case IDM_N_LOADGRAPHIC:

                                                labelflag = TRUE;

                                                strcpy(str,"?????out.*");
                                                {
                                        FARPROC lpfnLOADGRAPHICMsgProc;
lpfnLOADGRAPHICMsgProc =
MakeProcInstance((FARPROC)LOADGRAPHICMsgProc, hInst);
nRc = DialogBox(hInst, MAKEINTRESOURCE(600),
hWnd,lpfnLOADGRAPHICMsgProc);

        FreeProcInstance(lpfnLOADGRAPHICMsgProc);
                                                if (!nRc)
                                                        break;
                                                if(!GetGraphic(hWnd))
                                                        break;
                                                        GraphicFlag = TRUE;
                                                }
                                                GetWinCoord(hWnd);

                                                if(!GetFile(hWnd))
                                                        break;

                                                if
(DoTheImageAverageEvaluationForTheSmartRadarProject(hWnd))

AvgImageFlag=TRUE;
                                                else

AvgImageFlag=FALSE;

                                                ptPSize.x = ScanSizeX;
                                                ptPSize.y = ScanSizeY;

                                                break;

                                        case IDM_N_BATCH:

                        lpOpenDlg = MakeProcInstance((FARPROC) OpenDlg,
hInst);
                        hFile=DialogBox(hInst, "Open", hWnd, lpOpenDlg);
                                        FreeProcInstance(lpOpenDlg);
                                        if (!hFile)
                                                break;
                                        Batch = TRUE;
                                        ShowFileName = FALSE;
                                        Temp = WinInc;
                                        {
                                        FARPROC lpOverlapDlg;
                        lpOverlapDlg = MakeProcInstance((FARPROC)OverlapDlg,
hInst);
```

```
                nRc = DialogBox(hInst, MAKEINTRESOURCE(810),
hWnd,lpOverlapDlg);
                                        FreeProcInstance(lpOverlapDlg);
                                        }
                                        if (!nRc)
                                                break;
                                        BatchNumber = WinInc;
                                        WinInc = Temp;
                                        ImageFlag = TRUE;
                                        First = TRUE;


                                        for (i=1;i<=BatchNumber;i++){
                                                if (!GetFile(hWnd))
                                                        break;
                        sprintf(string,"Processing %d of %d:", i,
BatchNumber);

                                                Evaluate(hWnd);

                                                First = FALSE;
                                                GetNextFile(1);
                                                strcpy(FileName,
NextFile);
                                        }
                                        Batch = FALSE;
                                        ShowFileName = TRUE;

                                        break;

                                case IDM_N_REPLAY:

                                        labelflag = TRUE;

                                        strcpy(str,"?????out.*");
                                        Batch=TRUE;
                                        LastBatchFlag = FALSE;
                                        {
                                FARPROC lpfnLOADGRAPHICMsgProc;
lpfnLOADGRAPHICMsgProc
=MakeProcInstance((FARPROC)LOADGRAPHICMsgProc, hInst);
nRc = DialogBox(hInst, MAKEINTRESOURCE(600),
hWnd,lpfnLOADGRAPHICMsgProc);

        FreeProcInstance(lpfnLOADGRAPHICMsgProc);
                                        }
                                        if (!nRc)
                                                break;
                                        Temp = WinInc;
                                        {
                                                FARPROC
lpOverlapDlg;
                        lpOverlapDlg = MakeProcInstance((FARPROC)OverlapDlg,
hInst);
```

```
                nRc = DialogBox(hInst, MAKEINTRESOURCE(820),
hWnd,lpOverlapDlg);

        FreeProcInstance(lpOverlapDlg);
                                                        }
                                                        if (!nRc)
                                                                break;
                                                        GraphicNumber = WinInc;
                                                        WinInc = Temp;
                                                        Replay(hWnd);
                                                        ptPSize.x = ScanSizeX;
                                                        ptPSize.y = ScanSizeY;

                                        break;

                                        case IDM_N_PERCENTOVERLAP:
                                                        {
                                                                FARPROC
lpOverlapDlg;
                        lpOverlapDlg =
MakeProcInstance((FARPROC)OverlapDlg,hInst);
                nRc = DialogBox(hInst, MAKEINTRESOURCE(800),
hWnd,lpOverlapDlg);

FreeProcInstance(lpOverlapDlg);
                                                        }
                                                        if (!nRc)
                                                                break;
                                                        ScrX=0;
                                                        PrevX =0;
                                                        OrgX = ScanSizeX;
                                                        PrevY = Line+20-ScanSizeY;
/*Changed Imagesize to Line */

                                        break;

                                        case IDM_S_MARK:

for(i=0;i<MarkTargetNumber;i++)

        for(j=0;j<MarkPixelNumber;j++)

                        TargetPos[i][j] = -1;

                for(i=0;i<MarkTargetNumber;i++)
                        for(j=0;j<MarkPixelNumber;j++)
                                        Win[i][j] = -1;

                                        nScanWindows =
nXScanWindows*nYScanWindows;
                                                        PosSubX=ScanSizeX/2;
                                                        PosSubY=ScanSizeY/2;

        CheckPos=Pixel*PosSubY+PosSubX;
```

```
                                        hMenu = GetMenu(hWnd);
if (CheckMenuItem(hMenu,IDM_S_MARK,MF_UNCHECKED)==
MF_UNCHECKED){

        CheckMenuItem(hMenu,IDM_S_MARK,MF_CHECKED);
                                                MarkTarget = TRUE;
                                                TargetNo = 0;
                                                ImageTar = 0;
                                                ImageTarIndex = 0;
                                        }
                                        else
                                                MarkTarget = FALSE;

                                break;


                                case IDM_S_GO:

                                        if (!DoStats(hWnd)){
MessageBox(hWnd, "Stats not done.","Error",MB_OK | MB_ICONEXCLAMATION);
                                                        break;
                                        }

                                        if (!WriteStats(hWnd)){
MessageBox(hWnd, "Error writing Stats.", "Error",MB_OK |
MB_ICONEXCLAMATION);
                                                        break;
                                        }

                                        if (!SaveStats(hWnd)){
MessageBox(hWnd, "Error savings stats file.", "Error",MB_OK |
B_ICONEXCLAMATION);
                                                        break;
                                        }

                                        if (pPixelRecord != NULL){

        GlobalUnlock(hPixelRecord);
        pPixelRecord = GlobalFree(hPixelRecord);
                                        }

                                        if (pPixelRecord2 != NULL){

        GlobalUnlock(hPixelRecord2);
        pPixelRecord2 = GlobalFree(hPixelRecord2);
                                        }


                                break;

                                case IDM_S_IMAGEMAX:

                                        ImageMax =    ThresholdVal;
```

```
                                        break;

                                        case IDM_S_IMAGEMIN:

                                                ImageMin = ThresholdVal;

                                        break;

                                        case IDM_S_GRAPHICMAX:

                                                GraphicMax = TarVal;

                                        break;

                                        case IDM_S_GRAPHICMIN:

                                                GraphicMin = TarVal;

                                        break;

                                        case IDM_S_STEPS:

                                                Temp = WinInc;
                                                {
                                                FARPROC lpOverlapDlg;
                        lpOverlapDlg = MakeProcInstance((FARPROC)OverlapDlg,
hInst);
                nRc = DialogBox(hInst, MAKEINTRESOURCE(840),
hWnd,lpOverlapDlg);

        FreeProcInstance(lpOverlapDlg);
                                                }
                                                if (!nRc)
                                                        break;
                                                EvalStep = (float)WinInc;
                                                WinInc = Temp;

                                        break;

                                        case IDM_S_EVALSERIES:

                                                strcpy(str,"?????out.*");
                                                Batch=TRUE;
                                                LastBatchFlag = FALSE;
                                                {
                                        FARPROC lpfnLOADGRAPHICMsgProc;
lpfnLOADGRAPHICMsgProc
=MakeProcInstance((FARPROC)LOADGRAPHICMsgProc, hInst);
nRc = DialogBox(hInst, MAKEIN'TRESOURCE(600),
hWnd,lpfnLOADGRAPHICMsgProc);

        FreeProcInstance(lpfnLOADGRAPHICMsgProc);
                                                }
```

```
                                        if (!nRc)
                                                break;
                                        Temp = WinInc;
                                        {
                                                FARPROC
lpOverlapDlg;
                        lpOverlapDlg = MakeProcInstance((FARPROC)OverlapDlg,
hInst);
                nRc = DialogBox(hInst, MAKEINTRESOURCE(820),
hWnd,lpOverlapDlg);

        FreeProcInstance(lpOverlapDlg);
                                        }
                                        if (!nRc)
                                                break;
                                        GraphicNumber = WinInc;
                                        WinInc = Temp;
                        GraphicStep = (float)(GraphicMax-
GraphicMin)/EvalStep;
                        ImageStep = (float)(ImageMax-ImageMin)/EvalStep;
                                        TarVal = GraphicMax;
                                        TarValFloat = (float)TarVal;
                                        ThresholdVal = ImageMin;
                                ThresholdValFloat = (float)ThresholdVal;
                                        EvalSeriesFlag = TRUE;
                                strcpy(GraphFileTemp, GraphFileName);
                                        for (i=0;i<=EvalStep;i++){

        SetScrollPos(hWndTarThres,SB_CTL,TarVal,TRUE);

        SetScrollPos(hWndAvgThres,SB_CTL,ThresholdVal,TRUE);
                                                itoa (TarVal, szbuffer,
10);
                                        SetWindowText
(hWndTarVal,szbuffer);
                                                itoa (ThresholdVal, avgtbuffer,
10);
                                        SetWindowText
(hWndAvgVal,avgtbuffer);
                                        strcpy(GraphFileName,
GraphFileTemp);
                        if (TarVal>=GraphicMin &&
ThresholdVal<=ImageMax)
                                                        Replay(hWnd);
                                        TarValFloat = TarValFloat-
GraphicStep;
                                                TarVal =
(int)TarValFloat;
                                ThresholdValFloat =
ThresholdValFloat+ImageStep;
                                ThresholdVal = (int)ThresholdValFloat;
                                        }
                                        EvalSeriesFlag = FALSE;
                                        SummarizeData("stats.dat");
```

```
                                        ptPSize.x = ScanSizeX;
                                        ptPSize.y = ScanSizeY;

                                    break;

                                    default:
                                return DefWindowProc(hWnd, Message, wParam,
lParam);

                        }

                        break;      /* End of WM_COMMAND                    */

        case WM_CREATE:
                    /* The WM_CREATE message is sent once to a window when
the    */
                    /* window is created. The window procedure for the new
window */
                     /* receives this message after the window is created, but
*/
                     /* before the window becomes visible.                   */

                                    NumScans = 5;
                                    ScanSizeX = 9;
        ScanSizeY = 9;
        ptPSize.x = ScanSizeX;
        ptPSize.y = ScanSizeY;


    hMenu = GetMenu(hWnd);
    EnableMenuItem(hMenu, 0, MF_BYPOSITION);

                    break;     /* End of WM_CREATE                     */

 case WM_MOVE:   /* code for moving the window              */

    break;

 case WM_SIZE:   /* code for sizing client area             */

                     break;     /* End of WM_SIZE                       */

 case WM_VSCROLL:

     n = GetWindowWord (HIWORD (lParam), GWW_ID) ;
                    if (n==501){
       switch (wParam)
          {
              case SB_PAGEDOWN :
                TarVal += 10 ;

             case SB_LINEDOWN :
```

```
                    TarVal = min (400, TarVal + 1) ;
                    break ;

                 case SB_PAGEUP :
                    TarVal -= 10 ;

                 case SB_LINEUP :
                 TarVal = max (0, TarVal - 1) ;
                    break ;

                  case SB_TOP:
                    TarVal = 0 ;
                    break ;

                 case SB_BOTTOM :
                    TarVal = 400 ;
                    break ;

                 case SB_THUMBPOSITION :
                 case SB_THUMBTRACK :
                     TarVal = LOWORD (lParam) ;

                     break ;

                              default :
                     break ;
             }

            SetScrollPos (hWndTarThres, SB_CTL, TarVal, TRUE) ;
                                                itoa (TarVal, szbuffer, 10);
                                                SetWindowText
(hWndTarVal,szbuffer);
                                                ImageDraw = TRUE;
                         }

                         else{
        switch (wParam)
            {
                 case SB_PAGEDOWN :
                    ThresholdVal += 10 ;

                 case SB_LINEDOWN :
                    ThresholdVal = min (255, ThresholdVal + 1) ;
                    break ;

                 case SB_PAGEUP :
                    ThresholdVal -= 10 ;

                 case SB_LINEUP :
                    ThresholdVal = max (0, ThresholdVal - 1) ;
                    break ;

                  case SB_TOP:
                    ThresholdVal = 0 ;
```

```
            break ;

        case SB_BOTTOM :
            ThresholdVal = 255 ;
            break ;

        case SB_THUMBPOSITION :
        case SB_THUMBTRACK :
            ThresholdVal = LOWORD (lParam) ;

            break ;

        default :
            break ;
    }

    SetScrollPos (hWndAvgThres, SB_CTL, ThresholdVal, TRUE) ;
    itoa (ThresholdVal, avgtbuffer, 10);
    SetWindowText (hWndAvgVal,avgtbuffer);
                                        AvgDraw = TRUE;
                }
 return 0;


  case WM_LBUTTONDOWN:

if(!ImageFlag || UpdateImage) && !MarkTarget)
   break;
bTrack = TRUE;
strcpy (str, "");

if (MarkTarget){
   MarkX = LOWORD(lParam);
   MarkY = HIWORD(lParam);
   /* changed Imagesize to pixel and line */
   if (MarkX>Pixel || MarkY<20 || MarkY>Line+20){
      MarkCount=0;
                                        return 1;
   }
   MarkCount++;

   if (MarkCount == 1){
                                        LowLeftX = MarkX;
      LowLeftY = MarkY;
      return 1;
                              }
   else{
      MarkCount = 0;
      hDC = GetDC(hWnd);
      MoveTo(hDC, LowLeftX, LowLeftY);
      LineTo(hDC, MarkX, LowLeftY);
      LineTo(hDC, MarkX, MarkY);
      LineTo(hDC, LowLeftX, MarkY);
      LineTo(hDC, LowLeftX, LowLeftY);
```

```
            ReleaseDC(hWnd, hDC);

               test_size = (MarkX-LowLeftX)*(MarkY-LowLeftY);

                if(test_size > MarkPixelNumber){
                  MessageBox(hWnd, "Target Area is Too Large",
                   "Try again",MB_OK | MB_ICONEXCLAMATION);
                                                                        break;
                                                }

                                                {
                                                FARPROC lpMarkDlg;
                        lpMarkDlg = MakeProcInstance((FARPROC)MarkDlg,
hInst);
                    nRc = DialogBox(hInst, MAKEINTRESOURCE(830), hWnd,
lpMarkDlg);
                                                FreeProcInstance(lpMarkDlg);
                                                }

                                                if(!nRc)
                                                        MarkTarget =
FALSE;
                                                return 1;
                                        }
                                }

                                PrevX = LOWORD(lParam);
                                PrevY = HIWORD(lParam);

                                if (!(wParam & MK_SHIFT)){
                                        OrgX = LOWORD(lParam);
                                        OrgY = HIWORD(lParam);
                                }

                                /* Get the current mouse position */
                                OrgX += ptPSize.x;
                                OrgY += ptPSize.y;

                                /* changed Imagesize to pixel and line */
                                if (OrgX>Pixel || OrgY<20+ScanSizeY || OrgY>Line+20)
                                        break;

                                hDC = GetDC(hWnd);
                                MoveTo(hDC, OrgX, OrgY);
                                LineTo(hDC, OrgX, PrevY);
                                LineTo(hDC, PrevX, PrevY);
                                LineTo(hDC, PrevX, OrgY);
                                LineTo(hDC, OrgX, OrgY);
                                ReleaseDC(hWnd, hDC);

                                fileoffset = (Line+ScrY-OrgY)*Pixel + (OrgX-ScanSizeX);
                                /* changed Imagesize to pixel and line */
                                if(AvgImageFlag)
                                        ScrX =2*(Pixel+20);
```

```
                    else
                                ScrX = Pixel+20;
                    ScrY = 20;

                    if(!PunchThru(hWnd))
                                break;

                    if (!Click)
                                break;
                    StoreScanWin(fileoffset,hWnd);
                    blocktst();
                    WriteImageClass();
                    DisplayText(hWnd);

                    break;

          case WM_MOUSEMOVE:
                                {
                                RECT     rectClient;
                                int      NextX;
                                int      NextY;

          if (bTrack) {
                                                NextX = LOWORD(lParam);
             NextY = HIWORD(lParam);

                /* Do not draw outside the window's client area */
             GetClientRect (hWnd, &rectClient);
                                                if (NextX < rectClient.left) {
                NextX = rectClient.left;
             } else if (NextX >= rectClient.right) {
                                                                NextX =
rectClient.right - 1;
             }
             if (NextY < rectClient.top) {
                NextY = rectClient.top;
             } else if (NextY >= rectClient.bottom) {
                NextY = rectClient.bottom - 1;
             }

             if ((NextX != PrevX) || (NextY != PrevY))
             {
            /* Get the current mouse position */
                PrevX = NextX;
                PrevY = NextY;

                                                }
          }
       }
       break;

                    case WM_RBUTTONDOWN:

          if(!GraphicFlag)
```

```
                                    break;
    bTrack = FALSE;       /* No longer creating a selection */
  bChanges = TRUE;        /* Saves the current value    */
    X = LOWORD(lParam);
    Y = HIWORD(lParam);
    /* changed Imagesize to pixel and line */
    xhi = GraphicSize*nYScanWindows;
    yhi = Line+40+GraphicSize*nYScanWindows;
    ylo = Line+40;
  if(X > xhi || Y < ylo || Y > yhi)
    break;
                                    XRgn = Pixel+1;
                                    YRgn = Line+21;
    if (AvgImageFlag){
                                          XRgn = 3*Pixel+41;
                                          YRgn = Line+21;
    }
  GetImageBlock(X,Y);
    SetScanWin(hWnd);
                                    DisplayText(hWnd);
    /* changed Imagesize to pixel and line */
    if(AvgImageFlag)
                                          ScrX =2*(Pixel+20);
    else
       ScrX = Pixel+20;
    ScrY=20;

    if(!PunchThru(hWnd))
       break;

break;

case WM_LBUTTONUP:

    bTrack = FALSE;       /* No longer creating a selection */
    bChanges = TRUE;
  X = LOWORD(lParam);       /* Saves the current value     */
  Y = HIWORD(lParam);

    break;


case WM_ACTIVATE:
    if (!GetSystemMetrics(SM_MOUSEPRESENT))
                                    {
       if (!HIWORD(lParam))
       {
          if (wParam)
          {
              ptCursor.x = X;
              ptCursor.y = Y;
              ClientToScreen(hWnd, &ptCursor);
              SetCursorPos(ptCursor.x, ptCursor.y);
          }
```

```
            ShowCursor(wParam);
        }
    }
    break;

                case WM_PAINT:
                {
                                        PAINTSTRUCT     ps;
                                        hDC = BeginPaint(hWnd, &ps);


                                        if (hBitmapImage != NULL){
                                                // hDC =
GetDC(hWnd);
                                        hMemoryDC =
CreateCompatibleDC(hDC);
                                        SelectObject(hMemoryDC,
hBitmapImage);
BitBlt(hDC,0,20,Pixel,Line,hMemoryDC, 0,0,SRCCOPY);
DeleteDC(hMemoryDC);
//ReleaseDC(hWnd,hDC);
                                        }
                                        if (pImageAvgByte != NULL){
                                                //hDC =
GetDC(hWnd);
                                        hMemoryDC =
CreateCompatibleDC(hDC);
                                        SelectObject(hMemoryDC,
hImageAvgByte);
BitBlt(hDC,120,20,Pixel,Line,hMemoryDC, 0,0,SRCCOPY);
DeleteDC(hMemoryDC);
//ReleaseDC(hWnd,hDC);
                                        }
                                        if (pSpecImage != NULL){
                                        hMemoryDC =
CreateCompatibleDC(hDC);
                                        SelectObject(hMemoryDC,
hSpecImage);
BitBlt(hDC,240,20,xxsiz,yysiz,hMemoryDC, 0,0,SRCCOPY);
DeleteDC(hMemoryDC);
                                        }

                                        for (i=0;i<nScanCount;i++)
```

```
                                                        if (hBitmap[i] !=
NULL){
                                                    //hDC = GetDC(hWnd);
                                                hMemoryDC =
CreateCompatibleDC(hDC);

        SelectObject(hMemoryDC, hOldBitmap[i]);

        BitBlt(hDC,ScrX,ScrY,Pixel,Line,hMemoryDC,0,0,SRCCOPY);

        DeleteDC(hMemoryDC);

        //ReleaseDC(hWnd,hDC);
                                                                }

        if (OrgX != PrevX || OrgY != PrevY){

        //hDC=GetDC(hWnd);
                        MoveTo(hDC, OrgX, OrgY);
                        LineTo(hDC, OrgX, PrevY);
                        LineTo(hDC, PrevX, PrevY);
                        LineTo(hDC, PrevX, OrgY);
                        LineTo(hDC, OrgX, OrgY);

        //ReleaseDC(hWnd,hDC);
                                                                }
                                                EndPaint (hWnd, &ps);

         }
                                        break;

              case WM_CLOSE: /* close the window                    */
              /* Destroy child windows, modeless dialogs, then, this window */
              FreeMemory(hWnd);

         break;

         default:
         /* For any message for which you don't specifically provide a */
         /* service routine, you should return the message to Windows  */
         /* for default message processing.                    */
         return DefWindowProc(hWnd, Message, wParam, lParam);
   }
 return 0L;
}   /* End of WndProc */


/* This function loads the file from FileName */

int GetFile(HWND hWnd){

         FILE *fl;
         char date[HB],file[HB],time[HB],t2[HB-4];
         char temp[HB],srate[HB],ftype[HB],t1[HB-4];
```

```
        HANDLE   hBuffer;
        PSTR     pBuffer;       /* address of read/writ buffer   */
        PSTR     pToBuffer;

        int bytesread, r;
        long IOStatus = 0, bytesmoved;     /* result of file i/o     */
        int i,j,n;
        float  freq;


        /*read the header first */
        if ((f1 = fopen(FileName,"r"))==NULL){
       MessageBox(hWnd, "Image File error","Error 20", MB_OK |
MB_ICONHAND);
                    return NULL;
        }

        if (ShowFileName){

                        //check system resources & memory

                    sysres = GetFreeSystemResources(0x0000);
                    gdires = GetFreeSystemResources(0x0001);
                    useres = GetFreeSystemResources(0x0002);
                    memres = GetFreeSpace(memarg);

                    memint = memres/1024;

                    hDC = GetDC(hWnd);

                    TextOut(hDC, 8,5,FileName,strlen(FileName));
                    if (labelflag) TextOut(hDC, 125,5,"Scan-to-Scan",12);
                    TextY = (Line + 19);
                    if (labelflag) TextOut(hDC, 3,TextY,"Neural Processor",16);
                    TextOut(hDC, 465,400,"<Sensitivity Controls>",22);

                    itoa(sysres, sstr, 10);
                    TextOut(hDC, 20, 425, sstr, strlen(sstr));
                    itoa(gdires, sstr, 10);
                    TextOut(hDC, 50, 425, sstr, strlen(sstr));
                    itoa(useres, sstr, 10);
                    TextOut(hDC, 80, 425, sstr, strlen(sstr));
                    itoa((int)memint, sstr, 10);
                    TextOut(hDC, 110, 425, sstr, strlen(sstr));

                    ReleaseDC(hWnd, hDC);

        }

        fgets(file,65,f1);
        fgets(date,65,f1);
        fgets(time,65,f1);
        fgets(srate,65,f1);
```

```
        fscanf(f1,"%4d",&Pixel);
        fgets(t1,61,f1);
        fscanf(f1,"%4d",&Line);
        YLine = Line;
        fgets(t2,61,f1);
        fgets(ftype,65,f1);
        fgets(temp,65,f1);
        fclose(f1);

        MemSizeX = Pixel;
        MemSizeY = Line;
        while (MemSizeX%4)
                        MemSizeX++;
        while (MemSizeY%4)
                        MemSizeY++;

        strcpy(OldName,FileName);
        GlobalCompact(0);

        if (!GetMemFlag){
                        if (!GetAvgMemory(hWnd)){
                        MessageBox(hWnd, "Unable to Allocate Scan-to-Scan
Memory",
                                                "Error 69", MB_OK |
MB_ICONHAND);
                        }
        }

        /* Allocate bitmap buffer to the size of the image + 1 */

        if (pScanBuffer[0] != NULL){
                        GlobalUnlock(hScanBuffer[0]);
                        pScanBuffer[0] = GlobalFree(hScanBuffer[0]);
        }

        if ((hScanBuffer[0] = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                                        (MemSizeX*MemSizeY))) == NULL){
                        MessageBox(hWnd, "Memory Allocation Error.",
                                        "Error 8.2", MB_OK | MB_ICONHAND);
                        return (NULL);
        }

        hSaveCursor = SetCursor(hHourGlass);

        if ((pScanBuffer[0] = GlobalLock(hScanBuffer[0])) == NULL){
                        MessageBox(hWnd, "Global Lock Failed.",
                                        "Error 9.2", MB_OK | MB_ICONHAND);
                        return (NULL);
        }
        pTemp[0] = pScanBuffer[0];

        hBuffer = LocalAlloc(LMEM_MOVEABLE | LMEM_ZEROINIT,1024);
        if (!hBuffer){
```

```
                    MessageBox(hWnd, "Memory Allocation Error.",
                                "Error 8.3", MB_OK | MB_ICONHAND);
                    return (NULL);
        }
        pBuffer = LocalLock(hBuffer);

        if (!pBuffer){
        MessageBox(hWnd,"Local Lock Failed.","Error 9.3", MB_OK |
MB_ICONHAND);
                    return (NULL);
        }

        pToBuffer = pBuffer;

        close(hFile);

        if (Pixel < 1024 && Line < 768)
                    ScrY = 20;

        hFile = OpenFile(FileName, (LPOFSTRUCT) &OfStruct,  OF_READ);

        if (hFile == EOF){
                        MessageBox(hWnd, "Image File Error",
                                "Error 21", MB_OK | MB_ICONHAND);
                        return NULL;
        }

        fstat(hFile, &FileStatus);
        bytesmoved = _llseek(hFile, 512L, 0);  /* read the header off first */

        nScanCount=0;
        freq = 5.5;
        while (IOStatus < (FileStatus.st_size-bytesmoved)){
                    for (n=0;n<Line;n++) {

                                /* changed Pixel and line */
                                bytesread = read(hFile, pBuffer,Pixel);
                                for (i=0;i<bytesread;i++){

                                            if (SpectraFlag && n < Sline){
                                                    if (i < Spixel){
                    complex_data[index] = 255.0 - ((float
huge)((BYTE)pBuffer[i]))/511.0;
                    complex_data[index+1] = 0.0;
                            index = index + 2;
                                                    }
                                            }

                                 *pScanBuffer[nScanCount] = (BYTE)255 -
pBuffer[i];
                                 pScanBuffer[nScanCount]++;    /* tag it to the tail */
                                }
                                IOStatus += (long)bytesread;
                                pBuffer = pToBuffer;
```

```
                        }
                        nScanCount++;

                        if (pScanBuffer[nScanCount] != NULL){
                                GlobalUnlock(hScanBuffer[nScanCount]);
                        pScanBuffer[nScanCount] =
GlobalFree(hScanBuffer[nScanCount]);
                        }

if ((hScanBuffer[nScanCount] = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                                        (MemSizeX*MemSizeY))) == NULL){
                                MessageBox(hWnd, "Memory Allocation
Error.",
                                        "Error 10", MB_OK |
MB_ICONHAND);
                                return (NULL);
                        }

                        hSaveCursor = SetCursor(hHourGlass);

        if ((pScanBuffer[nScanCount] = GlobalLock(hScanBuffer[nScanCount])) ==
NULL){
                                MessageBox(hWnd, "Global Lock Failed.",
                                        "Error 11", MB_OK |
MB_ICONHAND);
                                return (NULL);
                        }

                        pTemp[nScanCount] = pScanBuffer[nScanCount];
        }

        LocalUnlock(hBuffer);
        LocalFree(hBuffer);

        close(hFile);
        for (i=0;i<=nScanCount;i++)
                        pScanBuffer[i] = pTemp[i];

        if (PalFlag == FALSE) SetUpPallette(hWnd);
        DoSwapVideo();

        ScrX = 0;
        ScrY = 20;

        if (hBitmapImage != NULL)
                        DeleteObject(hBitmapImage);

        if (nScanCount == 1){
                        hDC=GetDC(hWnd);
                        hMemoryDC = CreateCompatibleDC(hDC);
if ((hBitmapImage=CreateDIBitmap(hDC,(LPBITMAPINFOHEADER)&(pBitInfo-
>bmiHeader),
 CBM_INIT, (LPSTR)pScanBuffer[0], pBitInfo,DIB_RGB_COLORS)) == NULL) {
```

```
                        MessageBox(hWnd, "Not enough memory for
bitmap.",
                                        "Error 12", MB_OK |
MB_ICONHAND);
                                ReleaseDC(hWnd,hDC);
                                DeleteDC(hMemoryDC);
                                return NULL;
                        }

                        SelectObject(hMemoryDC, hBitmapImage);
                        BitBlt(hDC,0,20,Pixel,Line,hMemoryDC, 0,0,SRCCOPY);
                        ReleaseDC(hWnd,hDC);
                        DeleteDC(hMemoryDC);
        }
        else
                        for (i=0;i<nScanCount;i++) {
                                WriteBitmap(hWnd, pScanBuffer[i], ScrX,
ScrY, i);
                                ScrX += Pixel+5;
                        }
        BitMap0 = TRUE;


       k = 0;
       r = 0;
        if (nXScanWindows != 0 && nYScanWindows != 0){
                if (nXScanWindows != nYScanWindows){
                        //this is not square so make it so number one!

      . Get_Spec_mem(hWnd,nYScanWindows,nYScanWindows);
                        r = 0;
                        for (i=0;i<sdismemx;i++){
                                for (j=0;j<sdismemy;j++){

            if (k < nXScanWindows*nYScanWindows && j < nXScanWindows){
                                        pSpecImage[r] =
(pImageData[k]*255.0);
                                                k = k + 1;
                                        }else{
                                                pSpecImage[r] = (BYTE) 250;
                                        }
                                        r = r + 1;

                                }
                        }

                }else{

                        //display probability distribution

      Get_Spec_mem(hWnd,nXScanWindows,nYScanWindows);

                for (i=0;i<nXScanWindows*nYScanWindows;i++){
```

```
                                        pSpecImage[i] = (BYTE)
(pImageData[i]*255.0);
                                }


                        }
                xxsiz = sdismemx;
                yysiz = sdismemy;

                DoTheSpectraImageDisplay(hWnd,240,20);


                Free_Spec_mem();
        }



        return 1;
}

/*************************************************************************/
/*                                                                       */
/* Dialog Window Procedure                                               */
/*                                                                       */
/* This procedure is associated with the dialog box that is included in */
/* the function name of the procedure. It provides the service routines */
/* for the events (messages) that occur because the end user operates   */
/* one of the dialog box's buttons, entry fields, or controls.          */
/*                                                                       */
/* The SWITCH statement in the function distributes the dialog box      */
/* messages to the respective service routines, which are set apart by  */
/* the CASE clauses. Like any other Windows window, the Dialog Window   */
/* procedures must provide an appropriate service routine for their end */
/* user initiated messages as well as for general messages (like the    */
/* WM_CLOSE message).                                                    */
/* Dialog messages are processed internally by windows and passed to the*/
/* Dialog Message Procedure. IF processing is done for a Message the    */
/* Message procedure returns a TRUE, else , for messages not explicitly */
/* processed, it returns a FALSE                                         */
/*                                                                       */
/*************************************************************************/

BOOL FAR PASCAL NETLOADMsgProc(HWND hDlg, WORD Message, WORD
wParam, LONG lParam)
{
   char bslash[5] = "\\";

   switch (Message) {
      case WM_COMMAND:
         switch (wParam) {
            case IDC_LISTBOX:
               switch (HIWORD(lParam)) {
                  case LBN_SELCHANGE:
```

```
                    /* If item is a directory name, append "*.*" */
                    strcpy(str, "w*.*");
                    if (!DlgDirSelect(hDlg, str, IDC_LISTBOX))
                    {
                        SetDlgItemText(hDlg, IDC_FILENAME, str);
                        SendDlgItemMessage(hDlg,
                        IDC_FILENAME,
                        EM_SETSEL,
                        NULL,
                        MAKELONG(0, 0x7fff));
                    }
                    else
                    {
                        strcat(str, "w*.*");
                        DlgDirList(hDlg,str,IDC_LISTBOX,
                            IDC_DIRECTORY, 0x4010);
                    }

                    break;

                case LBN_DBLCLK:
                    goto getfilename;
            }
            return (TRUE);

        case IDOK:
getfilename:
            GetDlgItemText(hDlg, IDC_FILENAME, NetFileName, 128);
            if (strchr(NetFileName, '*') || strchr(NetFileName, '?')){
                SeparateFile(hDlg, (LPSTR) str, (LPSTR) DefSpec,
                    (LPSTR) NetFileName);
                if (str[0])
                    strcpy(DefPath, str);
                ChangeDefExt(DefExt, DefSpec);
                UpdateListBox(hDlg);
                return (TRUE);
            }

            if (!NetFileName[0]) {
                MessageBox(hDlg, "No filename specified.",
                    "Error 14", MB_OK | MB_ICONHAND);
                return FALSE;
            }

            GetDlgItemText(hDlg, IDC_DIRECTORY, str, 128);
            if ((GetScanSize(NetFileName)) == 0){

    MessageBox(hDlg, "Not a Net Arch File","Error 15",
                    MB_OK | MB_ICONINFORMATION);
                return FALSE;
            }
            else{
                itoa(ScanSizeX,SSize,10);
                strcat(str,bslash);
```

```
                strcat(str,NetFileName);
                strcpy(NetFileName,str);

        MessageBox(hDlg, NetFileName, "Network Architecture File Name",
                MB_OK | MB_ICONINFORMATION);
            }
            EndDialog(hDlg, TRUE);
            return TRUE;

          case IDCANCEL:
            EndDialog(hDlg, NULL);
            return (FALSE);
        }
        break;

                    case WM_INITDIALOG:         /* message: initialize   */
        UpdateListBox(hDlg);
         SetDlgItemText(hDlg, IDC_FILENAME, DefSpec);
                                SendDlgItemMessage(hDlg,  /* dialog handle
*/
                                 IDC_FILENAME,      /* where to send
message */
                                        EM_SETSEL,        /* select
characters  */
                                        NULL,             /* additional
information */
                                 MAKELONG(0, 0x7fff)); /* entire contents
*/
                                SetFocus(GetDlgItem(hDlg,
IDC_FILENAME)); /*keyboard focus is set to here*/
        DlgDirList(hDlg, str,IDC_LISTBOX, IDC_DIRECTORY, 0x4010);
        DlgDirSelect(hDlg, str, IDC_LISTBOX);
        SetDlgItemText(hDlg, IDC_FILENAME, str);
        return (FALSE); /* Indicates the focus is set to a control */
    }
    return FALSE;
}


int MakeFileName(HWND hWnd)

{
    char *chrtest;
    char *period;
    char strata[3];
    char strdat[3];
    int prd = '.';

    strcpy(FileName,GraphFileName);
    strcpy(strata,"ata");
    strcpy(strdat,"dat");

    period = strchr(FileName,prd);
    period -= 8;
```

```
    chrtest = period;

    if (*chrtest == 'n'){
        period = period + 5;
        strncpy(period,strata,3);
    }
    else if (*chrtest == 'c'){
        period = period + 5;
        strncpy(period,strdat,3);
    }
    else
        return NULL;

return 1;
}


int GetScanSize(char *NetFileName)
{
        int prd = '.';
   char *period;
   char temp[10];

   period = strchr(NetFileName,prd);
   period = period - 2; /* point to letter */
   if (isdigit((int)*period)){
       period = period - 5;
       strncpy(temp,period,2);
       temp[2]=' ';
       NumScans = atoi(temp);
       period = period + 2;
       strncpy(temp,period,2);
       temp[2]=' ';
       ScanSizeX = atoi(temp);
       period = period + 2;
       strncpy(temp,period,2);
       temp[2]=' ';
                        ScanSizeY = atoi(temp);
       period = period + 2;
       strncpy(temp,period,1);
       temp[1]=' ';
       NumOut = atoi(temp);
       period = period + 2;
       strncpy(temp,period,3);
       temp[3]=' ';
       NumLayers = atoi(temp);
   }
   else{
       period = period - 5;
       strncpy(temp,period,2);
       temp[2]=' ';
       NumScans = atoi(temp);
       period = period + 3;
       strncpy(temp,period,2);
```

```
        temp[2]=' ';
        ScanSizeX = atoi(temp);
        ScanSizeY=ScanSizeX;
        period = period + 3;
        strncpy(temp,period,1);
        temp[1]=' ';
        NumOut = atoi(temp);
        period = period + 2;
        strncpy(temp,period,3);
        temp[3]=' ';
        NumLayers = atoi(temp);
    }

    return 1;
}

/*************************************************************************/
/* WriteImageClass() Function                                            */
/*                                                                       */
/* This function writes the classification of the image component to     */
/* a screen box as the image is being scanned.                           */
/*                                                                       */
/*************************************************************************/

int WriteImageClass(void)
{
float Hval,Lval,Sval,pertargetfloat;

if (NumOut == 3)
        {
        Hval = (float)(TarVal);
        Lval = Hval/2;
        Sval = Hval/4;

        if (ythr[1] == 1)
                strcpy(evaluation,"clutter     ");
        else if (ythr[2] == 1)
                strcpy(evaluation,"noise       ");
        else if (ythr[3] == 1)
                strcpy(evaluation,"target      ");
        else
                strcpy(evaluation,"undefined   ");

        perclutter = (int)(yhat[1]*100.0);
        pernoise = (int)(yhat[2]*100.0);
        pertarget = (int)(yhat[3]*100.0);

        if (pertarget >= (int)Hval)
                strcpy(classstr,Htxx);
        else if (pertarget >= (int)Lval && perclutter <30 && pernoise <30)
                strcpy(classstr,Ltxx);
        else if (pertarget >= (int)Sval )
                strcpy(classstr,Stxx);
        else if (perclutter >= 50)
```

```
                    strcpy(classstr,Hcxx);
            else if (perclutter >= 40)
                    strcpy(classstr,Mcxx);
            else if (perclutter >= 10 && pertarget < (int)Sval && pernoise <30)
                    strcpy(classstr,Lcxx);
            else if (pernoise >= 30)
                    strcpy(classstr,Noxx);
            else
                    strcpy(classstr,Noxx);

            itoa(pernoise,amtnoise,10);
            AddDecimal(amtnoise);
            itoa(perclutter,amtclutter,10);
            AddDecimal(amtclutter);
            itoa((int)(yhat[3]*100.0),amttarget,10);

            return (TRUE);
            }
    else
            {
            Hval = (float)(TarVal);
            Lval = Hval/2.0;
            Sval = Hval/4.0;
            if (ythr[1] == 1)
                    strcpy(evaluation,"Target    ");
            else
                    strcpy(evaluation,"No target ");

            pertarget = (int)(yhat[1]*100.0);
            pertargetfloat = (yhat[1]*100.0);
            if (pertargetfloat >= Hval)
                    strcpy(classstr,Htxx);
            else if (pertargetfloat >= Lval)
                    strcpy(classstr,Ltxx);
            else if (pertargetfloat >= Sval)
                    strcpy(classstr,Stxx);
            else
                    strcpy(classstr,Noxx);

                /*Show only target or noise */

            if (!ShowAllFlag)
                    if (pertarget >= Sval)
                            strcpy(classstr,Htxx);

            itoa((int)(yhat[1]*100.0),amttarget,10);

            return (TRUE);
            }
}


int CreateOutFileName(void)
{
```

```
        int prd = '.';
        char *period;

        strcpy(OutName,FileName);
        period = strchr(OutName,prd);
        period = period - 3;
    strncpy(period,"out",3);
    return 1;
}


int pause(int waittime)
{
        long int  i;
        float    x,y=5.0;

        for (i=0;i<=waittime*10000;i++)
                x = exp(y);

        return (1);
}


BOOL FAR PASCAL LOADGRAPHICMsgProc(HWND hDlg, WORD Message,
WORD wParam, LONG lParam)
{
   char bslash[5] = "\\";

   switch (Message) {
      case WM_COMMAND:
         switch (wParam) {
            case IDC_LISTBOX:
               switch (HIWORD(lParam)) {
                  case LBN_SELCHANGE:
                     /* If item is a directory name, append "*.*" */
                     if (!DlgDirSelect(hDlg, str, IDC_LISTBOX))
                     {
                        SetDlgItemText(hDlg, IDC_FILENAME, str);
                        SendDlgItemMessage(hDlg,
                        IDC_FILENAME,
                        EM_SETSEL,
                        NULL,
                        MAKELONG(0, 0x7fff));
                     }
                     else
                     {

                        strcat(str,"?????out.*");
                        DlgDirList(hDlg,str,IDC_LISTBOX,
               IDC_DIRECTORY, 0x4010);
            }

                     break;
```

```
                case LBN_DBLCLK:
                    goto getfilename;
                }
                return (TRUE);

            case IDOK:
getfilename:
                GetDlgItemText(hDlg, IDC_FILENAME, GraphFileName, 128);
                if (strchr(GraphFileName, '*') || strchr(GraphFileName,'?')) {
                    SeparateFile(hDlg, (LPSTR) str, (LPSTR) DefSpec,
                        (LPSTR) GraphFileName);
                    if (str[0])
                        strcpy(DefPath, str);
                    ChangeDefExt(DefExt, DefSpec);
                    UpdateListBox(hDlg);
                    return (TRUE);
                }

                if (!GraphFileName[0]) {
                    MessageBox(hDlg, "No filename specified.",
                        "Error 16", MB_OK | MB_ICONHAND);
                    return (TRUE);
                }

                GetDlgItemText(hDlg, IDC_DIRECTORY, str, 128);
                strcat(str,bslash);
                strcat(str,GraphFileName);
                strcpy(GraphFileName,str);
                MessageBox(hDlg, GraphFileName, "Graphic File Name",
                    MB_OK | MB_ICONINFORMATION);
                EndDialog(hDlg, TRUE);
                return (TRUE);

            case IDCANCEL:
                EndDialog(hDlg, NULL);
                return (FALSE);
        }
        break;

                    case WM_INITDIALOG:         /* message: initialize   */
        UpdateListBox(hDlg);
        SetDlgItemText(hDlg, IDC_FILENAME, DefSpec);
                                SendDlgItemMessage(hDlg,   /* dialog handle
*/
                            IDC_FILENAME,         /* where to send message
*/
                            EM_SETSEL,            /* select characters */
                            NULL,               /* additional information */
                            MAKELONG(0, 0x7fff));   /*entire contents     */
      SetFocus(GetDlgItem(hDlg, IDC_FILENAME)); /*keyboard focus is set  to
here*/
        DlgDirList(hDlg, str,IDC_LISTBOX, IDC_DIRECTORY, 0x4010);
        DlgDirSelect(hDlg, str, IDC_LISTBOX);
        SetDlgItemText(hDlg, IDC_FILENAME, str);
```

```
        return (FALSE); /* Indicates the focus is set to a control */
        }
        return FALSE;
}

/*************************************************************************/
/*                                                                       */
/* This function get the coordinates of the cursor and displays the      */
/* network classificated %'s for the clutter,nnoise, and target.         */
/*                                                                       */
/*************************************************************************/
int GetImageBlock(int X, int Y)
{
        float    xflo, yflo, clu, noi, tar;
        int      xint, yint, sector, imageoffset;
        long            million=1000000;

        xflo = (float)X/GraphicSize;
        xint = X/GraphicSize;
        if (xflo > xint)
                        xint++;
        Y = Y - ylo;
        yflo = (float)Y/GraphicSize;
        yint = Y/GraphicSize;
        if (yflo > yint)
                        yint++;
        yint = nYScanWindows - yint;
        sector = (yint)*nXScanWindows+xint;
        imageoffset = sector*NumOut-NumOut;
        fileoffset=pScanWinCoord[sector-1];
        if (NumOut == 3){
                        clu = 100.0*pImageData[imageoffset];
                        noi = 100.0*pImageData[(imageoffset+1)];
                        tar = 100.0*pImageData[(imageoffset+2)];
                        itoa((int)clu,amtclutter,10);
                        itoa((int)noi,amtnoise,10);
                        itoa((int)tar,amttarget,10);

        }
        else {
                        tar = 100.0*pImageData[(imageoffset)];
                        itoa((int)tar,amttarget,10);
                        AddDecimal(amttarget);
        }

        /* Section to determine corresponding block of radar image  */
        /* changed Imagesize to pixel and line */

        if ( yint+1 == nYScanWindows)
                        Yorg = 20+ScanSizeY;
        else
                        Yorg = Line+20-(yint)*WinInc;
        if ( xint == nXScanWindows)
                        Xorg = Pixel-ScanSizeX;
```

```
        else
                        Xorg = (xint-1)*WinInc;
        ScrY = 20;

        return 1;
}


/* SetScanWin draws a square on the radar image corresponding */
/* to the current cursor position on the graphic              */

void SetScanWin(HWND hWnd) {

                        ScrX=0;
                        hRgn = CreateRectRgn(0, 20, XRgn, YRgn);
                        InvalidateRgn (hWnd, hRgn, TRUE);
                        DeleteObject(hRgn);

      UpdateWindow(hWnd);

      hDC = GetDC(hWnd);
      MoveTo(hDC, Xorg, Yorg);
      LineTo(hDC, Xorg+ScanSizeX, Yorg);
      LineTo(hDC, Xorg+ScanSizeX, Yorg-ScanSizeY);
      LineTo(hDC, Xorg, Yorg-ScanSizeY);
      LineTo(hDC, Xorg, Yorg);

      if (AvgImageFlag){
         Xorg+=120;
         MoveTo(hDC, Xorg, Yorg);
                                LineTo(hDC, Xorg+ScanSizeX, Yorg);
                                LineTo(hDC, Xorg+ScanSizeX, Yorg-
ScanSizeY);
                                LineTo(hDC, Xorg, Yorg-ScanSizeY);
                                LineTo(hDC, Xorg, Yorg);
                        }
                        ReleaseDC(hWnd, hDC);
}


/*************************************************************/
/*  This function gets the graphic info from the             */
/*  network output file.                                     */
/*************************************************************/

int GetGraphic(HWND hWnd){

        int             n,i,j,k;
        char    Ftest[40];
        float   bigest=-1.0;

        if (ImageSet == TRUE){
                        GlobalUnlock(hImageData);
                        pImageData = GlobalFree(hImageData);
```

```
        }

        if ((graph = fopen(GraphFileName,"r")) == NULL){
                        return NULL;
        }

        n = 0;
        i = 1;

        if ((fscanf(graph,"%s",&Ftest)) != 1){
                        MessageBox(hWnd, "Graphic File error.","Error 18.1",
                                        MB_OK | MB_ICONHAND);
                        fclose(graph);
                        return NULL;
        }

        if (Ftest[0] == 'a'){
                        AverageEvalFlag = TRUE;
                        fclose(graph);
                        if ((graph = fopen(GraphFileName,"r")) == NULL)
                                return NULL;
                        if
((fscanf(graph,"%s%s%s%d%d%d%d%d%d%d%d%d",&Ftest,&NetFileName,&FileNa
me,&WinInc,&NumScans,&NumOut,&nXScanWindows,

        &nYScanWindows,&ScanSizeX,&ScanSizeY,&Pixel,&Line)) != 12){
                                MessageBox(hWnd, "Graphic File error.",
                                        "Error 18.2", MB_OK | MB_ICONHAND);
                                fclose(graph);
                                return NULL;
                        }
        }

        else {
                        AverageEvalFlag = FALSE;
                        fclose(graph);
                        if ((graph = fopen(GraphFileName,"r")) == NULL)
                                return NULL;

        if((fscanf(graph,"%s%s%d%d%d%d%d%d%d%d%d",&NetFileName,&FileNa
me,&WinInc,

        &NumScans,&NumOut,&nXScanWindows,&nYScanWindows,&ScanSizeX,
                                        &ScanSizeY,&Pixel,&Line)) != 11){
                                MessageBox(hWnd, "Graphic File error.","Error
18.3", MB_OK | MB_ICONHAND);
                                fclose(graph);
                                return NULL;
                        }
        }

        SelectGraphicBitmap();

        if (!ShowAllFlag){
```

```
                        ScrX = 0;
                        InvalidateRect (hWnd, NULL, TRUE);
                        UpdateWindow(hWnd);
                        hDC = GetDC(hWnd);
                        MoveTo(hDC, 0,Line+40+GraphicSize*nYScanWindows);
                        LineTo(hDC, 0,Line+40);
                        LineTo(hDC, GraphicSize*nXScanWindows-
GraphicSize,Line+40);
                        LineTo(hDC, GraphicSize*nXScanWindows-
GraphicSize,Line+40+GraphicSize*nYScanWindows);
                        LineTo(hDC, 0,Line+40+GraphicSize*nYScanWindows);
                        ReleaseDC(hWnd, hDC);
        }

        memory2 = (long int) (NumOut*nXScanWindows*nYScanWindows*(long
int)sizeof(float));

        if ((hImageData = GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
                                memory2)) == NULL){
                MessageBox(hWnd, "Memory Allocation Error.",
                                "Error 8.4", MB_OK | MB_ICONHAND);
                return (NULL);
        }

        if ((pImageData = (float huge *)GlobalLock(hImageData)) == NULL){
                MessageBox(hWnd, "Global Lock Failed.",
                                "Error 9.4", MB_OK | MB_ICONHAND);
                return (NULL);
        }

        ImageSet = TRUE;

        j = 0;

        SetCursor(hHourGlass);
        UpdateImage = TRUE;
        while ((fscanf(graph,"%f",&value))!= EOF){
                pImageData[j++] = value;
                bigest = max(value,bigest);
        }

        for (k=0;k<j;k++){
                pImageData[k] = pImageData[k]/bigest;
                yhat[i++] = pImageData[k];
                if (i == (NumOut+1)){
                        i = 1;
                        if (!CorrelFlag){
                                WriteImageClass();
                                if (*classstr == 'H' || *classstr == 'T')

DisplayGraphics(hWnd,n++);
                                else
                                        n++;
                        }
```

```
                }
        }

        SetCursor(hArrow);
        UpdateImage = FALSE;
        fclose(graph);

        return 1;
}


/*********************************************************************/
/*  This function gets the second graphic info from the          */
/*  network output file when correlation is requested.           */
/*********************************************************************/

int GetGraphic2(HWND hWnd){

        int             n,i,j,k;
        char    Ftest[40];
        float   bigest=-1.0;

        if (hImageData2){
                GlobalUnlock(hImageData2);
                pImageData2 = GlobalFree(hImageData2);
        }

        if ((graph = fopen(GraphFileName,"r")) == NULL){
                return NULL;
        }

        n = 0;
        i = 1;

        if ((fscanf(graph,"%s",&Ftest)) != 1){
                MessageBox(hWnd, "Graphic File error.","Error 18.1",
                                MB_OK | MB_ICONHAND);
                fclose(graph);
                return NULL;
        }

        if (Ftest[0] == 'a'){
                AverageEvalFlag = TRUE;
                fclose(graph);
                if ((graph = fopen(GraphFileName,"r")) == NULL)
                        return NULL;
                if
((fscanf(graph,"%s%s%s%d%d%d%d%d%d%d%d%d",&Ftest,&NetFileName,&FileNa
me,&WinInc,&NumScans,&NumOut,&nXScanWindows,

        &nYScanWindows,&ScanSizeX,&ScanSizeY,&Pixel,&Line)) != 12){
                        MessageBox(hWnd, "Graphic File error.",
                                "Error 18.2", MB_OK | MB_ICONHAND);
                        fclose(graph);
```

```
                        return NULL;
                    }
        }

        else {
                    AverageEvalFlag = FALSE;
                    fclose(graph);
                    if ((graph = fopen(GraphFileName,"r")) == NULL)
                        return NULL;

    if((fscanf(graph,"%s%s%d%d%d%d%d%d%d%d%d",&NetFileName,&FileNa
me,&WinInc,

    &NumScans,&NumOut,&nXScanWindows,&nYScanWindows,&ScanSizeX,
                        &ScanSizeY,&Pixel,&Line)) != 11){
                    MessageBox(hWnd, "Graphic File error.","Error
18.3", MB_OK | MB_ICONHAND);
                    fclose(graph);
                    return NULL;
                }
        }

        SelectGraphicBitmap();

        if (!ShowAllFlag){
                    ScrX = 0;
                    InvalidateRect (hWnd, NULL, TRUE);
                    UpdateWindow(hWnd);
                    hDC = GetDC(hWnd);
                    MoveTo(hDC, 0,Line+40+GraphicSize*nYScanWindows);
                    LineTo(hDC, 0,Line+40);
                    LineTo(hDC, GraphicSize*nXScanWindows-
GraphicSize,Line+40);
                    LineTo(hDC, GraphicSize*nXScanWindows-
GraphicSize,Line+40+GraphicSize*nYScanWindows);
                    LineTo(hDC, 0,Line+40+GraphicSize*nYScanWindows);
                    ReleaseDC(hWnd, hDC);
        }

        memory2 = (long int)
(NumOut*nXScanWindows*nYScanWindows*sizeof(float));

        if ((hImageData2 = GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
                        memory2)) == NULL){
                    MessageBox(hWnd, "Memory Allocation Error.",
                        "Error 8.5", MB_OK | MB_ICONHAND);
                    return (NULL);
        }

        if ((pImageData2 = (float huge *)GlobalLock(hImageData2)) == NULL){
                    MessageBox(hWnd, "Global Lock Failed.",
                        "Error 9.4", MB_OK | MB_ICONHAND);
                    return (NULL);
        }
```

```
        ImageSet = TRUE;

        j = 0;

        SetCursor(hHourGlass);
        UpdateImage = TRUE;
        while ((fscanf(graph,"%f",&value))!= EOF){
                        pImageData2[j++] = value;
                        bigest = max(value,bigest);
        }

        for (k=0;k<j;k++){
                        pImageData2[k] = pImageData2[k]/bigest;
                        yhat[i++] = pImageData2[k];
                        if (i == (NumOut+1)){
                                        i = 1;
                                        if (!CorrelFlag){
                                                        WriteImageClass();
                                                        if (*classstr == 'H' || *classstr == 'T')

DisplayGraphics(hWnd,n++);
                                                        else
                                                                        n++;
                                        }
                        }
        }
        SetCursor(hArrow);
        UpdateImage = FALSE;
        fclose(graph);

        return 1;
}



/*************************************************************************/
/*                                                                       */
/* nCwRegisterClasses Function                                           */
/*                                                                       */
/* The following function registers all the classes of all the windows   */
/* associated with this application. The function returns an error code  */
/* if unsuccessful, otherwise it returns 0.                              */
/*                                                                       */
/*************************************************************************/

int nCwRegisterClasses(void)
{
 WNDCLASS  wndclass;  /* struct to define a window class          */
 memset(&wndclass, 0x00, sizeof(WNDCLASS));


 /* load WNDCLASS with window's characteristics                    */
 wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_BYTEALIGNWINDOW;
```

```
 wndclass.lpfnWndProc = WndProc;
 /* Extra storage for Class and Window objects                    */
 wndclass.cbClsExtra = 0;
 wndclass.cbWndExtra = 0;
 wndclass.hInstance = hInst;
 wndclass.hIcon = LoadIcon(hInst, "RADARIMG");
 wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
 /* Create brush for erasing background                           */
 wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
 wndclass.lpszMenuName = szAppName;  /* Menu Name is App Name */
 wndclass.lpszClassName = szAppName; /* Class Name is App Name */
 if(!RegisterClass(&wndclass))
  return -1;

 return(0);
} /* End of nCwRegisterClasses                                    */


/*********************************************************************/
/* CwUnRegisterClasses Function                                      */
/*                                                                   */
/* Deletes any refrences to windows resources created for this       */
/* application, frees memory, deletes instance, handles and does     */
/* clean up prior to exiting the window                              */
/*                                                                   */
/*********************************************************************/

void CwUnRegisterClasses(void)
{
        WNDCLASS  wndclass;   /* struct to define a window class     */
        memset(&wndclass, 0x00, sizeof(WNDCLASS));

        UnregisterClass(szAppName, hInst);
}  /* End of CwUnRegisterClasses                                   */


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

   FUNCTION: SaveAsDlg(HWND, unsigned, WORD, LONG)

   PURPOSE: Allows user to change name to save file to

   COMMENTS:

      This will initialize the window class if it is the first time this
      application is run.  It then creates the window, and processes the
      message loop until a PostQuitMessage is received.  It exits the
      application by returning the value passed by the PostQuitMessage.

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

int FAR PASCAL SaveAsDlg(HWND hDlg,unsigned message,WORD
wParam,LONG lParam)
```

```
{
    char TempName[128];

    switch (message) {
        case WM_INITDIALOG:

            /* If no filename is entered, don't allow the user to save to it */

            if (!FileName[0])
                bSaveEnabled = FALSE;
            else {
                bSaveEnabled = TRUE;

                /* Process the path to fit within the IDC_PATH field */

                DlgDirList(hDlg, DefPath, NULL, IDC_PATH, 0x4010);

                /* Send the current filename to the edit control */

                SetDlgItemText(hDlg, IDC_EDIT, DefSpec);

                /* Accept all characters in the edit control */

                SendDlgItemMessage(hDlg, IDC_EDIT, EM_SETSEL, 0,
                    MAKELONG(0, 0x7fff));
            }

            /* Enable or disable the save control depending on whether the
             * filename exists.
             */

            EnableWindow(GetDlgItem(hDlg, IDOK), bSaveEnabled);

            /* Set the focus to the edit control within the dialog box */

            SetFocus(GetDlgItem(hDlg, IDC_EDIT));
            return (FALSE);             /* FALSE since Focus was changed */

        case WM_COMMAND:
            switch (wParam) {
                case IDC_EDIT:

                    /* If there was previously no filename in the edit
                     * control, then the save control must be enabled as soon as
                     * a character is entered.
                     */

                    if (HIWORD(lParam) == EN_CHANGE && !bSaveEnabled)
                    EnableWindow(GetDlgItem(hDlg, IDOK), bSaveEnabled = TRUE);
                    return (TRUE);

                case IDOK:

                    /* Get the filename from the edit control */
```

```
                GetDlgItemText(hDlg, IDC_EDIT, TempName, 128);

                /* If there are no wildcards, then separate the name into
                 * path and name. If a path was specified, replace the
                 * default path with the new path.
                 */

                 if (CheckFileName(hDlg, (PSTR) FileName, (PSTR) TempName)) {
                    SeparateFile(hDlg, (LPSTR) str, (LPSTR) DefSpec,
                       (LPSTR) FileName);
                    if (str[0])
                       strcpy(DefPath, str);

                    /* Tell the caller a filename was selected */

                    EndDialog(hDlg, IDOK);
                 }
                 return (TRUE);

             case IDCANCEL:

                 /* Tell the caller the user canceled the SaveAs function */

                 EndDialog(hDlg, IDCANCEL);
                 return (TRUE);
          }
          break;

    }
    return (FALSE);
}

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

    FUNCTION: CheckFileName(HWND, PSTR, PSTR)

    PURPOSE: Check for wildcards, add extension if needed

    COMMENTS:

       Make sure you have a filename and that it does not contain any
       wildcards. If needed, add the default extension. This function is
       called whenever your application wants to save a file.

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

BOOL CheckFileName(HWND hWnd,PSTR pDest,PSTR pSrc)
{
    PSTR pTmp;
    if (!pSrc[0])
       return (FALSE);       /* Indicates no filename was specified */
    pTmp = pSrc;
    while (*pTmp) {          /* Searches the string for wildcards */
       switch (*pTmp++) {
```

```
            case '*':
            case '?':
               MessageBox(hWnd, "Wildcards not allowed.",
                   "Error 2", MB_OK | MB_ICONEXCLAMATION);
               return (FALSE);
         }
   }
   AddExt(pSrc, DefExt);          /* Adds the default extension if needed */
   if (OpenFile(pSrc, (LPOFSTRUCT) &OfStruct, OF_EXIST) >= 0) {
      strcpy(str, "Replace existing ");
      strcat(str,pSrc);
      strcat(str,"?");
      if (MessageBox(hWnd, str, "RID",
            MB_OKCANCEL | MB_ICONHAND) == IDCANCEL)
         return (FALSE);
   }
   strcpy(pDest, pSrc);
   return (TRUE);
}

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

   FUNCTION: SaveFile(HWND)

   PURPOSE: Save current file

   COMMENTS:

      This saves the current contents of the Edit buffer, and changes
      bChanges to indicate that the buffer has not been changed since the
      last save.

      Before the edit buffer is sent, you must get its handle and lock it
      to get its address.  Once the file is written, you must unlock the
      buffer.  This allows Windows to move the buffer when not in immediate
      use.

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

BOOL SaveFile(HWND hWnd)
{
        BOOL          bSuccess;
        long          ArrPosition;
        int           count;
        HANDLE   hBuff;
        FILE          *fp;

        BYTE huge *pToBuff;
        BYTE huge *pBuff;
        BYTE huge *pArr;


   pArr = pTemp[0];
   bNew =FALSE;
```

```
   fp = fopen(FileName,"w");

        if (fp && (ScrY != 20))   /* make sure another smaller bitmap was not
opened */
   {
      hSaveCursor = SetCursor(hHourGlass);
                        WriteFileHeader(fp);

        /* screen only has space for 713 lines, title and menu take up the rest */
      ArrPosition = (long)(Line-713-1)*Pixel;
      ArrPosition += (long)(713-OrgY)*Pixel + (OrgX-ScanSizeX);
      WriteToFile(pArr, ArrPosition, fp);

      fclose(fp);
      SetCursor(hSaveCursor);
      bSuccess = TRUE;            /* Indicates the file was saved     */
      bChanges = FALSE;           /* Indicates changes have been saved */
      return (bSuccess);

   }
   else if(fp && (ScrY == 20))
   {
      hSaveCursor = SetCursor(hHourGlass);
      WriteFileHeader(fp);

      /* DI Bitmap origin in lower left corner of screen*/

      ArrPosition = (long)(Line+ScrY-OrgY)*Pixel + (OrgX-ScanSizeX);
      WriteToFile(pArr, ArrPosition, fp);

      if ((hBuff = GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
            FileStatus.st_size-512)) == NULL){
         MessageBox(hWnd, "Memory Allocation Error.",
               "Error 8.6", MB_OK | MB_ICONHAND);
         return FALSE;
      }

      if ((pBuff = (BYTE huge *)GlobalLock(hBuff)) == NULL){
         MessageBox(hWnd, "Global Lock Failed.",
               "Error 9.5", MB_OK | MB_ICONHAND);
         return FALSE;
      }

      pToBuff = pBuff;

                        for (count=1;count<NumScans;count++) {
         GetNextFile(count);
        hFile = OpenFile(NextFile, (LPOFSTRUCT) &OfStruct, OF_READ);
                              _llseek(hFile, 512L, 0);  /* read the header off first */
         _lread(hFile, (LPSTR)pBuff, FileStatus.st_size-512);
         WriteToFile(pBuff, ArrPosition, fp);
         pBuff = pToBuff;
         close(hFile);
      }
```

```
                    GlobalUnlock(hBuff);
        GlobalFree(hBuff);
        fclose(fp);                 /* output file */
        SetCursor(hSaveCursor);
        bSuccess = TRUE;            /* Indicates the file was saved     */
        bChanges = FALSE;           /* Indicates changes have been saved */
        return (bSuccess);

    }

    else
    {
        sprintf(str, "Attempt to save file failed!");
        MessageBox(hWnd, str, NULL, MB_OK | MB_ICONEXCLAMATION);
        return (FALSE);
    }
}

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

    FUNCTION: OpenDlg(HWND, unsigned, WORD, LONG)

    PURPOSE: Let user select a file, and return. Open code not provided.

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

HANDLE FAR PASCAL OpenDlg(HWND hDlg,unsigned message,WORD
wParam,LONG lParam)
{
    HANDLE hFile;

    switch (message) {
        case WM_COMMAND:
            switch (wParam) {

                case IDC_LISTBOX:
                    switch (HIWORD(lParam)) {

                        case LBN_SELCHANGE:
                            /* If item is a directory name, append "*.*" */
                            if (!DlgDirSelect(hDlg, str, IDC_LISTBOX))
                            { SetDlgItemText(hDlg, IDC_EDIT, str);
                              SendDlgItemMessage(hDlg,
                              IDC_EDIT,
                              EM_SETSEL,
                              NULL,
                              MAKELONG(0, 0x7fff));
                            }
                            else
                            {
                              strcat(str, DefSpec);
                              DlgDirList(hDlg,str,IDC_LISTBOX,
```

```
                        IDC_PATH, 0x4010);
                }

                break;

            case LBN_DBLCLK:
                goto openfile;
        }
        return (TRUE);

    case IDOK:
openfile:
        GetDlgItemText(hDlg, IDC_EDIT, OpenName, 128);
        if (strchr(OpenName, '*') || strchr(OpenName, '?')) {
            SeparateFile(hDlg, (LPSTR) str, (LPSTR) DefSpec,
                (LPSTR) OpenName);
            if (str[0])
                strcpy(DefPath, str);
            ChangeDefExt(DefExt, DefSpec);
            UpdateListBox(hDlg);
            return (TRUE);
        }

        if (!OpenName[0]) {
            MessageBox(hDlg, "No filename specified.",
                "Error 3", MB_OK | MB_ICONHAND);
            return (TRUE);
        }

        AddExt(OpenName, DefExt);


        /* The routine to open the file would go here, and the */
        /* file handle would be returned instead of NULL.   */

        if ((int)(hFile = OpenFile((LPSTR) OpenName,
(LPOFSTRUCT)&OfStruct,
                OF_READ)) == EOF) {
            sprintf(str, "Error %d opening %s.",
                OfStruct.nErrCode, OpenName);
            MessageBox(hDlg, str, NULL,
                MB_OK | MB_ICONHAND);
        }
        else {

            fstat(hFile, &FileStatus);
        sprintf(str, "File size is %ld bytes.", FileStatus.st_size);

            MessageBox(hDlg, str, OpenName,
                MB_OK | MB_ICONINFORMATION);

            strcpy(FileName, OpenName);
            EndDialog(hDlg, hFile);
            return (TRUE);
```

```
            }
        case IDCANCEL:
            EndDialog(hDlg, NULL);
            return (FALSE);
        }
        break;

    case WM_INITDIALOG:                 /* message: initialize   */
        UpdateListBox(hDlg);
        SetDlgItemText(hDlg, IDC_EDIT, DefSpec);

                        SendDlgItemMessage(hDlg,        /* dialog handle
*/
                                IDC_EDIT,           /* where to send
message */
                                EM_SETSEL,          /* select characters
*/
                                NULL,               /* additional information
*/
                                MAKELONG(0, 0x7fff));   /* entire
contents    */
        SetFocus(GetDlgItem(hDlg, IDC_EDIT));
        return (FALSE); /* Indicates the focus is set to a control */
    }
    return FALSE;
}


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

    FUNCTION: UpdateListBox(HWND);

    PURPOSE: Update the list box of OpenDlg

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

void UpdateListBox(HWND hDlg)
{
    DlgDirList(hDlg, str, IDC_LISTBOX, IDC_PATH, 0x4010);

    /* To ensure that the listing is made for a subdir. of
     * current drive dir...
     */
    if (!strchr (DefPath, ':'))
    DlgDirList(hDlg, DefSpec, IDC_LISTBOX, IDC_PATH, 0x4010);

    /* Remove the '..' character from path if it exists, since this
     * will make DlgDirList move us up an additional level in the tree
     * when UpdateListBox() is called again.
     */
    if (strstr (DefPath, ".."))
                        DefPath[0] = '\0';

   SetDlgItemText(hDlg, IDC_EDIT, DefSpec);
```

```
}

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

   FUNCTION: ChangeDefExt(PSTR, PSTR);

   PURPOSE: Change the default extension

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

void ChangeDefExt(PSTR Ext,PSTR Name)
{
   PSTR  pTptr;

   pTptr = Name;
   while (*pTptr && *pTptr != '.')
      pTptr++;
   if (*pTptr)
      if (!strchr(pTptr, '*') && !strchr(pTptr, '?'))
         strcpy(Ext, pTptr);
}

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

   FUNCTION: SeparateFile(HWND, LPSTR, LPSTR, LPSTR)

   PURPOSE: Separate filename and pathname

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

void SeparateFile(HWND hDlg, LPSTR lpDestPath,LPSTR lpDestFileName,LPSTR
lpSrcFileName)
{
   LPSTR lpTmp;
   char  cTmp;

   lpTmp = lpSrcFileName + (long) lstrlen(lpSrcFileName);
   while (*lpTmp != ':' && *lpTmp != '\\' && lpTmp > lpSrcFileName)
      lpTmp = AnsiPrev(lpSrcFileName, lpTmp);
   if (*lpTmp != ':' && *lpTmp != '\\') {
      lstrcpy(lpDestFileName, lpSrcFileName);
      lpDestPath[0] = 0;
      return;
   }
   lstrcpy(lpDestFileName, lpTmp + 1);
   cTmp = *(lpTmp + 1);
   lstrcpy(lpDestPath, lpSrcFileName);
   *(lpTmp + 1) = cTmp;
   lpDestPath[(lpTmp - lpSrcFileName) + 1] = 0;
}

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

   FUNCTION: AddExt(PSTR, PSTR);
```

```
   PURPOSE: Add default extension

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

void AddExt(PSTR Name, PSTR Ext)
{
   PSTR pTptr;

   pTptr = Name;
   while (*pTptr && *pTptr != '.')
      pTptr++;
   if (*pTptr != '.')
      strcat(Name, Ext);
}


/*********************************************************************************/
/*                                                                               */
/* Dialog Window Procedure                                                       */
/*                                                                               */
/* This procedure is associated with the dialog box that is included in */
/* the function name of the procedure. It provides the service routines */
/* for the events (messages) that occur because the end user operates   */
/* one of the dialog box's buttons, entry fields, or controls.          */
/*                                                                               */
/*********************************************************************************/

BOOL FAR PASCAL CaptureDlg(HWND hWndDlg, WORD Message, WORD
wParam, LONG lParam)
{

int MyFlag;
switch(Message)
   {
   case WM_INITDIALOG:
         cwCenter(hWndDlg, 0);
         /* initialize working variables                                */

         SetDlgItemInt(hWndDlg, scx, ScanSizeX, 9);
         SetDlgItemInt(hWndDlg, scy, ScanSizeY, 9);
         SetDlgItemInt(hWndDlg, pi, ScanScans, 5);

         return FALSE;
         /* End of WM_INITDIALOG                                        */

   case WM_CLOSE:
         /* Closing the Dialog behaves the same as Cancel               */
         PostMessage(hWndDlg, WM_COMMAND, IDCANCEL, 0L);
         break; /* End of WM_CLOSE                                      */

   case WM_COMMAND:
         switch(wParam)
           {
```

```
            case scx: /* Edit Control                                    */
                ScanSizeX = (int)GetDlgItemInt(hWndDlg, scx, &MyFlag, 0);

                break;

            case scy: /* Edit Control                                    */
                ScanSizeY = (int)GetDlgItemInt(hWndDlg, scy, &MyFlag, 0);

                break;

            case pi: /* Edit Control                                     */
                ScanScans = (int)GetDlgItemInt(hWndDlg, pi, &MyFlag, 0);

                break;


            case IDOK:
                EndDialog(hWndDlg, TRUE);

                break;

            case IDCANCEL:
          /* Ignore data values entered into the controls       */
              /* and dismiss the dialog window returning FALSE      */
              EndDialog(hWndDlg, FALSE);
              break;
            }
                          break;   /* End of WM_COMMAND                          */

            default:
                          return FALSE;
            }
 return TRUE;
} /* End of CaptureDlg                                  */

/*************************************************************************/
/*                                                                       */
/* Dialog Window Procedure                                               */
/*                                                                       */
/* This procedure is associated with the dialog box that is included in */
/* the function name of the procedure. It provides the service routines */
/* for the events (messages) that occur because the end user operates   */
/* one of the dialog box's buttons, entry fields, or controls.          */
/*                                                                       */
/*************************************************************************/

BOOL FAR PASCAL MarkDlg(HWND hWndDlg, WORD Message, WORD wParam,
LONG lParam)
{

int StartPixel, CurrPixel, CheckCoord;
int i, j, k, n;
```

```
switch(Message)
        {
        case WM_INITDIALOG:

                        return FALSE;
                        /* End of WM_INITDIALOG                        */

        case WM_CLOSE:
                        /* Closing the Dialog behaves the same as Cancel      */
                        PostMessage(hWndDlg, WM_COMMAND, IDEND, 0L);
                       break; /* End of WM_CLOSE                        */

        case WM_COMMAND:
                        switch(wParam)
                                        {
        case IDOK:
                                       StartPixel=(Line+ScrY-LowLeftY)*Pixel +
LowLeftX;
                                                       CurrPixel=StartPixel;
                                                       ImageTarIndex = 0;

        TargetPos[ImageTar][ImageTarIndex++] = CurrPixel;
                                                       n=0;
                                                       for(i=LowLeftY;i>=MarkY;i--){

for(j=LowLeftX;j<=MarkX;j++){

        CheckCoord=CurrPixel-CheckPos;

        for(k=0;k<nScanWindows;k++)

if(CheckCoord == pScanWinCoord[k]){

                TargetWin[TargetNo][n++]=k;

                break;
                        }
                       CurrPixel++;

        TargetPos[ImageTar][ImageTarIndex++] = CurrPixel;
                                                                       }
                        CurrPixel+=Pixel-MarkX+LowLeftX;
                                                       }
                                                       ImageTar++;
                                                       TargetNo++;

                                                       EndDialog(hWndDlg, TRUE);

                                       break;

                                       case IDCANCEL:

                                                       EndDialog(hWndDlg, TRUE);
```

```
                                        break;

                                        case IDTARSAVE:

                                        if ( (Targetdat = fopen("target.dat","w"))!=
NULL){
                                                for (i=0;i<ImageTar;i++){
                                                        fprintf(Targetdat,"-1
");
                                                        for
(j=0;j<MarkPixelNumber;j++)
                                                                if (TargetPos[i][j] == -
1)
                                                                        break;
                                                                else
                                                        fprintf(Targetdat,"%d
",TargetPos[i][j]);
                                                                }
                                                fclose(Targetdat);
                                                }
                                                else
                                MessageBox(hWndDlg, "File error","error opening file",
                                        MB_OK | MB_ICONINFORMATION);


                                break;


                                        case IDEND:

                                                TargetNo=0;
                                                EndDialog(hWndDlg, FALSE);

                                        break;
                                        }
                                break;   /* End of WM_COMMAND                    */

        default:
                                return FALSE;
        }
 return TRUE;
} /* End of MarkDlg                                */



/*************************************************************************/
/*                                                */
/* Dialog Window Procedure                                */
/*                                                */
/* This procedure is associated with the dialog box that is included in */
/* the function name of the procedure. It provides the service routines */
/* for the events (messages) that occur because the end user operates   */
/* one of the dialog box's buttons, entry fields, or controls.          */
```

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

BOOL FAR PASCAL OverlapDlg(HWND hWndDlg, WORD Message, WORD
wParam, LONG lParam)
{

 int MyFlag;
 switch(Message)
        {
        case WM_INITDIALOG:

                        cwCenter(hWndDlg, 0);
                        /* initialize working variables                    */
                                SetDlgItemInt(hWndDlg, wi, WinInc, 1);
                                if (Batch){
                              SendDlgItemMessage(hWndDlg,   /* dialog handle
*/
                                IDC_EDIT,               /* where to send
message  */
                                EM_SETSEL,          /* select characters
*/
                                NULL,               /* additional
information */
                                                MAKELONG(0, 0x7fff));
                                        SetFocus(GetDlgItem(hWndDlg,
wi));
                                }
                                return FALSE;
                                /* End of WM_INITDIALOG                    */

                case WM_CLOSE:
                                /* Closing the Dialog behaves the same as
Cancel          */
                              PostMessage(hWndDlg, WM_COMMAND,
IDCANCEL, 0L);
                                break; /* End of WM_CLOSE
*/

                case WM_COMMAND:
                                switch(wParam)
                                 {
                                 case  wi: /* Edit Control
*/
                              WinInc = (int)GetDlgItemInt(hWndDlg, wi,
&MyFlag, 0);
                                                                break;

                                        case IDOK:
                                                        EndDialog(hWndDlg,
TRUE);
                                                                break;
                                        case IDCANCEL:
```

```
                                        /* Ignore data values entered into the
controls*/
                                        /* and dismiss the dialog window returning
FALSE */
                                                        EndDialog(hWndDlg,
FALSE);
                                                                break;
                                        }
                                break;  /* End of WM_COMMAND
*/

                default:
                                return FALSE;
        }
 return TRUE;
} /* End of OverlapDlg                      */


void cwCenter(HWND hWnd, int top)
{
 POINT    pt;
 RECT     swp;
 RECT     rParent;
 int      iwidth;
 int      iheight;

 /* get the rectangles for the parent and the child              */
 GetWindowRect(hWnd, &swp);
 GetClientRect(hWndMain, &rParent);

 /* calculate the height and width for MoveWindow                */
 iwidth = swp.right - swp.left;
 iheight = swp.bottom - swp.top;

 /* find the center point and convert to screen coordinates      */
 pt.x = (rParent.right - rParent.left) / 2;
 pt.y = (rParent.bottom - rParent.top) / 2;
 ClientToScreen(hWndMain, &pt);

 /* calculate the new x, y starting point                        */
 pt.x = pt.x - (iwidth / 2);
 pt.y = pt.y - (iheight / 2);

 /* top will adjust the window position, up or down              */
 if(top)
        pt.y = pt.y + top;

 /* move the window                                              */
 MoveWindow(hWnd, pt.x, pt.y, iwidth, iheight, FALSE);
}
```

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

        FUNCTION: About(HWND, unsigned, WORD, LONG)

        PURPOSE:  Processes messages for "About" dialog box

        MESSAGES:

                        WM_INITDIALOG - initialize dialog box
                        WM_COMMAND    - Input received

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

BOOL FAR PASCAL About(hDlg, message,wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
LONG lParam;
{
        switch (message) {
                        case WM_INITDIALOG:
                                        return (TRUE);

                        case WM_COMMAND:
                        if (wParam == IDOK
                                                || wParam == IDCANCEL) {
                                                EndDialog(hDlg, TRUE);
                                                return (TRUE);
                                        }
                                        break;
        }
        return (FALSE);
}



/****************************************************************************/
/* GetWinCoord Function                                                     */
/*                                                                          */
/* Purpose:                                                                 */
/*

*/
/*    1) To calculate the number of scan windows needed to process          */
/*      an entire image                                                     */
/*    2) To determine the starting position for each scan window            */
/*      and store them in an array pScanWinCoord[]                          */
/*                                                                          */
/* Main Variables:                                                          */
/*                                                                          */
/*    nqScanWindows - local, contains # scan winows per scan line           */
/*    nrScanWindows - global, contains # bytes remaining at end of line     */
/*    nScanWindows - global, total number of scan windows                   */
```

```
/*    TempWindow - running count of number of scan windows          */
/*    Row      - identifies current row number                      */
/*    CurrRowPos - identifies position of the pointer in the current row */
/*    RowBegin - offset in bytes of the current row                 */
/*    pScanWinCoord - array to hold various starting points         */
/*                                                                  */
/* Description:                                                     */
/*                                                                  */
/*    This function obtains the quotient and remainder for the division */
/* ImageSize by ScanSize to determine the number of scan windows. If the */
/* result has no remainder then the number of scan windows per row is */
/* nqScanWindows else the number of scan windows per line is incremented. */
/*                                                                  */
/*    When storing the positions in the array both the beginning and end */
/* of the row are checked to see if a scan window will fit exactly. If */
/* it will not, the position is decremented such that it will fit exactly.*/
/*                                                                  */
/*                                                                  */
/* Return:                                                          */
/*                                                                  */
/*    This function will return the total number of scan windows or -1 */
/* on error.                                                        */
/*                                                                  */
/********************************************************************************/

int GetWinCoord(HWND hWnd)
{
        int TempWindow =0, Row = 0;
        int CurrRowPos=0, RowBegin=0;
        int i=0;

        /* changed Imagesize to pixel and line */
        for (i=0;Pixel-WinInc*i>ScanSizeX;i++)
                ;
        nXScanWindows = i+1;

        for (i=0;Line-WinInc*i>ScanSizeY;i++)
                ;
        nYScanWindows = i+1;

        nScanWindows = nXScanWindows*nYScanWindows;

            if(GetScanMemFlag){
                pScanWinCoord = pToScanWinCoord;
                for(i=0; i<nScanWindows; i++)
                    pScanWinCoord[i]=0;
            }
            else
                GetScanMemory(hWnd);


        /* Calculate the starting positions of each scan window */

        while (TempWindow < nScanWindows) {
```

```
                    if (Line-Row < ScanSizeY)
                                Row = Line-ScanSizeY;
                    RowBegin = Pixel*Row;
        for (CurrRowPos=0;Pixel-CurrRowPos>ScanSizeX;CurrRowPos+=WinInc){

        pScanWinCoord[TempWindow++]=RowBegin+CurrRowPos;
                    }
                    CurrRowPos = Pixel-ScanSizeX;
                    pScanWinCoord[TempWindow++]=RowBegin+CurrRowPos;

                    Row+=WinInc;
        }
        return nScanWindows;

}   /* End of GetWinCoord                          */

/*************************************************************************/
/* StoreScanWin  Function                                    */
/*                                                           */
/* Purpose:                                                  */
/*                                                           */
/*    1) To store the data from the scan window indicated by the function*/
/*       argument into an array for the specified number of scans.      */
/*                                                           */
/*                                                           */
/* Main Variables:                                           */
/*                                                           */
/*    TempOffset - temp var to store the original offset             */
/*    ScanWinData - global, array to hold the scan window data for      */
/*              specified number of scans                    */
/*    NumScans - global, specifies the number of scans               */
/*                                                           */
/*                                                           */
/* Description:                                              */
/*                                                           */
/*    This function will store in an array the data for a given scan    */
/* window. It will also, for the number of scans specified, open the    */
/* appropriate radar data files and append the scan window for each file */
/* to the array. This code resembles the code for the SaveFile function. */
/*                                                           */
/*                                                           */
/* Return:                                                   */
/*                                                           */
/*    This function will currently has no return value. It should,     */
/* however, be modified to return 0 if successful or -1 if the function  */
/* failed.                                                   */
/*                                                           */
/*************************************************************************/

int StoreScanWin(int Offset,HWND hWnd)
{
        int     i=0, j, numlines, x=0, nScanRow=0, TempOffset,bytesread;
        int     Limit, Length, TempScans;
        long int  VectorSize;
```

```
        BYTE huge *pTempWin;
        int     posfound, ExtNum,count;
        char    NextFile[128];
        char    FileTemp[128];
        char    FileExt[4];
        char    FileExt2[4];
        BYTE huge *lpSWB;
        char    *pchar;

        pTempWin = pTemp[0];
        TempOffset = Offset;
        while (nScanRow < ScanSizeY) {
                pTemp[0] +=   Offset;
                for (i=0;i<ScanSizeX;i++) {
                        pScanWinData[x++] = *pTemp[0];
                        pTemp[0]++;
                }
                Offset = Pixel - ScanSizeX;
                nScanRow++;
        }
        pTemp[0] = pTempWin;

        /* Code to append successive scans to the array. */

        VectorSize = FileStatus.st_size-512;

        if ((hScanWinBuff = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                        VectorSize)) == NULL){
                MessageBox(hWnd, "Memory Allocation Error.",
                        "Error 8.8", MB_OK | MB_ICONHAND);
                return NULL;
        }

        if ((pScanWinBuff = (BYTE huge *)GlobalLock(hScanWinBuff)) == NULL){
                MessageBox(hWnd, "Global Lock Failed.",
                        "Error 9.7", MB_OK | MB_ICONHAND);
                return NULL;
        }

        lpSWB = pScanWinBuff;
        i=0;
        j=0;

        pchar = strstr(OldName,".");
        posfound = pchar - OldName;
        for (i=0;i<=posfound;i++)
                NextFile[i] = OldName[i];
        NextFile[i] = '\0';
        strcpy(FileTemp,NextFile);

        while (OldName[i] != '\0')
                FileExt[j++] = OldName[i++];
        ExtNum = atoi(FileExt);
```

```
        if(EvalAvg)
                        TempScans = ScanScans + 1;
        else
                        TempScans = ScanScans;

        for (count=1;count<TempScans;count++) {
                        ExtNum++;
                        if (ExtNum < 10)
                                        strcat(NextFile,"00");
                        else if(ExtNum <100 && ExtNum >=10)
                                        strcat(NextFile,"0");

                        itoa(ExtNum,FileExt2,10);
                        strcat(NextFile,FileExt2);
                        hFile = OpenFile(NextFile, (LPOFSTRUCT) &OfStruct,

        OF_READ);
                        if (!hFile){
        MessageBox(hWnd, "File Open Error!","Error 33", MB_OK |
MB_ICONHAND);
                                        return NULL;
                        }

                        if(!(bytesread =_llseek(hFile, 512L, 0))){ /* read the header
off first */
                                        MessageBox(hWnd, "File Open Error!",
                                                "Error 33", MB_OK |
MB_ICONHAND);
                                        return NULL;
                        }
                if(!(bytesread=_lread(hFile,
(LPSTR)pScanWinBuff,FileStatus.st_size-512))){
                                        MessageBox(hWnd, "File Open Error!",
                                                "Error 33", MB_OK |
MB_ICONHAND);
                                        return NULL;
                        }
                        pScanWinBuff = lpSWB;

                        Offset = TempOffset;

                        for (numlines=0;numlines<ScanSizeY;numlines++)
                        {
                                        pScanWinBuff += Offset;
                                        for (i=0;i<ScanSizeX;i++){
                                                pScanWinData[x++] =
*pScanWinBuff;
                                                pScanWinBuff++;
                                        }
                                        Offset = Pixel - ScanSizeX;
                        }

                        strcpy(NextFile,FileTemp);
```

```
                        pScanWinBuff = lpSWB;
                        _lclose(hFile);

        }
        pScanWinBuff = lpSWB;
        GlobalUnlock(hScanWinBuff);
        GlobalFree(hScanWinBuff);

        if (EvalAvg){
                        Limit = ScanSizeX*ScanSizeY*(TempScans-1);
                        Length = ScanSizeX*ScanSizeY;
                        x = 0;
                        for(i=0;i<Limit;i++){
pScanWinData[x] = (BYTE)((float)(pScanWinData[x]+pScanWinData[x+Length-
1])/2);
                                x++;
                        }
        }

        return 1;
}   /* End of StoreScanWin                          */


void DisplayGraphics(HWND hWnd, int window)
{
        nScrX = ((window+1)%nXScanWindows)*GraphicSize-GraphicSize;
        nScrY = (40+Line+nYScanWindows*GraphicSize)-
                (int)((window+1)/nXScanWindows)*GraphicSize-GraphicSize;

        if (ShowAllFlag || (*classstr == 'H' || *classstr == 'T')){
                        hDC=GetDC(hWnd);
                        hMemoryDC = CreateCompatibleDC(hDC);
                        hBitmapGraphic = LoadBitmap(hInst, classstr);
                        if (hBitmapGraphic == NULL) {
                                        DeleteDC(hMemoryDC);
                                        ReleaseDC(hWnd,hDC);
                                        return;
                        }
                        SelectObject(hMemoryDC, hBitmapGraphic);
        BitBlt(hDC,nScrX,nScrY,GraphicSize,GraphicSize,hMemoryDC,
0,0,SRCCOPY);
                        DeleteDC(hMemoryDC);
                        ReleaseDC(hWnd,hDC);
                        DeleteObject(hBitmapGraphic);
        }
}


void MoveScanWin(HWND hWnd, int window)
{
        int check;

        DoSwapVideo();
```

```
        check = (window+1)%(nXScanWindows);

        if (PrevY-WinInc >= 20 && EndRow) {
                        OrgY -= WinInc;
                        PrevY -= WinInc;
                        EndRow = FALSE;
        }

        if (PrevY-WinInc < 20 && EndRow) {
                        PrevY = 20;
                        OrgY = PrevY+ScanSizeY;
                        EndRow = FALSE;
        }


        if (check || !window){
                        ScrX=0;
                        if (!Batch){
                                        hRgn = CreateRectRgn(0, 20, XRgn, YRgn);
                                        InvalidateRgn (hWnd, hRgn, TRUE);
                                        DeleteObject(hRgn);
                        }
                        else
                                        InvalidateRect (hWnd, NULL, TRUE);

                        UpdateWindow(hWnd);

                        hDC = GetDC(hWnd);
                        MoveTo(hDC, OrgX, OrgY);
                        LineTo(hDC, OrgX, PrevY);
                        LineTo(hDC, PrevX, PrevY);
                        LineTo(hDC, PrevX, OrgY);
                        LineTo(hDC, OrgX, OrgY);
                        ReleaseDC(hWnd, hDC);


                        PrevX+=WinInc;
                        OrgX+=WinInc;
        }

if (!check && window) {
                        OrgX = Pixel;
                        PrevX = OrgX - ScanSizeX;

                if(window != nScanWindows-1)
                                        EndRow = TRUE;

                ScrX=0;
                        if (!Batch){
                                        hRgn = CreateRectRgn(0, 20, XRgn, YRgn);
                                        InvalidateRgn (hWnd, hRgn, TRUE);
                                        DeleteObject(hRgn);
                        }
                        else
```

```
                                InvalidateRect (hWnd, NULL, TRUE);

                        UpdateWindow(hWnd);

                        hDC = GetDC(hWnd);
                        MoveTo(hDC, OrgX, OrgY);
                        LineTo(hDC, OrgX, PrevY);
                        LineTo(hDC, PrevX, PrevY);
                        LineTo(hDC, PrevX, OrgY);
                        LineTo(hDC, OrgX, OrgY);
                        ReleaseDC(hWnd, hDC);

                PrevX=0;
                OrgX=ScanSizeX;
  }
}


void WriteFileHeader(FILE *fp){

        int numchar,j,i;

                fprintf(fp,"%5u",ScanSizeX*ScanSizeY*ScanScans);
                numchar = 5;

                for (i=6;i<=(64*4);i++)
                        {  j = fprintf(fp,"x");
                                numchar += j;
                        }

                        j = fprintf(fp,"%4d pixels",ScanSizeX);
                        numchar += j;
                        while (numchar < (5*64))
                        {  j=fprintf(fp,"x");
                                numchar += j;
                        }

                        j=fprintf(fp,"%4d lines",ScanSizeY);
                        numchar += j;
                        while (numchar < (6*64))
                        {  j=fprintf(fp,"x");
                                numchar += j;
                        }

                        while (numchar < (64*8))
                        {  j = fprintf(fp,"x");
                                numchar += j;
                        }  /* header 512 bytes */
}

void WriteSpecFileHeader(void){

        int numchar,j,i;
```

```
                fprintf(sp,"%5u",Spixel*Sline);
                numchar = 5;

                for (i=6;i<=(64*4);i++)
                        {   j = fprintf(sp,"x");
                                        numchar += j;
                        }

                        j = fprintf(sp,"%4d pixels",Spixel);
                        numchar += j;
                        while (numchar < (5*64))
                        {   j=fprintf(sp,"x");
                                        numchar += j;
                        }

                        j=fprintf(sp,"%4d lines",Sline);
                        numchar += j;
                        while (numchar < (6*64))
                        {   j=fprintf(sp,"x");
                                        numchar += j;
                        }

                        /* header 512 bytes */
}


int GetNextFile(int count)
{
        int  i,j=0, posfound;
        char FileTemp[128];
        char FileExt[4];
        char FileExt2[4];
        char *pchar;

        pchar = strstr(OldName,".");
        posfound = pchar - OldName;
        for (i=0;i<=posfound;i++)
                        NextFile[i] = OldName[i];
        NextFile[i] = '\0';
        strcpy(FileTemp,NextFile);
        while (OldName[i] != '\0')
                        FileExt[j++] = OldName[i++];
        ExtNum = atoi(FileExt);
        ExtNum+=count;
        if (ExtNum < 10)
                        strcat(NextFile,"00");
        else if(ExtNum <100 && ExtNum >=10)
                        strcat(NextFile,"0");

        itoa(ExtNum,FileExt2,10);
        strcat(NextFile,FileExt2);
        return 1;

}
```

```
void WriteToFile(BYTE huge *pScanData, long position, FILE *fp)
{
        int i, k, numlines;

        for (k=0;k<position;k++)
                                pScanData++;
        for (numlines=0; numlines < ScanSizeY; numlines++){
                if (numlines != 0)
                {   for (i=0;i<(Pixel-ScanSizeX);i++)
                                pScanData++;
                }
                for (i=0;i<ScanSizeX;i++)
                {
                        fprintf(fp,"%c",*pScanData);
                        pScanData++;
                }
        }
}

int WriteToBuffer(int fileoffset, int count, HWND hWnd)
{
        int i, numlines,x;
        int hFile,bytesread;
        long bytes =0;
        BYTE huge *pToRead;

        if ((hReadBuffer = GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
                        FileStatus.st_size-512)) == NULL){
                MessageBox(hWnd, "Memory Allocation Error.",
                        "Error 8.9", MB_OK | MB_ICONHAND);
                return NULL;
        }

        if ((pReadBuffer = (BYTE huge *)GlobalLock(hReadBuffer)) == NULL){
                MessageBox(hWnd, "Global Lock Failed.",
                        "Error 9.8", MB_OK | MB_ICONHAND);
                return NULL;
        }

        pToRead = pReadBuffer;
        if ((hFile = OpenFile(NextFile, (LPOFSTRUCT) &OfStruct,
                OF_READ)) == -1){
       MessageBox(hWnd, "Error opening file.", "FAILED!", MB_OK |
       B_ICONINFORMATION);
                }
        bytes = _llseek(hFile, 512L, 0);     /* read the header off first */
        if (!bytes){
       MessageBox(hWnd, "seek error.","Error", MB_OK |
MB_ICONINFORMATION);
                return NULL;
                }
        bytesread = _lread(hFile, (LPSTR)pReadBuffer, FileStatus.st_size-512);
```

```
        if (!bytesread){
                MessageBox(hWnd, NextFile,"Error reading NextFile", MB_OK |
                        MB_ICONINFORMATION);
                        return NULL;
                        }

        pReadBuffer = pToRead;
        pReadBuffer+=fileoffset;
        x=0;
        /* changed Memsize to Line*/

        for (numlines=0; numlines < MemSizeY; numlines++){
                        if (numlines != 0)
                                        pReadBuffer+=Pixel-ScanSizeX;

                        if (numlines<ScanSizeY)
                                        for (i=0;i<MemSizeX;i++)
                                          if(i>=ScanSizeX){        /* Fill in white space
*/

       *pDisplayBuffer[count]=(BYTE)250;
                                                        pDisplayBuffer[count]++;

pAverageBuffer[x++]+=(BYTE)250;
                                                        }
                                                        else{
                                                        *pDisplayBuffer[count] =
*pReadBuffer;
                                                        pAverageBuffer[x++] +=
*pReadBuffer;

pDisplayBuffer[count]++;
                                                                        pReadBuffer++;
                                                        }
                                else
                                        for (i=0;i<MemSizeX;i++){       /* Fill in white
space */
                                                *pDisplayBuffer[count]=(BYTE)250;
                                                pDisplayBuffer[count]++;
                                                pAverageBuffer[x++]+=(BYTE)250;
                                        }

        }
        pReadBuffer = pToRead;
        close(hFile);
        GlobalUnlock(hReadBuffer);
        GlobalFree(hReadBuffer);
        return 1;
}


int WriteBitmap(HWND hWnd, BYTE huge *data, int X, int Y, int count)
{
        hDC=GetDC(hWnd);
```

```
        if (hDC == NULL){
                MessageBox(hWnd, "GetDC failed",
                        "Error 13", MB_OK | MB_ICONHAND);
                return NULL;
        }
        hMemoryDC = CreateCompatibleDC(hDC);
        if (hMemoryDC == NULL){
                MessageBox(hWnd, "CreateCompatibleDC failed",
                        "Error 13", MB_OK | MB_ICONHAND);
                return NULL;
        }
        if ((hBitmap[count]=CreateDIBitmap(hDC,
(LPBITMAPINFOHEADER)&(pBitInfo->bmiHeader),
                        CBM_INIT, (LPSTR)data,
(LPBITMAPINFO)pBitInfo,DIB_RGB_COLORS))== NULL) {
                        MessageBox(hWnd, "Not enough memory for
bitmap.",
                                "Error 13", MB_OK |
MB_ICONHAND);
                        ReleaseDC(hWnd,hDC);
                        DeleteDC(hMemoryDC);
                        return NULL;
                }
        hOldBitmap[count] = SelectObject(hMemoryDC, hBitmap[count]);
        BitBlt(hDC,X,Y,MemSizeX,MemSizeY,hMemoryDC, 0,0,SRCCOPY);
        SelectObject(hMemoryDC, hOldBitmap[count]);
        DeleteObject(hBitmap[count]);
        ReleaseDC(hWnd,hDC);
        DeleteDC(hMemoryDC);
        return 1;
}

void DisplayText(HWND hWnd)
{
        if(AvgImageFlag)
                TextPosX=2*(Pixel+20);
        else
                TextPosX=Pixel+20;

        TextPosY=ScanSizeY+30;
        hDC = GetDC(hWnd);
        if (NumOut == 3){
                TextOut(hDC,
TextPosX,TextPosY,evaluation,strlen(evaluation));
                TextOut(hDC, TextPosX,TextPosY+20,strcat(amtclutter, " % Clutter
")
                                ,strlen(amtclutter));
                TextOut(hDC, TextPosX,TextPosY+40,strcat(amtnoise, " % Noise ")
                                ,strlen(amtnoise));
                TextOut(hDC, TextPosX,TextPosY+60,strcat(amttarget, " % Target
")
                                ,strlen(amttarget));
        }
        else
```

```
                    TextOut(hDC, TextPosX,TextPosY,strcat(amttarget, " %
Target ")
                                                    ,strlen(amttarget));
        ReleaseDC(hWnd,hDC);
}

int Evaluate(HWND hWnd)
{
        int             i,j,test;

        if(!ImageFlag)
                    return NULL;

        SetCursor(hHourGlass);
        if (First)
                    GetWinCoord(hWnd);

        SelectGraphicBitmap();
        CreateOutFileName();
        fin2 = fopen(OutName,"w");
        if (!fin2){
        MessageBox(hWnd, "Error opening file.","Error X", MB_OK |
MB_ICONHAND);
                    return NULL;
        }
        if (EvalAvg)
                    fprintf(fin2,"average\n");

        fprintf(fin2,"%s\n",NetFileName);
        fprintf(fin2,"%s\n",FileName);
        fprintf(fin2,"%d\n",WinInc);
        fprintf(fin2,"%d\n",NumScans);
        fprintf(fin2,"%d\n",NumOut);
        fprintf(fin2,"%d\n",nXScanWindows);
        fprintf(fin2,"%d\n",nYScanWindows);
        fprintf(fin2,"%d\n",ScanSizeX);
        fprintf(fin2,"%d\n",ScanSizeY);
        fprintf(fin2,"%d\n",Pixel);
        fprintf(fin2,"%d\n",Line);

        ScrX=0;
        PrevX =0;
        OrgX = ScanSizeX;
        PrevY = Line+20-ScanSizeY;
        OrgY = Line+20;
        nScrX=0;

        XRgn = Pixel+1;
        YRgn = Pixel+21;

        VideoFlag = ShowAllFlag;

        for (i=0;i<nScanWindows;i++){
```

```
                        if (VideoFlag) MoveScanWin(hWnd,i);

                                if(Batch){
                                        TextOut(hDC, 130, 20, string, strlen(string));
                                        TextOut(hDC,
137,45,FileName,strlen(FileName));
                                }

                        if((test = StoreScanWin(pScanWinCoord[i],hWnd)) ==
NULL)
                        MessageBox(hWnd, "Null",NULL, MB_OK |
MB_ICONHAND);

                        blocktst();

                        if (fin2){
                                        for (j=1;j<=NumOut;j++)
                                                fprintf(fin2,"%f\n",yhat[j]);
                                                fprintf(fin2,"\n");
                        }


                        if (!Batch){
                                        WriteImageClass();
                                        DisplayGraphics(hWnd,i);
                                        DisplayText(hWnd);
                        }
        }

        if (fclose(fin2) != 0)
        MessageBox(hWnd, "Error closing file.","Error 6", MB_OK |
MB_ICONHAND);
        else if(!Batch){
                        strcpy(GraphFileName,OutName);
                        if (!GetGraphic(hWnd))
                                        return NULL;
        }
        return 1;
}

/* Code to free all allocated memory on WM_EXIT or WM_CLOSE */

void FreeMemory(HWND hWnd){

        int i;


        ResetPallette();
        FreeAvgMemory();
        GlobalUnlock(hPixelRecord);
        pPixelRecord = GlobalFree(hPixelRecord);
        GlobalUnlock(hPixelRecord2);
        pPixelRecord2 = GlobalFree(hPixelRecord2);
        GlobalUnlock(hImageAvgByte);
```

```
        pImageAvgByte = GlobalFree(hImageAvgByte);
        GlobalUnlock(hlpMonsterArr1);
        plpMonsterArr1 = GlobalFree(hlpMonsterArr1);
        GlobalUnlock(hScanWinCoord);
        pScanWinCoord = GlobalFree(hScanWinCoord);
        GlobalUnlock(hImageData);
        pImageData = GlobalFree(hImageData);
        GlobalUnlock(hImageData2);
        pImageData2 = GlobalFree(hImageData2);
        GlobalUnlock(hScanWinData);
        pScanWinData = GlobalFree(hScanWinData);
        DeleteObject(hBitmapImage);
        DeleteObject(hOldBitmapImage);
        unlock_ram();
        GlobalUnlock(hBitInfo);
        pBitInfo = GlobalFree(hBitInfo);
        for (i=0;i<nScanCount;i++){
                DeleteObject(hBitmap[i]);
                DeleteObject(hOldBitmap[i]);
                GlobalUnlock(hScanBuffer[i]);
                pScanBuffer[i] = GlobalFree(hScanBuffer[i]);
        }

        if (!ClearAll) {
                DestroyWindow(hWnd);
                if (hWnd == hWndMain)
                        PostQuitMessage(0);
        }
}

/* Determined the Graphic size to use for the cartoon */

void SelectGraphicBitmap(void){

        nScrX=0;
        GraphicSize=12;
        nScrY=Line+40+GraphicSize*(nYScanWindows-1);
        if (nScrY > 440){
                GraphicSize=8;
                nScrY=Line+40+GraphicSize*(nYScanWindows-1);
        }
        if (nScrY > 440){
                GraphicSize=6;
                nScrY=Line+40+GraphicSize*(nYScanWindows-1);
        }
        if (nScrY > 440){
                GraphicSize=4;
                nScrY=Line+40+GraphicSize*(nYScanWindows-1);
        }
        if (nScrY > 440){
                GraphicSize=3;
                nScrY=Line+40+GraphicSize*(nYScanWindows-1);
        }
```

```
if (nScrY > 440){
        GraphicSize=1;
        nScrY=Line+40+GraphicSize*(nYScanWindows-1);
}

switch(GraphicSize){
        case 12:
                strcpy(Noxx,"No12");
                strcpy(Lcxx,"Lc12");
                strcpy(Mcxx,"Mc12");
                strcpy(Hcxx,"Hc12");
                strcpy(Stxx,"St12");
                strcpy(Ltxx,"Lt12");
                strcpy(Htxx,"Ht12");
                break;

        case 8:
                strcpy(Noxx,"No08");
                strcpy(Lcxx,"Lc08");
                strcpy(Mcxx,"Mc08");
                strcpy(Hcxx,"Hc08");
                strcpy(Stxx,"St08");
                strcpy(Ltxx,"Lt08");
                strcpy(Htxx,"Ht08");
                break;

        case 6:
                strcpy(Noxx,"No06");
                strcpy(Lcxx,"Lc06");
                strcpy(Mcxx,"Mc06");
                strcpy(Hcxx,"Hc06");
                strcpy(Stxx,"St06");
                strcpy(Ltxx,"Lt06");
                strcpy(Htxx,"Ht06");
                break;

        case 4:
                strcpy(Noxx,"No04");
                strcpy(Lcxx,"No04");
                strcpy(Mcxx,"No04");
                strcpy(Hcxx,"No04");
                strcpy(Stxx,"Tr04");
                strcpy(Ltxx,"Tr04");
                strcpy(Htxx,"Tr04");
                break;

        case 3:
                strcpy(Noxx,"No03");
                strcpy(Lcxx,"No03");
                strcpy(Mcxx,"No03");
                strcpy(Hcxx,"No03");
                strcpy(Stxx,"Tr03");
                strcpy(Ltxx,"Tr03");
```

```
                                        strcpy(Htxx,"Tr03");
                                        break;

                        case 1:
                                        strcpy(Noxx,"No01");
                                        strcpy(Lcxx,"No01");
                                        strcpy(Mcxx,"No01");
                                        strcpy(Hcxx,"No01");
                                        strcpy(Stxx,"Tr01");
                                        strcpy(Ltxx,"Tr01");
                                        strcpy(Htxx,"Tr01");
                                        break;

        }
} /* End of SelectgraphicBitmap  */


void network(void) {

        int     i,j;
        long int  ref=0L;
        double    val,net;


        /*insert inputs*/
        for (i=1;i<=neuron;i++)
                pScanWinNorm[i] = (float huge) (((float huge)
pScanWinCopy[i])/NORM);

        for (i=neuron+1;i<=neuronstotal;i++)
        {
         net = 0.0;
         for (j=1;j<=neuron;j++)
         {
                        val = (double)plpMonsterArr1[ref];
                        net = net + val*(double)pScanWinNorm[j];
                        ref = ref + 1L;
         }
         pScanWinNorm[i] = 1.0/(1.0+exp(-net*LAMDA));
        }

        for (i=neuronstotal+1;i<=neuronstotal+NumOut;i++)
        {
         net = 0.0;
         for (j=neuron+1;j<=neuronstotal;j++)
         {
                        val = (double)plpMonsterArr1[ref];
                        net = net + val*(double)pScanWinNorm[j];
                        ref = ref + 1L;
         }
         pScanWinNorm[i] = 1.0/(1.0+exp(-net*LAMDA));

        }
```

```
        for (i=1;i<=NumOut;i++)
                yhat[i] = pScanWinNorm[i+neuronstotal];

} /* End of network function */


int neurons(HWND hWnd) {

        int         i,j;
        float       ww_val;
        FILE        *fin1;
        long int    dex=0L;

        fin1 = fopen(NetFileName,"r");

        if (!fin1){
                MessageBox(hWnd, "No fin1.",
                        "Error 6", MB_OK | MB_ICONHAND);
                return NULL;
        }

        for (i=neuron+1;i<=neuronstotal;i++){
                for (j=1;j<=neuron;j++){
                        fscanf(fin1,"%f ",&ww_val);
                        plpMonsterArr1[dex] = ww_val;
                        dex = dex + 1L;
                }
        }

        for (i=neuronstotal+1;i<=neuronstotal+NumOut;i++){
                for (j=neuron+1;j<=neuronstotal;j++){
                        fscanf(fin1,"%f ",&ww_val);
                        plpMonsterArr1[dex] = ww_val;
                        dex = dex + 1L;
                }
            }

            fclose(fin1);
            return 1;
} /* END */


int nettst(HWND hWnd)
{
 neuron = (long int)ScanSizeX*ScanSizeY*NumScans;
 neuronstotal =(long int) (neuron+NumLayers);
 memory1 = (long int)
((NumLayers+NumOut+1L)*(neuron+NumLayers+NumOut+1L));


        if (plpMonsterArr1 != NULL){
                GlobalUnlock(hlpMonsterArr1);
                plpMonsterArr1 = GlobalFree(hlpMonsterArr1);
```

```
        }

        if ((hlpMonsterArr1 = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                                memory1*sizeof(double))) == NULL){
                        MessageBox(hWnd, "Memory Allocation Error.",
                                "Error 8.11", MB_OK | MB_ICONHAND);
                        return NULL;
        }

        if ((plpMonsterArr1 = (double huge *)GlobalLock(hlpMonsterArr1)) ==
NULL){
                        MessageBox(hWnd, "Global Lock Failed.",
                                "Error 9.9", MB_OK | MB_ICONHAND);
                        return NULL;
        }

        testdon = donniearrays(hWnd);


        if (plpMonsterArr1)
                        neurons(hWnd);

        return 1;
}

void blocktst(void)
{
        int     i;

        for(i=1;i<=neuron;i++)
                        pScanWinCopy[i] = (unsigned int huge) pScanWinData[i];

        network();

        for (i=1;i<=NumOut;i++)
                        if(yhat[i] > 0.5)
                                ythr[i] = 1;
                        else
                                ythr[i] = 0;
}

void unlock_ram(void){

        GlobalUnlock(hScanWinNorm);
        pScanWinNorm = GlobalFree(hScanWinNorm);
        GlobalUnlock(hScanWinCopy);
        pScanWinCopy = GlobalFree(hScanWinCopy);


}

int donniearrays(HWND hWnd) {
```

```
  memory1 = (long int)
((NumLayers+NumOut+1L)*(neuron+NumLayers+NumOut+1L));
  memory3 = (long int) (neuronstotal+NumLayers+NumOut+1L);

        if (pScanWinNorm != NULL){
                GlobalUnlock(hScanWinNorm);
                pScanWinNorm = GlobalFree(hScanWinNorm);
        }

        if ((hScanWinNorm = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                        memory1*sizeof(double))) == NULL){
                MessageBox(hWnd, "Memory Allocation Error.",
                        "Error 8.12", MB_OK | MB_ICONHAND);
                return NULL;
        }

        if ((pScanWinNorm = (double huge *)GlobalLock(hScanWinNorm)) ==
NULL){
                MessageBox(hWnd, "Global Lock Failed.",
                        "Error 9.11", MB_OK | MB_ICONHAND);
                return NULL;
        }

        if (pScanWinCopy != NULL){
                GlobalUnlock(hScanWinCopy);
                pScanWinCopy = GlobalFree(hScanWinCopy);
        }

        if ((hScanWinCopy = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                        (neuronstotal+1L)*sizeof(unsigned int))) ==
NULL){
                MessageBox(hWnd, "Memory Allocation Error.",
                        "Error 8.13", MB_OK | MB_ICONHAND);
                return NULL;
        }

        if ((pScanWinCopy = (unsigned int huge *)GlobalLock(hScanWinCopy))
                        == NULL){
                MessageBox(hWnd, "Global Lock Failed.",
                        "Error 9.12", MB_OK | MB_ICONHAND);
                return NULL;
        }

        return 1;
}



int DoTheImageAverageEvaluationForTheSmartRadarProject(HWND hWnd)
{
        char    TempAvgFile[80];
        long int  i,j,n,bytesread,bytesmoved,x;
```

```
        BYTE    AvgVal;
        float           avemax;


        pReadAvgBuffer = pToReadAvgBuffer;
        pImageAvgBuffer = pToImageAvgBuffer;
        pImageAvgByte = pToImageAvgByte;

        for (x=0;x<(MemSizeX*MemSizeY);x++) pImageAvgBuffer[x] = 0.0;

        x = 0L;
        strcpy(TempAvgFile,FileName);
        for (i=0;i<ScanScans;i++){
                        hFile = OpenFile(FileName, (LPOFSTRUCT)
&OfStruct,OF_READ);

                        if (hFile == -1){
                                        MessageBox(hWnd, "Image avg file Error",
                                                "Error 21", MB_OK |
MB_ICONHAND);
                                        return NULL;
                        }

                        bytesmoved = _llseek(hFile, 512L, 0);
                        if (bytesmoved == -1){
                                        MessageBox(hWnd, "_llseek error",
                                                "Error 98", MB_OK |
MB_ICONHAND);
                                        return NULL;
                        }

                        x = 0L;
                        for (n=0;n<Line;n++){
                                bytesread =
_lread(hFile,(LPSTR)pReadAvgBuffer,Pixel);
                                        if (bytesread == 0){
                                         MessageBox(hWnd, "_lread error",
                                         "Error tr", MB_OK | MB_ICONHAND);
                                                return NULL;
                                        }
                                        pReadAvgBuffer = pToReadAvgBuffer;
                                        for (j=0;j<bytesread;j++){
                         pImageAvgBuffer[(long)x] += (float
huge)pReadAvgBuffer[j];
                                                x++;
                                        }
                        }
                        _lclose(hFile);
                        strcpy(OldName,FileName);
                        GetNextFile(1);
                        strcpy(FileName,NextFile);
                        pReadAvgBuffer = pToReadAvgBuffer;
        }
```

```
        avemax = -1000.0;
        for (i=0;i<Pixel*Line;i++){
                pImageAvgBuffer[i] = pImageAvgBuffer[i]/(float
huge)ScanScans;
        }

        for (i=0;i<Pixel*Line;i++){
                avemax = max(avemax,pImageAvgBuffer[i]);
        }

        for (i=0;i<Pixel*Line;i++){
                pImageAvgBuffer[i] = (pImageAvgBuffer[i]/avemax)*255.0;
        }

        for (i=0;i<Pixel*Line;i++){
                if(Threshold){
                        ThresholdAvgFlag = TRUE;
                        AvgVal = (BYTE huge)pImageAvgBuffer[i];
                        if(AvgVal>ThresholdVal)
                                pImageAvgByte[i]=220;
                        else
                                pImageAvgByte[i]=45;
                }
                else {
                        ThresholdAvgFlag = FALSE;
                    pImageAvgByte[i] = (BYTE
huge)pImageAvgBuffer[i];
                }
        }

        if (hImageAvgByte != NULL)
                DeleteObject(hImageAvgByte);

        DoSwapVideo();

        hDC=GetDC(hWnd);
        hMemoryDC = CreateCompatibleDC(hDC);

        for (i=0;i<Pixel*Line;i++){
                pImageAvgByte[i] = (BYTE)255 - pImageAvgByte[i];
        }

       if
((hImageAvgByte=CreateDIBitmap(hDC,(LPBITMAPINFOHEADER)&(pBitInfo-
>bmiHeader), CBM_INIT, (LPSTR)pImageAvgByte, pBitInfo,DIB_RGB_COLORS))
== NULL) {
                MessageBox(hWnd, "Not enough memory for bitmap.",
                        "Error 14", MB_OK | MB_ICONHAND);
                ReleaseDC(hWnd,hDC);
                DeleteDC(hMemoryDC);
                return NULL;
        }

        SelectObject(hMemoryDC, hImageAvgByte);
```

```
        BitBlt(hDC,120,20,Pixel,Line,hMemoryDC, 0,0,SRCCOPY);
        ReleaseDC(hWnd,hDC);
        DeleteDC(hMemoryDC);

        strcpy(FileName,TempAvgFile);
        strcpy(OldName,FileName);


        return (1);
}


int DoTheSpectraImageDisplay(HWND hWnd, int xpos, int ypos)
{
        long int  i,j,n,x;

        if (hSpecImage != NULL) DeleteObject(hSpecImage);

        DoSwapVideo();

        hDC=GetDC(hWnd);
        hMemoryDC = CreateCompatibleDC(hDC);

        if
((hSpecImage=CreateDIBitmap(hDC,(LPBITMAPINFOHEADER)&(pBitInfo-
>bmiHeader), CBM_INIT, (LPSTR)pSpecImage, pBitInfo,DIB_RGB_COLORS)) ==
NULL) {
                MessageBox(hWnd, "Not enough memory for Spectra
bitmap.",
                        "Error 14-1", MB_OK | MB_ICONHAND);
                ReleaseDC(hWnd,hDC);
                DeleteDC(hMemoryDC);
                return NULL;
        }

        SelectObject(hMemoryDC, hSpecImage);

        BitBlt(hDC,xpos,ypos,xxsiz,yysiz,hMemoryDC, 0,0,SRCCOPY);

        if (SpectraFlag){
                StretchhDC = GetDC(hWnd);
StretchBlt(StretchhDC,xpos,ypos+yysiz+20,175,175,hDC,xpos,ypos,xxsiz,yysiz,SRCC
OPY);
                ReleaseDC(hWnd,StretchhDC);
        }

        if (!SpectraFlag) TextOut(hDC,235,5,"Neural Distribution",19);


        ReleaseDC(hWnd,hDC);
        DeleteDC(hMemoryDC);

            pBitInfo->bmiHeader.biWidth = Pixel;
            pBitInfo->bmiHeader.biHeight = Line;
```

```
                pBitInfo->bmiHeader.biSizeImage = (Pixel*Line);

        return (1);
}


int PunchThru(HWND hWnd){

        int i,MaxScans;

        MemSizeX = ScanSizeX;
        MemSizeY = ScanSizeY;
        while(MemSizeX%4)
                        MemSizeX++;
        while(MemSizeY%4)
                        MemSizeY++;

        pBitInfo->bmiHeader.biWidth = MemSizeX;
        pBitInfo->bmiHeader.biHeight = MemSizeY;
        pBitInfo->bmiHeader.biSizeImage = MemSizeX*MemSizeY;


        if ((hAverageBuffer = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                                        (MemSizeX*MemSizeY*sizeof(float)))) ==
NULL){
                        MessageBox(hWnd, "Memory Allocation Error.",
                                        "Error 8.17", MB_OK | MB_ICONHAND);
                        return (NULL);
        }

        if ((pAverageBuffer = (float huge *)GlobalLock(hAverageBuffer)) ==
NULL){
                        MessageBox(hWnd, "Global Lock Failed.",
                                        "Error 9.15", MB_OK | MB_ICONHAND);
                        return (NULL);
        }

        pToAverage = pAverageBuffer;

        if ((hAverageByte = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,
                                                (MemSizeX*MemSizeY))) == NULL){
                        MessageBox(hWnd, "Memory Allocation Error.",
                                        "Error 8.18", MB_OK | MB_ICONHAND);
                        return (NULL);
        }

        if ((pAverageByte = (BYTE huge *)GlobalLock(hAverageByte)) == NULL){
                        MessageBox(hWnd, "Global Lock Failed.",
                                        "Error 9.16", MB_OK | MB_ICONHAND);
                        return (NULL);
        }
```

```
        pToAverageByte = pAverageByte;

        MaxScans = min(5,ScanScans);
        for(i=1;i<=ScanScans;i++){
                hDisplayBuffer[i] = GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT,

(MemSizeX*MemSizeY));
                        if (hDisplayBuffer[i] == NULL){
                                MessageBox(hWnd, "Memory Allocation
Error.",
                                "Error 6", MB_OK | MB_ICONHAND);
                                return NULL;
                        }
                        pDisplayBuffer[i] = GlobalLock(hDisplayBuffer[i]);
                        if (pDisplayBuffer[i] == NULL){
                                MessageBox(hWnd, "Memory Allocation
Error.",
                                        "Error 7", MB_OK |
MB_ICONHAND);
                                LocalFree(hDisplayBuffer[i]);
                                return NULL;
                        }
                        pToDisplay[i] = pDisplayBuffer[i];
                        strcpy(OldName,FileName);
                        GetNextFile(i-1);
                        WriteToBuffer(fileoffset, i, hWnd);
                        pDisplayBuffer[i] = pToDisplay[i];
                        SetCursor(hSaveCursor);
                        if(i<=MaxScans){
                                WriteBitmap(hWnd, pDisplayBuffer[i], ScrX,
ScrY, i);
                                ScrX += ScanSizeX+5;
                        }
                }

        pAverageBuffer = pToAverage;

        for (i=0;i<MemSizeX*MemSizeY;i++)
                        pAverageByte[i]=(pAverageBuffer[i]/ScanScans);

        pAverageByte = pToAverageByte;

        ScrX += ScanSizeX+10;
        WriteBitmap(hWnd, pAverageByte, ScrX, ScrY, MaxScans+1);

        pBitInfo->bmiHeader.biWidth = Pixel;
        pBitInfo->bmiHeader.biHeight = Line;
        pBitInfo->bmiHeader.biSizeImage = (Pixel*Line);

        for (i=1;i<=MaxScans+1;i++){
                        GlobalUnlock(hDisplayBuffer[i]);
                        GlobalFree(hDisplayBuffer[i]);
        }
```

```
        pAverageByte = pToAverageByte;
        pAverageBuffer = pToAverage;
        GlobalUnlock(hAverageByte);
        GlobalFree(hAverageByte);
        GlobalUnlock(hAverageBuffer);
        GlobalFree(hAverageBuffer);

        return 1;
}


/*************************************************************/
/*                                                           */
/* function ScanGraphic                                      */
/*                                                           */
/* This function scans an image to determine how many non-   */
/* connected objects are present.                            */
/*                                                           */
/*************************************************************/


int huge ScanGraphic(HWND hWnd,int number){

        long int i,j,Pixelhit,Hval,OldHit;
        targetsG = 0;
        targetpixels = 0;

        Pixelhit=0;
        OldHit=0;

        if (!ImageSet)
                        return NULL;

        if (number == 1){
                mtarpix =
(long)(TargetNumber+1)*(long)(PixelNumber+1)*(long)sizeof(int);
                        if (pPixelRecord == NULL){
                if ((hPixelRecord = GlobalAlloc(GMEM_MOVEABLE
|GMEM_ZEROINIT,
                                                        mtarpix)) == NULL){
                                MessageBox(hWnd, "Memory Allocation Error.",
                                        "Error 8.19", MB_OK |
MB_ICONHAND);
                                        return (NULL);
                                }

                        if ((pPixelRecord = (int huge *)GlobalLock(hPixelRecord)) ==
NULL){
MessageBox(hWnd, "Global Lock Failed.","Error 9.1", MB_OK | MB_ICONHAND);
                                        return (NULL);
                                }
                        }
                        /* reset PixelRecord array  */
```

```
                    for (i=0;i<TargetNumber;i++)
                            for (j=0;j<PixelNumber;j++)

        pPixelRecord[i*PixelNumber+j] = -1;

         }

         if (number == 2){
                mtarpix =
(long)(TargetNumber+1)*(long)(PixelNumber+1)*(long)sizeof(int);
                    if (pPixelRecord2 == NULL){
                if ((hPixelRecord2 = GlobalAlloc(GMEM_MOVEABLE
| GMEM_ZEROINIT,
                                                    mtarpix)) == NULL){
 MessageBox(hWnd, "Memory Allocation Error.","Error 8.20", MB_OK |
MB_ICONHAND);
                                        return (NULL);
                                    }

                if ((pPixelRecord2 = (int huge *)GlobalLock(hPixelRecord2)) ==
NULL){
                                        MessageBox(hWnd, "Global Lock
Failed.",
                                        "Error 9.1", MB_OK |
MB_ICONHAND);
                                        return (NULL);
                                    }
                    }
                    /* reset PixelRecord array  */

                    for (i=0;i<TargetNumber;i++)
                            for (j=0;j<PixelNumber;j++)

        pPixelRecord2[i*PixelNumber+j] = -1;
         }

         Hval = (int)((float)TarVal)/4.0;

         for (i=0;i<nScanWindows;i++){
                    ScanPos = i;
                    if (number == 1)
                            Pixelhit = (int)(100.0*pImageData[i]);
                    if (number == 2)
                            Pixelhit = (int)(100.0*pImageData2[i]);

                    if (Pixelhit < Hval)
                            Pixelhit = 0;    //Pixelhit is 1 for a target

                    if (Pixelhit){
                            OldHit = CheckOldHit(i,number);
                            if (!OldHit){
                                    targetpixels = 0;
                                    IdentifyBlip(hWnd,i,number);
```

```
                                        targetsG++;
                                        if (targetsG>TargetNumber){
                                                targetsG =
TargetNumber;
                                                return NULL;
                                        }
                                }
                        }
                }

                return 1;
}


/*************************************************************/
/*                                                           */
/* function ScanImageAvg                                     */
/*                                                           */
/* This function scans an averaged anf thresholded           */
/* image to determine how many non- connected objects        */
/* are present.                                              */
/*                                                           */
/*************************************************************/


int huge ScanImageAvg(HWND hWnd){

        long int i,j,Pixelhit,Hval=100,OldHit;
        targetsI = 0;
        targetpixels = 0;

        Pixelhit=0;
        OldHit=0;

        if (pImageAvgByte == NULL)
                        return NULL;

        if (pPixelRecord == NULL){
        mtarpix = (long)(TargetNumber+1)*(long)(PixelNumber+1)*(long)sizeof(int);
                if ((hPixelRecord = GlobalAlloc(GMEM_MOVEABLE
|GMEM_ZEROINIT,
                                                mtarpix)) == NULL){
                                        MessageBox(hWnd, "Memory Allocation
Error.",
                                                "Error 8.21", MB_OK |
MB_ICONHAND);
                                        return (NULL);
                        }

        if ((pPixelRecord = (int huge *)GlobalLock(hPixelRecord)) == NULL){
                                         MessageBox(hWnd, "Global Lock Failed.",
                                        "Error 9.1", MB_OK | MB_ICONHAND);
                                        return (NULL);
                        }
```

```
        }

        /* reset PixelRecord array */

                        for (i=0;i<TargetNumber;i++)
                                for (j=0;j<PixelNumber;j++)

        pPixelRecord[i*PixelNumber+j] = -1;

        for (i=0;i<Pixel*Line;i++){
                        ScanPos = i;
                        Pixelhit = (int)(pImageAvgByte[i]);

                        if (Pixelhit > Hval)
                                Pixelhit = 0;    //Pixelhit is 1 for a target

                        if (Pixelhit){
                                OldHit = CheckOldHit(i,1);
                                if (!OldHit){
                                        targetpixels = 0;
                                        IdentifyImageBlip(hWnd,i);
                                        targetsI++;
                                        if (targetsI>TargetNumber){
                                                targetsI =
TargetNumber;
                                                return NULL;
                                        }
                                }
                        }
        }

        return 1;
}


/*****************************************************************/
/*                                                               */
/* function IdentifyBlip                                         */
/*                                                               */
/* This is a recursive function which determines the extents */
/* and location of each object in the scaned image.              */
/*                                                               */
/*                                                               */
/*****************************************************************/

int huge IdentifyBlip(HWND hWnd,int HitPosition,int number){

        int Pixelhit,Hval,OldHit;

        if (HitPosition < 0)
                return;
```

```
                Hval = (int)((float)TarVal)/4.0;

                if (number == 1)
                        Pixelhit = (int)(100.0*pImageData[HitPosition]);
                if (number == 2)
                        Pixelhit = (int)(100.0*pImageData2[HitPosition]);

                if (Pixelhit < Hval)
                                Pixelhit = 0;    //Pixelhit is 1 for a target

                if (Pixelhit){
                        /* if (!FirstPixel)*/
                                OldHit = CheckOldHit(HitPosition,number);

                                /* Record new hit  */

                                if (!OldHit){
                                                if (number == 1)

pPixelRecord[targetsG*(PixelNumber)+targetpixels] = HitPosition;
                                                if (number == 2)

pPixelRecord2[targetsG*(PixelNumber)+targetpixels] = HitPosition;
                                                targetpixels++;
                                                if (targetpixels>PixelNumber){
                                                                targetpixels =
PixelNumber;
                                                                return NULL;
                                                }

                                                /* Lower left corner */

                                                if (ScanPos == 0){

IdentifyBlip(hWnd,HitPosition+nXScanWindows,number);

IdentifyBlip(hWnd,HitPosition+nXScanWindows+1,number);

IdentifyBlip(hWnd,HitPosition+1,number);
                                                }

                                                /* Lower right corner */

                                                else if (HitPosition+1 ==
nXScanWindows){

IdentifyBlip(hWnd,HitPosition+nXScanWindows-1,number);

IdentifyBlip(hWnd,HitPosition+nXScanWindows,number);

IdentifyBlip(hWnd,HitPosition-1,number);
                                                }

                                                /* Upper left corner */
```

```
                                        else if (HitPosition ==
nScanWindows-nXScanWindows){
IdentifyBlip(hWnd,HitPosition-nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition+1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows,number);
}
                                        /* Upper right corner */
                                        else if (HitPosition+1 ==
nScanWindows){
IdentifyBlip(hWnd,HitPosition-1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows-1,number);
                                                    return 1;
                                        }
                                        /* Against left side */
                                        else if
((HitPosition%nXScanWindows)==0 ){
IdentifyBlip(hWnd,HitPosition+nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition+nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition+1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows,number);
                                        }
                                        /* Against right side */
                                        else
if(((HitPosition+1)%nXScanWindows)==0){
IdentifyBlip(hWnd,HitPosition+nXScanWindows-1,number);
IdentifyBlip(hWnd,HitPosition+nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition-1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows-1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows,number);
```

```
                                        }
                                        /* Last Row */
                                        else if (HitPosition >=
nScanWindows-nXScanWindows){
IdentifyBlip(hWnd,HitPosition+1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition-1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows-1,number);
                                        }
                                        /* First row */
                                        else if (HitPosition+1 <
nXScanWindows){
IdentifyBlip(hWnd,HitPosition+nXScanWindows-1,number);
IdentifyBlip(hWnd,HitPosition+nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition+nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition+1,number);
IdentifyBlip(hWnd,HitPosition-1,number);
                                        }
                                        /* Inside */
                                        else{
IdentifyBlip(hWnd,HitPosition+nXScanWindows-1,number);
IdentifyBlip(hWnd,HitPosition+nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition+nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition+1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows+1,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows,number);
IdentifyBlip(hWnd,HitPosition-nXScanWindows-1,number);
IdentifyBlip(hWnd,HitPosition-1,number);
```

```
                                                    }

                                        }
                        }

                        return 1;
}


/*************************************************************/
/*                                                           */
/* function IdnetifyImageBlip                                */
/*                                                           */
/* This is a recursive function which determines the extents */
/* and location of each object in the averaged thresholded   */
/* image.                                                    */
/*                                                           */
/*                                                           */
/*************************************************************/

int huge IdentifyImageBlip(HWND hWnd,int HitPosition){

                        int Pixelhit,Hval=100,OldHit;

                        Pixelhit = (int)(pImageAvgByte[HitPosition]);

                        if (Pixelhit > Hval)
                                        Pixelhit = 0;    //Pixelhit is 1 for a target

                        if (Pixelhit){
                                OldHit = CheckOldHit(HitPosition,1);

                                        /* Record new hit  */

                                        if (!OldHit){

pPixelRecord[targetsI*(PixelNumber)+targetpixels] = HitPosition;
                                                        targetpixels++;
                                                        if (targetpixels>PixelNumber){
                                                                        targetpixels =
PixelNumber;
                                                                return NULL;
                                                        }

                                                        /* Lower left corner */

                                                         if (ScanPos == 0){

IdentifyImageBlip(hWnd,HitPosition+Pixel);

IdentifyImageBlip(hWnd,HitPosition+Pixel+1);

IdentifyImageBlip(hWnd,HitPosition+1);
                                                        }
```

```
                                        /* Lower right corner */
                                        else if (HitPosition+1 == Pixel){
IdentifyImageBlip(hWnd,HitPosition+Pixel-1);
IdentifyImageBlip(hWnd,HitPosition+Pixel);
IdentifyImageBlip(hWnd,HitPosition-1);
                                        }
                                        /* Upper left corner */
                                        else if (HitPosition == Pixel*Line-
Pixel){
IdentifyImageBlip(hWnd,HitPosition-Pixel+1);
IdentifyImageBlip(hWnd,HitPosition+1);
IdentifyImageBlip(hWnd,HitPosition-Pixel);
                                        }
                                        /* Upper right corner */
                                        else if (HitPosition+1 == Pixel*Line){
IdentifyImageBlip(hWnd,HitPosition-1);
IdentifyImageBlip(hWnd,HitPosition-Pixel);
IdentifyImageBlip(hWnd.HitPosition-Pixel-1);
                                                return 1;
                                        }
                                        /* Against left side */
                                        else if ((HitPosition%Pixel)==0 ){
IdentifyImageBlip(hWnd,HitPosition+Pixel);
IdentifyImageBlip(hWnd,HitPosition+Pixel+1);
IdentifyImageBlip(hWnd,HitPosition+1);
IdentifyImageBlip(hWnd,HitPosition-Pixel+1);
IdentifyImageBlip(hWnd,HitPosition-Pixel);
                                        }
                                        /* Against right side */
                                        else if(((HitPosition+1)%Pixel)==0){
```

```
IdentifyImageBlip(hWnd,HitPosition+Pixel-1);
IdentifyImageBlip(hWnd,HitPosition+Pixel);
IdentifyImageBlip(hWnd,HitPosition-1);
IdentifyImageBlip(hWnd,HitPosition-Pixel-1);
IdentifyImageBlip(hWnd,HitPosition-Pixel);
                                        }
                                        /* Last Row */
                                        else if (HitPosition >= Pixel*Line-
Pixel){
IdentifyImageBlip(hWnd,HitPosition+1);
IdentifyImageBlip(hWnd,HitPosition-Pixel+1);
IdentifyImageBlip(hWnd,HitPosition-1);
IdentifyImageBlip(hWnd,HitPosition-Pixel);
IdentifyImageBlip(hWnd,HitPosition-Pixel-1);
                                        }
                                        /* First row */
                                        else if (HitPosition+1 < Pixel){
IdentifyImageBlip(hWnd,HitPosition+Pixel-1);
IdentifyImageBlip(hWnd,HitPosition+Pixel);
IdentifyImageBlip(hWnd,HitPosition+Pixel+1);
IdentifyImageBlip(hWnd,HitPosition+1);
IdentifyImageBlip(hWnd,HitPosition-1);
                                        }
                                        /* Inside */
                                        else{
IdentifyImageBlip(hWnd,HitPosition+Pixel-1);
IdentifyImageBlip(hWnd,HitPosition+Pixel);
IdentifyImageBlip(hWnd,HitPosition+Pixel+1);
```

```
IdentifyImageBlip(hWnd,HitPosition+1);

IdentifyImageBlip(hWnd,HitPosition-Pixel+1);

IdentifyImageBlip(hWnd,HitPosition-Pixel);

IdentifyImageBlip(hWnd,HitPosition-Pixel-1);

IdentifyImageBlip(hWnd,HitPosition-1);
                                                }

                                        }
                }
                return 1;
}


/*************************************************************/
/*                                                           */
/* function CheckOldHit                                      */
/*                                                           */
/* This function checks the PixelRecord array to see if the  */
/* current pixel has already been included.                  */
/*                                                           */
/*                                                           */
/*************************************************************/

int huge CheckOldHit(int HitPosition, int number){

        long int i = 0;
        long int j = 0;
        int OldHit;
        /* Check if already found */

        OldHit = 0;

        if (number == 1)
                        for (i=0;i<TargetNumber;i++)
                                        for (j=0;j<PixelNumber;j++)
                                                if (pPixelRecord[i*(PixelNumber)+j]
== -1)
                                                        break;
                                        else if (pPixelRecord[i*(PixelNumber)+j] ==
HitPosition)
                                                        OldHit = 1;

        if (numb' er == 2)
                        for (i=0;i<TargetNumber;i++){
                                        if (pPixelRecord2[i*(PixelNumber)+j] == -1)
                                                        break;
                                        for (j=0;j<PixelNumber;j++)
                                                if (pPixelRecord2[i*(PixelNumber)+j]
== -1){
```

```
                                                        j=0;
                                                        break;
                                                }
                                        else if (pPixelRecord2[i*(PixelNumber)+j] ==
HitPosition)
                                                        OldHit = 1;
                                }


        if (number == 3)
                        for (j=0;j<PixelNumber;j++)
                                if (pAddFirRecord1[j] == -1)
                                        break;
                                else if (pAddFirRecord1[j] == HitPosition)
                                        OldHit = 1;

        if (number == 4)
                        for (j=0;j<PixelNumber;j++)
                                if (pAddFirRecord2[j] == -1)
                                        break;
                                else if (pAddFirRecord2[j] == HitPosition)
                                        OldHit = 1;

        return (OldHit);
}


int huge PaintBlack(HWND hWnd, int window)
{
        int Ystart,line,Xpos;

        Ystart = nYScanWindows*GraphicSize+Line+40;
        line = window/nXScanWindows;
        Xpos = (window%nXScanWindows)*GraphicSize;
        line = Ystart-(line*GraphicSize)-GraphicSize;

        hDC=GetDC(hWnd);
        hMemoryDC = CreateCompatibleDC(hDC);
        hBitmapGraphic = LoadBitmap(hInst, "BlackBox");
        if (hBitmapGraphic == NULL) {
                        DeleteDC(hMemoryDC);
                        ReleaseDC(hWnd,hDC);
                        return NULL;
        }
        SelectObject(hMemoryDC, hBitmapGraphic);
        BitBlt(hDC,Xpos,line,GraphicSize,GraphicSize,hMemoryDC,
0,0,SRCCOPY);

        DeleteDC(hMemoryDC);
        ReleaseDC(hWnd,hDC);
        DeleteObject(hBitmapGraphic);
   return 1;
}
```

```
/* Statistical evaluation routine */

int DoStats(HWND hWnd) {

        int i,j,k;

         /* initialize TargetPos */

                for(i=0;i<MarkTargetNumber;i++)
                        for(j=0;j<MarkPixelNumber;j++)
                                TargetPos[i][j] = -1;

                for(i=0;i<MarkTargetNumber;i++)
                        for(j=0;j<MarkPixelNumber;j++)
                                TargetWin[i][j] = -1;

                TargetNo = 0;
                CurrPixel = 0;
                ImageTar = 0;
                ImageTarIndex = 0;

                if ((Targetdat = fopen("target.dat","r")) != NULL){
                        while (CurrPixel != -1){
                                if (fscanf(Targetdat,"%d",&CurrPixel) != 1)
                                        break;
                                else if (CurrPixel == -1)
                                                TargetNo++;
                        }
                        while (fscanf(Targetdat,"%d",&CurrPixel) != EOF){
                                if (CurrPixel == -1){
                                        TargetNo++;
                                        ImageTar++;
                                        ImageTarIndex = 0;
                                }
                                else
                         TargetPos[ImageTar][ImageTarIndex++] = CurrPixel;
                        }

                }
                else
                        return NULL;

                fclose(Targetdat);

                /* Set up the TargetWin array  */

                nScanWindows = nXScanWindows*nYScanWindows;
                PosSubX=ScanSizeX/2;
                PosSubY=ScanSizeY/2;
                CheckPos=Pixel*PosSubY+PosSubX;
```

```
                for (i=0;i<MarkTargetNumber;i++)
                        for (j=0;j<MarkPixelNumber;j++){
                                CurrPixel = TargetPos[i][j];
                                if (CurrPixel == -1)
                                        break;
                                CheckCoord=CurrPixel-CheckPos;
                                for(k=0;k<nScanWindows;k++)
                                        if(CheckCoord ==
pScanWinCoord[k]){
                                                TargetWin[i][j]=k;
                                                break;
                                        }
                        }

                CurrPixel = 0;

                if (!ImageSet){
        MessageBox(hWnd,"ImageData not
loaded.","Error",MB_OK | MB_ICONHAND);
                        return NULL;
                }

                SetCursor(hHourGlass);

                if (!CorrelFlag)
                        if (!ScanGraphic(hWnd,1)){
        MessageBox(hWnd, "ScanGraphic
error.","Error",MB_OK | MB_ICONHAND);
                                return NULL;
                        }


                /* Loop to check for targets */
                nFalseAlarmG = 0;
                TargetFoundG=0;
                for(i=0;i<MarkTargetNumber;i++)
                        for(j=0;j<MarkPixelNumber;j++)
                                if (CheckOldHit(TargetWin[i][j],1)){
                                        TargetFoundG++;
                                        break;
        }

        if (ThresholdAvgFlag){
                        targetsI = 0;
      if (!ScanImageAvg(hWnd)){
        MessageBox(hWnd, "ScanImageAvg error.",
           "Error", MB_OK | MB_ICONHAND);
        return NULL;
      }

      /* Loop to check for image average targets */

      nFalseAlarmI = 0;
      TargetFoundI = 0;
```

```
                    for(i=0;i<MarkTargetNumber;i++){
                            for(j=0;j<MarkPixelNumber;j++)
                                    if (CheckOldHit(TargetPos[i][j],1)){
                TargetFoundI++;
                break;
            }
                    }
        }

        return 1;
}

/* Function to write the stats information to the screen */

int WriteStats(HWND hWnd){

        hDC=GetDC(hWnd);

        /* Do stats for the Graphic */
        /* Print number of objects  */

        strcpy(Ntarstr,"Statistics for the Network");

TextOut(hDC,GraphicSize*nXScanWindows+20,Line+40,Ntarstr,strlen(Ntarstr));

        temptar = targetsG;
        itoa(temptar,Ntarstr,10);
        strcat(Ntarstr," objects found.    ");

TextOut(hDC,GraphicSize*nXScanWindows+20,Line+60,Ntarstr,strlen(Ntarstr));

        /* Print number of targets */

        itoa(TargetFoundG,Ntarstr,10);
        strcat(Ntarstr," targets found of ");
        itoa(TargetNo,Ntottar,10);
        strcat(Ntarstr,Ntottar);

TextOut(hDC,GraphicSize*nXScanWindows+20,Line+80,Ntarstr,strlen(Ntarstr));

        /* print number of false alarms */

        nFalseAlarmG = targetsG-TargetFoundG;
        itoa(nFalseAlarmG,Nfalsealarm,10);
        strcat(Nfalsealarm," false alarms.    ");


TextOut(hDC,GraphicSize*nXScanWindows+20,Line+100,Nfalsealarm,strlen(Nfalse
alarm));


        /* Do stats for the Image  */
```

```
        if (ThresholdAvgFlag){

                        /* Print number of objects */

                        strcpy(Ntarstr,"Statistics for the Image.");


TextOut(hDC,GraphicSize*nXScanWindows+20,Line+140,Ntarstr,strlen(Ntarstr));

                        temptar = targetsI;
                        itoa(temptar,Ntarstr,10);
                        strcat(Ntarstr," objects found.    ");


TextOut(hDC,GraphicSize*nXScanWindows+20,Line+160,Ntarstr,strlen(Ntarstr));

                        /* Print number of targets */
                        itoa(TargetFoundI,Ntarstr,10);
                        strcat(Ntarstr," targets found of ");
                        itoa(TargetNo,Ntottar,10);
                        strcat(Ntarstr,Ntottar);


TextOut(hDC,GraphicSize*nXScanWindows+20,Line+180,Ntarstr,strlen(Ntarstr));

                        /* print number of false alarms */
                        nFalseAlarmI = targetsI-TargetFoundI;
                        itoa(nFalseAlarmI,Nfalsealarm,10);
                        strcat(Nfalsealarm," false alarms.    ");


TextOut(hDC,GraphicSize*nXScanWindows+20,Line+200,Nfalsealarm,strlen(Nfalse
alarm));
        }

        ReleaseDC(hWnd,hDC);

        SetCursor(hArrow);

        return 1;
}


/*  Function to save the statistics evaluation to a file  */

int SaveStats(HWND hWnd){

        FILE *stats;
        float DetectEffI=0.0,DetectEffG=0.0,FalseAEffI=0.0;
        float FalseAEffG=0.0,PerformI=0.0,PerformG=0.0,Batchss=0.0;

        if ((stats = fopen("stats.dat","a+")) == NULL)
                        return NULL;
```

```
        Batchss = (float) max(GraphicNumber,1);

        if (!StatsHeaderFlag){
                        targetsIAvg = 0.0;
                        targetsGAvg = 0.0;
                        nFalseAlarmIAvg = 0.0;
                        nFalseAlarmGAvg = 0.0;
                        TargetFoundIAvg  = 0.0;
                        TargetFoundGAvg = 0.0;
                        SetUpLaserJet(stats);
                        fprintf(stats,"\t\t\tPAGE %d\n",Page);
                        fprintf(stats,"\nName of weight file ... %s\n",NetFileName);
                        fprintf(stats,"Starting file ... %s\n",FileName);
                        fprintf(stats,"The number of processed files ...
%2.0f\n",Batchss);
                        fprintf(stats,"The Network threshold = %d\n",TarVal);
                        fprintf(stats,"The Scan-to-Scan threshold   =
%d\n",ThresholdVal);
                        fprintf(stats,"The number of identified targets =
%d\n",TargetNo);
                        LinesPrinted=10;
                        if (CorrelFlag){
                                fprintf(stats,"Graphic correlation ON\n");
                                LinesPrinted++;
                        }
                        if (AverageEvalFlag){
                                fprintf(stats,"Average evaluation ON\n");
                                LinesPrinted++;
                        }
                        fprintf(stats,"\n");
                        if (!EvalSeriesFlag)
                                fprintf(stats,"\t\t\t\t\tObject\tTarget\tFalse
alarm\n");
                        if(FormFeed(stats))
                                fprintf(stats, "\t\t\tPAGE %d\n", Page);
                        StatsHeaderFlag = TRUE;
        }

                        nFalseAlarmG = targetsG - TargetFoundG;
                        targetsGAvg = targetsGAvg + (float)targetsG;
                        nFalseAlarmGAvg = nFalseAlarmGAvg +
(float)nFalseAlarmG;
                        TargetFoundGAvg = TargetFoundGAvg +
(float)TargetFoundG;

                        if (ThresholdAvgFlag){
                          nFalseAlarmI = targetsI - TargetFoundI;
                                targetsIAvg = targetsIAvg + (float)targetsI;
                                nFalseAlarmIAvg = nFalseAlarmIAvg +
(float)nFalseAlarmI;
                                TargetFoundIAvg = TargetFoundIAvg +
(float)TargetFoundI;
                        }
```

```
        if (!EvalSeriesFlag){
                        fprintf(stats,"\n%s\n",FileName);
fprintf(stats,"Neural net\t\t\t\t %d\t\t %d\t\t
%d\n",targetsG,TargetFoundG,nFalseAlarmG);
                        LinesPrinted +=3;

                        if (ThresholdAvgFlag){
fprintf(stats,"Scan to scan\t\t\t %d\t\t %d\t\t
%d\n",targetsI,TargetFoundI,nFalseAlarmI);
                                LinesPrinted++;
                                if(FormFeed(stats))
                                        fprintf(stats, "\t\t\tPAGE %d\n",
Page);
                        }
        }


        if (LastBatchFlag){
                        targetsIAvg = targetsIAvg/Batchss;
                        targetsGAvg = targetsGAvg/Batchss;
                        nFalseAlarmIAvg = nFalseAlarmIAvg/Batchss;
                        nFalseAlarmGAvg = nFalseAlarmGAvg/Batchss;
                        TargetFoundIAvg = TargetFoundIAvg/Batchss;
                        TargetFoundGAvg = TargetFoundGAvg/Batchss;
                        DetectEffI = TargetFoundIAvg/TargetNo*100.0;
                        DetectEffG = TargetFoundGAvg/TargetNo*100.0;
                        FalseAEffI = nFalseAlarmIAvg/nScanWindows*100.0;
                        FalseAEffG = nFalseAlarmGAvg/nScanWindows*100.0;

PerformI =
100*(TargetFoundIAvg*TargetFoundIAvg/(TargetNo*TargetNo*(1.0+nFalseAlarmI
Avg)));
PerformG =
100*(TargetFoundGAvg*TargetFoundGAvg/(TargetNo*TargetNo*(1.0+nFalseAlarm
GAvg));

                        /* print summary */

                        fprintf(stats,"\n\nSummary of results\n\n");
 fprintf(stats,"\t\t\tObject\tTarget\tFalse A\tT Eff\t\tFA Eff\tPerform\n");
fprintf(stats,"Neural net\t\t %.3f\t %.3f\t %.3f\t %.1f\t\t
%.4f\t%7.2f\n",targetsGAvg,TargetFoundGAvg,nFalseAlarmGAvg,DetectEffG,Fals
eAEffG,PerformG);
 fprintf(stats,"Scan to scan\t %.3f\t %.3f\t %.3f\t %.1f\t
%.4f\t%7.2f\n\n",targetsIAvg,TargetFoundIAvg,nFalseAlarmIAvg,DetectEffI,False
AEffI,PerformI);
                        LinesPrinted = 100;
                        FormFeed(stats);
                        LastBatchFlag = FALSE;
                        StatsHeaderFlag = FALSE;
                        Page = 1;
        }

        fclose(stats);
```

```
        return 1;
}

void AddDecimal(char *str)
{
        int len,i=0,j=0;

        len = strlen(str);

        for (i=0;i<=len;)
                        if (i == (len-1))
        Decimal[i++] = '.';
                        else
        Decimal[i++] = str[j++];

        Decimal[i]='\0';
        strcpy(str,Decimal);
}


/* The function will coorelate the objects in the two */
/* object arrays.                                     */

int CorrelGraphic(HWND hWnd)

{
        long int i,j,k,m, pixel;

        if (pCoorelate == NULL){
                if ((hCoorelate = GlobalAlloc(GMEM_MOVEABLE
|GMEM_ZEROINIT,
                                                (TargetNumber*sizeof(int)))) ==
NULL){
                                        MessageBox(hWnd, "Memory Allocation
Error.",
                                                "Error 8.22", MB_OK |
MB_ICONHAND);
                                        return (NULL);
                        }

                        if ((pCoorelate = (int huge *)GlobalLock(hCoorelate)) ==
NULL){
                                                MessageBox(hWnd, "Global Lock
Failed.",
                                                "Error 9.1", MB_OK |
MB_ICONHAND);
                                        return (NULL);
                        }
        }

        KillTheClone(pPixelRecord);
        KillTheClone(pPixelRecord2);
```

```
        /* Main fuzzy AND loop  */
        i = 0;
start:
        if (i<TargetNumber){
                for (j=0;j<PixelNumber;j++){
                        if (pPixelRecord[i*(PixelNumber)+j] < 0){
                                i++;
                                goto start;
                        }
                        else{
                                pixel =
pPixelRecord[i*(PixelNumber)+j];
                                k = 0;
                                start2:
                                if (k<TargetNumber){
                                        for
(m=0;m<PixelNumber;m++){
                                                if
(pPixelRecord2[k*(PixelNumber)+m] < 0){
                                                        k++;
                                                        goto
start2;
                                                }
                                                if (pPixelRecord2[k*(PixelNumber)+m] ==
pixel){
                                                        pCoorelate[i] =
-1;
                                                        i++;
                                                        goto
start;
                                                }
                                        }
                                        k++;
                                        goto start2;
                                }
                                i++;
                                goto start;
                        }
                }
                i++;
                goto start;
        }

        /* pPixelRecord will only contain the coorelated objects */
        /* after this loop                                       */

        i = 0;
        start4:
        if (i<TargetNumber){
                if (pCoorelate[i] == 0){
                        for (j=0;j<PixelNumber;j++){
                                pPixelRecord[i*(PixelNumber)+j] = -
1;
```