Towards the Reliable Automation of Courtesy Amount Recognition

by

Patricia J. Liu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author .....
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by .....
Amar Gupta
Senior Research Scientist
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committtte on Graduate Theses

Towards the Reliable Automation of Courtesy Amount Recognition

by

Patricia J. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

This thesis explores ways of enhancing and improving an existing system that recognizes the courtesy (dollar) amount on a bank check. Three main areas are examined. The first is the segmentation of the numeral string into its separate components. An improved addition to the current segmentation is presented. The second area of emphasis is on the preprocessing steps, where each piece of data is subjected to different structural manipulations to produce an input to the recognition neural net that is as uniform as possible for better results. The different possible orders of the preprocessing steps will be analyzed and tested to determine if the order that the preprocessing steps are done matters, and if so, which order produces the best results. Lastly, a courtesy amount locater is presented. This completely new addition to the existing system is able to find the courtesy amount on any type of check

Thesis Supervisor: Amar Gupta
Title: Senior Research Scientist

Contents

List of Figures

Chapter 1

Introduction

Approximately sixty-one billion checks are written in the United States each year, making the processing expenses of these checks a heavy financial burden. In an effort to reduce costs, banks have automated much of the process. When a check is deposited, there are only two fields that are of interest to the bank: the account number and the amount of the check. For purposes of automation, the account number is written in *MICR* (Magnetic Ink Character Recognition) ink for accurate reading by machines. For double checking purposes, the amount of the check is written both in words, called the legal amount, and in numerals, called the courtesy amount. It is the legal amount that should be read for processing, but in reality, only the courtesy amount is read, and it is processed, i.e. keyed in to the computer, by a human operator. From the computer, the amount is printed onto the bottom of the check, and the rest of the processing is then automated.

It is this human intervention in check processing that is very costly yet been necessary. The automation of processing the courtesy amount is no trivial task. First of all, this problem would be considered off-line recognition, meaning it would not have the advantages of being able to make observations of events such as the stroke sequence, speed, and pressure points that on-line recognition can [[13], [12]]. In addition, since the checks are written by thousands of different people, it would not be possible to "learn" the handwriting of the people who wrote the checks, as something like an Apple Newton, which is personalized for individual users, could.

The only thing that is not a pure disadvantage in this task is the fact that it would be numeral recognition. On the one hand, there are only 10 possibilities for what a character could be, as opposed to 26, and there is only one "word." On the other hand, since it *is* numerals, they can be in any combination. Furthermore, it is not possible to use a dictionary to help recognize the string, as it is possible in word recognition.

Faced with the problems described above, there has been great success in areas of offline numeral recognition. One example is the recognition of addresses, in particular, zip codes, on mail. Though the recognition of zip codes and the recognition of courtesy amounts seem quite similar, there is one critical difference. When processing zip codes, the length of the numeral string is known— it can only be one of a few possible lengths. Knowing the length greatly helps in the segmentation of the string into its individual characters which are then recognized. For the courtesy amount, the numeral string could be of any length, which makes segmentation very difficult at times. Furthermore, finding the location of the zip code on an envelope is much simpler, due to the rigid structure of address blocks, than finding the courtesy amount, where there is no standardized structure. Even with all of these problems, much research has been done on the automation of courtesy amount processing.

The purpose of the research done at the MIT Optical Character Recognition (OCR) Lab is to develop a system to execute the complete automation of the recognition of bank check (courtesy amount) processing. Though the group has had many successes, including a completed program called *Winbank* that can recognize a scanned in courtesy amount, there is still some ways to go before the processing of the amount can be used in a completely automated capacity. The purpose of this thesis is to improve the capabilities and accuracy of this existing application to bring it closer to reliable automation.

1.1 Current System

The current *Winbank* system scans in a specified location of the check that should contain the courtesy amount. It then takes in the scanned in greyscale image and proceeds to binarize the image to a bitmap array, consisting of only dark and white pixels. From there, the bitmap undergoes as many as three stages of segmentation to separate it into individual characters. After categorizing these characters as either a number or "other" (period, comma, or slash), grammar on the makeup and sequence of the entire numeral string is used to double check features, such as making sure that a period is not a comma by following the rules that there will always be three numerals after a comma but only two numerals after a period.

For all the characters categorized as a numeral, three preprocessing steps, normalization, thinning/thickening, and slant correction, are performed on these individual characters to make them more uniform before feeding them in to the neural net for character recognition. Then the characters, depending on the activation or confidence level results of the neural net, may go through post processing, where the structural makeup of the candidate is analyzed to make the final decision between, say, a 4 and a 9. Results of each individual character make up the final result of the recognition of the entire bitmap.

1.2 Statement of Thesis Research

The areas that the thesis work will concentrate on are the segmentation stage, the preprocessing stage, and a courtesy amount locater. For the segmentation stage, improvements will be implemented to ensure that at the end of segmentation, the bitmap will be correctly and thoroughly segmented. Though the steps in the preprocessing steps are very effective, the order in which these steps are performed is questionable as to whether it is indeed the optimal order; therefore, the preprocessing order will be tested and analyzed. Finally, a completely new addition to the existing system will be made: a courtesy amount locater, a program to locate the courtesy amount

on any type of checks.

Chapter 2

Segmentation

2.1 Introduction

Character segmentation is analogous to the chicken and egg paradox: It is not possible to recognize the individual characters of a string until the string has been segmented, but at the same token, it is not possible to segment a string until the individual characters that make up the string have been recognized [10]. The two possible approaches to solve this paradox are sequential systems and closed-loop/feedback systems. As the name suggests, a sequential system performs segmentation based on some initial hypotheses, and these segments would serve as input to recognition, with the results of the recognition being the final output. Closed-loop/feedback systems, on the other hand, perform segmentation and recognition many times, each time using the results of the previous iteration as feedback as to what can be improved in the current execution of segmentation/recognition, to finally reach a result.

Winbank is a sequential system that has three stages of segmentation. The entire numeral string is sent through the first stage of segmentation, which segments on continuous columns of white pixels, in effect producing "blobs" of continuous, connected black pixels. Due to the simplistic nature of this method for the purpose of "quick and dirty" segmentation, a blob could be an individual character, two or more touching characters, or part of a character, as might happen in the case of the hat of a 5 not connected to the rest of the 5. For further processing, each blob that is above

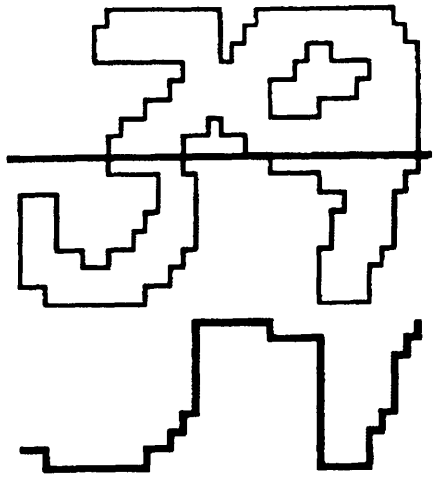

Figure 2-1: Bottom profile of a blob.

a certain width to height ratio is hypothesized to consist of two touching characters and is therefore fed into the second stage of segmentation.

The second stage of segmentation tries to separate a blob into its two individual characters. As a preparation for this, the top and bottom profiles of the blob are taken. A profile is the border between dark and white pixels of the blob. In this case, only the bottom-most and top-most profiles, ie. the outermost top and bottom borders, are found. An example of a bottom profile is given in Figure 2-1.

The blob is then separated into two blobs, using a line spanning from the minimum point in the top profile to the maximum point of the bottom profile. If in this process, the line passes through too many contours, this means that the line may have gone through a number. So, the blob is sent to the third stage of segmentation for another attempt at separation.

The third stage of segmentation uses the Hit and Deflect Strategy (HDS). HDS tries to cut through the blob by charting a path of the least amount "resistance" by following these rules for a dark pixel ([10], pg. 49-50]):

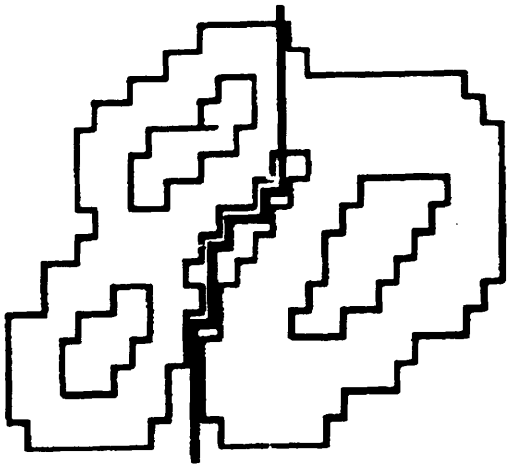

Figure 2-2: HDS segmentation of overlapping digits, an 8 and a 0

1. If the current pixel is not bordered by any white pixels, move forward.
2. If the pixel in front is going to be outside the top or bottom contour, then move forward.
3. If the pixels to the left and right are white, and the pixel in front is black, move forward.
4. If the pixel on to the left is black and the pixel to the right is white, then move right.
5. If the pixel to the right is black and the pixel to the left is white, then move left.
6. If the pixel is blocked to the left, right, and forward, move forward.

A cut on the blob using the HDS rules is shown in Figure 2-2. HDS concludes segmentation, and the resulting blobs are sent to be categorized and preprocessed for recognition.

2.2 Statement of Problem

From the description of the system, it is apparent that the current segmentation algorithm is sequential. In order to improve segmentation, the most effective change to the system would be the addition of a feedback loop to the existing benefits of initial hypotheses found in the sequential system. There are varying degrees of feedback loop that can be implemented. One possible solution is to put in a pseudo feedback loop by executing different ways of segmentation to see which way gives the best result. Here, the "best result" would be the result that generates the overall greatest amount of activation/confidence levels for each digit being recognized. The method of achieving this feedback loop would be to subject the blob to the second and, if necessary, the third stage of segmentation, for different combinations of the n most minimum points of the top profile and m most maximum points of the bottom profile. So, for example, if $n = 3$ and $m = 3$, there would be 9 different combinations and 9 different resulting blob cuts. The resulting blobs would all go into recognition. In the end, whichever cut generates blobs with the highest level of recognition would be the actual cut. This addition to the algorithm solves some of the bad segmentation cuts when the cut should not have been made from the most maximum/minimum point.

2.3 Solution

Implementation of this segmentation improvement was not simple and required major alterations. In the original application, to split a blob meant forming a path used to cut the blob, keeping track of every coordinate in the path. Then at the end of the segmentation stage, the blob would then be cut into two blobs using the coordinates of the path. When all of the blobs have been processed in the segmentation stage, they would all be passed in to recognition.

So, in order to implement the proposed change, trying out segmentation for 3 minimum and 3 maximum would be 9 combinations for each blob. This number is then compounded because each blob's combinations would have to be paired with the

different combinations of the other blobs. So in this case, for x blobs, there would be 3^x combinations to keep track of and send through recognition! Furthermore, each blob is stored as a 300x100 array, the same array size as the original input bitmap. So, the amount of memory necessary just for the blobs that need to be sent through recognition would be $300 \times 400 \times 3^x$! It is easy to see that these storage requirements would quickly exceed the amount of memory available on a typical PC.

This problem is not easily solved due to the hardcoded sequential nature of the system. It is extremely difficult to backtrack or go back to different parts of the program, and even if that were implemented, it would be fairly hard to keep track of the numerous combinations analyzed and each of their results. With these design problems already painfully apparent, the simplest solution was to redesign and therefore rewrite the program essentially from scratch, using simpler and more efficient data structures and algorithms.

The first step taken was to design the data structures. In the new design, only the original bitmap would be kept as a 300×100 array. All the blobs are stored as a structure consisting of 100 indices, one index for each row, and for each index, the starting and stopping point of the blob at that row/index. So, when using a blob, it is possible to just look at their starting and ending positions and then get the image between those points off of the original bitmap. Of course, it is necessary to use temporary bitmaps from time to time, but only one is used at any given time, and they are destroyed immediately after use. The new data structures were able to cut down the memory usage by 150 times per blob.

For preprocessing, after normalization, a blob gets resized to a 16×16 array and is kept as such. This, too, is not memory intensive, since only one blob is manipulated at a time, and at the worst, four of these 16×16 bitmaps, one for the results of each preprocessing stage, are used at a time.

With the new data structures, all of the procedures from the very beginning to recognition needed to be rewritten. The procedures following recognition did not need to be rewritten, because the output from preprocessing in both the new and old programs were 16×16 arrays. Here is a description of the new program and algorithm:

When the courtesy amount is scanned in, it is a greyscale image. This image is then thresholded, similar to the method in the original program, to a binarized bitmap format containing only white and dark pixels. Then the procedure *getblob*() takes the bitmap and performs the first stage of segmentation, producing connected components known as "blobs". These blobs are kept as the same structure as the original bitmap, meaning a 300×100 array, but are immediately converted into the new blob type. The memory for the blobs are then freed, and only the new blob type blobs are left. In this conversion process, noise that was stored as blobs are eliminated in the following way: Noise blobs will contain very few black pixels. So a predetermined blob weight minimum thresholding gets rid of the noise. These steps complete the first stage of segmentation. From this point on until the end, the blobs will be manipulated one at a time.

After the initial stage of segmentation, there is a long sequence of procedures that operate on each blob. First, physical measurements of the blob are examined to determine if the blob is comprised of one character or two touching characters. If the blob is only one character, that blob is first categorized, based on its physical makeup and measurements, to be a numeral, a dash/slash, a period, a comma, or garbage. If the blob is a numeral, then it is preprocessed, recognized, and maybe even post-processed.

If the blob has been determined to be two characters, then the top and bottom profiles of the blob are taken, with the procedure returning either the n most minimum from the top profile or the m most maximum points from the bottom profile, depending on whether the input parameter is TOP or $BOTTOM$. The results of the profiling will be used for the next stage(s) of segmentation. Looping over each combination of the result of profiling, segmentation and recognition (which also includes post-processing) are performed.

Within this loop, the blob first goes through the second stage of segmentation. If this is successful, the output is two blobs of the new blob type that come from the input blob. Since the new blob types contain only the beginning and ending points of a blob, after the actual calculation of the segmentation, it is very simple to just make

two copies of the input blob and reset either their ending points or starting points to the new coordinates.

If the second stage is unsuccessful, returning a null, then the blob is passed to the third stage of segmentation. The procedure follows the algorithm of the original version quite closely, with the exception of actually setting the output blobs' starting and ending points, which is described above.

The results of the segmentation, two new blobs, are then treated as the case of two separate blobs, each containing a single character and are processed as such. Recognition results on the two blobs are averaged together and compared with the reigning highest recognition rate. If this new result is higher, then two variables, one for each blob, are set to the result of this recognition. When all of the combinations have been exhausted, the two variables that represent the recognition result of the best segmentation are returned.

2.4 Results and Discussion

Since the new segmentation algorithm is actually an addition to the original algorithm, its performance is at least as good as the original algorithm, which is 75.4% accuracy [[10]] for splitting connected components. So, testing was done to measure of how much better this new algorithm by using data that the original segmentation algorithm failed on. Approximately 180 checks containing touching blobs were run on *Winbank* to obtain fifty pieces of data that failed on the original segmentation. This data was then tested using the new segementation algorithm. Seven out of these fifty were segmented correctly by the new algorithm. This means that there was an increase in performace by 3.8%.

This number is lower than what it should be mainly due to the aspect ratio factor. Often times, due to the large aspect ratio needed to be considered a blob made up of two characters, some blobs comprised of touching characters were not even sent through the second and third stages of segmentation due to this aspect ratio. This loose constraint allowed several touching characters to go unsegmented,

thus producing poor results. Many possible improvements to increase the accuracy of segmentation are discussed in the next section.

2.5 Possible Improvements

Even with the addition to segmentation, there are still many possible improvements that can be made. Segmentation can be improved greatly by completely integrating segmentation and recognition. This can solve some of the most troubling segmentation problems, such as abnormally sized characters, fragmented characters, and incorrect segmentation when one of the numbers in a pair of touching characters is narrow.

The problem of abnormally sized characters occurs, because the aspect ratio threshold of width to height does not accommodate these characters and therefore segments it into two blobs. With an integrated system, it is possible to realize from the results of the recognition that the segmentation was possibly incorrect. Then, a check can be made using certain heuristics, such as ones based on contours, to see if the blob may be part of one character. If two adjacent blobs fall in to this case, then they can be joined and then sent through recognition again to see if the results improved. On the other hand, if only one blob falls under this case, it should be joined it with the adjacent blobs, one at a time, to examine the results of recognition.

When fragmented characters occur, a basic solution to the problem is to follow the same procedure as is done when abnormally sized characters are segmented. Some further improvements can be made as well. For example, the most common fragmented character is a 5: when the hat of the 5 is not connected to the rest of the 5. Since this is the most frequently occurring problem of this type, solutions to take care of this particular case should be implemented. After segmentation, the hat of the 5 could possibly have been classified as a dash. One way to prevent this is to look at the vertical location of the proposed dashes. A true dash usually occurs near the midline of the text, but the hat of a 5 is going to be closer to the top of the text. So, when a blob that looks like a dash but is located higher up, joining can

be used immediately. If the hat of the 5 is not classified as a dash and is passed to preprocessing, it is possible look at the aspect ratio and then the amount of rotation needed to achieve its narrowest width. If that angle is greater than some prespecified amount, it could be joined with an adjacent blob to be fed into recognition.

When two touching numbers are narrow, the blob may not come up as a candidate for segmentation and might be passed directly to preprocessing and recognition. In order to alleviate this problem, a possible solution would be to adjust the aspect ratio threshold so that more candidates for segmentation are generated. The downside is that the better this solution works, the greater the possibility that large characters will be segmented, but this is all right, if the solution to the two previous problems mentioned above are implemented as well.

Another improvement to segmentation would be adding the capability to segment a blob into three parts if necessary. Currently, segmentation only segments a blob into two parts, but often times, there are three characters that are touching. In these cases, the blob is doomed from the start.

Chapter 3

Preprocessing

3.1 Introduction

The purpose of preprocessing is to manipulate the blobs to be as uniform as possible before feeding them in to the neural net for recognition in order to obtain more accurate results. To achieve this uniformity, blobs that have been classified as a numeral go through three stages of preprocessing: normalization, thinning and thickening, and slant correction. Normalization scales a blob to a 16×16 size array. The resulting image is then thinned to a thickness of only one dark pixel wide. This is then thickened to a uniform thickness. Lastly, the image is slant corrected. Slant correction straightens the image to make it as upright as possible. After the three stages, the image is fed in to the neural net for recognition. An example of preprocessing is given in Figure 3-1.

3.2 Statement of Problem

Though these preprocessing steps have proven to be very effective, the order in which these steps are done is questionable. To test whether the current sequence is indeed the best, it is necessary to test all possible orders and examine their results. The one assumption that can be made is that thinning will always be done immediately before thickening, which makes the number of combinations to test only nine.

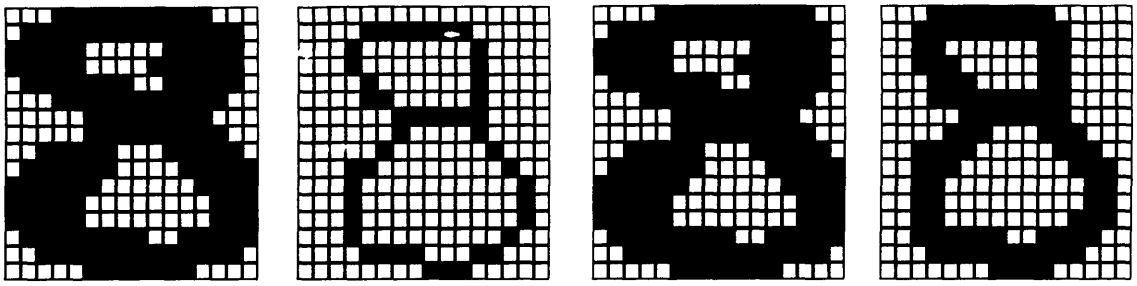

Figure 3-1: A segmented digit after normalization, thinning, thickening, and slant correction

The testing of these possible sequences is not as straightforward as it seems. It is necessary to change numerous aspects of the program. Since in the original program normalization comes first, the size of the blob that the subsequent preprocessing steps work with is exclusively 16×16. With the testing, it is not guaranteed that the size of the blob is always going to be 16×16, since normalization will not always come first. So, it is necessary to adapt the preprocessing steps to take in as input and work with any size array.

Another major problem is the neural net for recognition. In the original program, the weights used for the neural net were all the results of training by data that were preprocessed in the same way that the test data was to be preprocessed, namely normalized, thinned/thickened, and then slant corrected. Now, when any other sequence of preprocessing is used, the neural net needs to be retrained with data that has been processed in the same way that the test data will be.

Lastly, with the necessary massive amount of retraining and testing, it would be beneficial to automate these processes. Otherwise the training and testing would require many more hours to manually type in commands for each of the thousands of data necessary for the entire process.

3.3 Solution

Given the problems described in the previous section, a program to test the preprocessing order was developed. There were numerous steps in testing the preprocessing steps. It was decided ahead of time that the data for both training and testing would come from a CD of data containing thousands of handwriting samples, released by the National Institute for Standards and Technology (*NIST*). This data is organized as a single row of 128 numbers, each of the numbers represented as 128×128 bitmaps. For each file containing the images of the numbers is a related file that contains the numbers that make up the image file. So, fortunately, with this format, automation would not be difficult.

The first step to making the *NIST* data usable was to convert the image, which was in a special *NIST* graphics format, to a binarized bitmap format. In order to do this, using a procedure that *NIST* provided, the height, width, and data were read off of the image file. Using the height and width, an array is allocated to the size of the image. Then each pixel of the image is binarized to 0 or 1.

Given that the structure of the image array is comprised of individual characters that are 128×128, a block of 128×128 is copied into a temporary bitmap of that size. This "blob" will now go through preprocessing. Since normalization does not always come first, each of the preprocessing steps uses the blob as a 128x128 array for manipulation. Naturally, normalization was easily adapted to input and output any size array. (In this case, the output of normalization is a 128×128 array with an image occupying only a 16×16 space.) Thinning and thickening were easily adapted so that it was possible to pass as a parameter the size of the bitmap into the procedures.

On the other hand, slant correction had to be changed dramatically, though the basic concept was the same as the original. Since in the original procedure, the input to slant correction was always a 16×16 size array, the temporary array for performing slant correction was 72 pixels wide to accomodate for possible shifting that may occur during rotations. With the new slant correction, the input is not necessarily 16×16, making it difficult to come up with the correct sized temporary array. The solution

```
3
0000111111100000
0111111111111000
1111100000111100
0000000000111100
0000000000111000
0000000011111000
0000111111111000
0000111111111110
0000010000011111
0000000000000111
0000000000000111
0000000000001111
0000000000011111
0000001111111110
0111111111111000
0111111111100000
```

Figure 3-2: The format used for training the neural net

to the possible shifting was to just shift back each pixel as they were calculated.

Since normalization is always performed, the output bitmap of any preprocessing sequence would be a 128×128 bitmap with the image occupying a 16×16 space. This output is then trimmed of its extra rows and columns by copying only the 16×16 space to a 16×16 array. This array can now be sent to either recognition or training without needing any changes to the original recognition and training code.

After preprocessing, training and testing take two different paths. Training the neural net requires a single file consisting of the number that the image is supposed to represent and the image itself represented by a 16×16 square of 0's and 1's (see Figure 3-2). So for each character, the related file that contains the list of numbers in the image is accessed and the number represented by the character is written to the output file. Then, the array that was the result of preprocessing is written to the output file as text.

As for testing, after preprocessing, the bitmap is fed into the neural net for recognition. If the activation rates are not high enough, the bitmap will be fed into

post-processing. At the end of these processes, the number that the image is supposed to be (accessed from the related text file of numbers) and the output of the recognition and/or post-processing are written to the output file and compared. In addition, there are three structures used to prepare a summary of the testing section: "*uncertain*", "*right*", and "*wrong*". For each of the structures, there are 10 indices or boxes. If the numbers are equal, then the "*right*" structure at the index of the number is incremented. If the output of the recognition/post-processing is a * then the "*uncertain*" structure at the index of what the number is supposed to be is incremented by one. If the two numbers differ, then the "*wrong*" structure at the index of what the number is supposed to be is incremented by one. The results in the structures are displayed on standard output.

3.4 Final Version of Programs

The final training program takes in an input file and an output file name. Given the input filename, the program constructs the filename of the related file that contains the list of numbers in the image. From there, the image is converted to a usable bitmap form. Then each blob goes through preprocessing. Lastly, the number that the image is supposed to be, found in the related file, and the image are written to the output file.

The testing program also takes in an input file and an output filename, and it also constructs the name of the related file. After image conversion, each blob goes through preprocessing, recognition, and perhaps post-processing. The number that the image is supposed to be along with the result of recognition/post-processing are written to the output file. Lastly, the number of *right*'s, *wrong*'s, and *uncertain*'s for each digit are displayed.

3.5 Method of Testing

With the training and testing programs tested to be working correctly, it was time to do the actual testing of the preprocessing orders. Since about five hundred samples for each digit would be needed for training and another five hundred for each digit would be needed for testing, being able to automate the processes was very important. The training and testing programs were already able to automatically generate results with one input file and one outfile. To take the automation one step further, the programs were modified to take in any amount of arguments greater than 2, for one output file and at least one input file. All the arguments before the last one would be considered input files, and the last argument would be the output file. Only one output file was needed since the training set required that all the data be in only file anyway. As for the testing, the output file was just a log of the testing session, so it did not matter very much whether it was kept as one file or many.

To facilitate entering in the input files, Unix commands were used to list the files needed and then the listing was directed to a file, thus making an input file containing the names of many files. When the time came to perform testing, another Unix command was used to feed in the input file. Since there was only one output file, the filename was just typed in. So, the overall effect of the command was the same as having many filenames all typed in as arguments and then the one output filename.

The method of automatically training and testing data described above is a fairly accurate account of how it was actually done. One unaccounted deviation was the fact that when each file contains 128 numbers, and training or testing requires nearly 50 files, there gets to be some memory problems. So, in actuality the files were operated on using groups of 10. These resulting files, if they were for training, were then concatenated together to form the final training set file. For testing, since the number of *right*'s, *wrong*'s, and *uncertain*'s was more important, these results from each set of ten were entered into a spreadsheet and manipulated from there.

3.6 Data

Before testing on unfamiliar data, it was necessary to first use the training data as testing data to make sure that everything had been done correctly and that the neural net was functioning correctly. Under perfect conditions, the results of the tests would all be 100% correct recognition. Given this, here are the results of testing on the training set:

slant correction-normalization-thinning/thickening	90.90%
normalization-slant correction-thinning/thickening	88.67%
thinning/thickening-normalization-slant correction	98.63%
thinning/thickening-slant correction-normalization	98.63%
normalization-thinning/thickening-slant correction	93.69%
slant correction-thinning/thickening-normalization	98.18%

Since the results are acceptable, the testing was performed on unfamiliar data. Two sets of testing were done. The first one is the testing without postprocessing and therefore the absence of the uncertainty possibility; and the second one with postprocessing and the uncertainty possibility, meaning that if the activation rates for the blob was low, then it was passed into post-processing. The first set of tests shows the raw results of recognition on images straight from preprocessing. A comparison of the two sets can provide some insight as to the confidence levels of the recognition. Naturally, the two tests were done on the same data. Here are the results:

slant correction-normalization-thinning/thickening	86.45%
normalization-slant correction-thinning/thickening	84.81%
thinning/thickening-normalization-slant correction	91.67%
thinning/thickening-slant correction-normalization	93.42%
normalization-thinning/thickening-slant correction	88.28%
slant correction-thinning/thickening-normalization	92.92%

(with uncertainty)

slant correction-normalization-thinning/thickening	88.28%
normalization-slant correction-thinning/thickening	87.71%
thinning/thickening-normalization-slant correction	93.76%
thinning/thickening-slant correction-normalization	94.97%
normalization-thinning/thickening-slant correction	92.61%
slant correction-thinning/thickening-normalization	94.80%

3.7 Discussion

Clearly, these results, which proved that normalizing then thinning/thickening, and lastly slant correction is not the best order, were not expected, but upon further consideration, the result may not be so surprising. In the sequences that generated the two best results, normalizing was always done last. fact may be correlated with their good results. Normalization in *Winbank* always scales the bitmap down. The good part of normalizing to a smaller image is the fact that garbage is cleaned up, but the bad part is that much of the details of the original bitmap can be lost or distorted. So, when the bitmap is normalized first, the thinning and thickening on the normalized bitmap may be accentuating the wrong details. It may be better to do thinning and thickening on the natural, original bitmap so that nothing is misinterpreted or lost. For slant correction, it does not seem to matter as much when it is done, because the details of the bitmap are not changed, only the angle of the character. Also, from analyzing the data, it is clear that though allowing uncertainty gives better results, the overall relative accuracies among the orders stayed about the same.

Chapter 4

Courtesy Amount Locater

4.1 Introduction

So far it seems that only improvements in accuracy are needed to reliably automate the *Winbank* system, but that is not the case. There is one feature that the system completely lacks, and that is a courtesy amount locater. Most of the emphasis in the research has been put on what occurs after getting the courtesy amount, but one of the biggest improvements that needs to be made is in locating the courtesy amount.

The current *Winbank* system scans in "dummy" mode. This means that a check must be put on the scanner in the top right hand corner, because the scanner only scans a fixed region. The most obvious problem with running in dummy mode is that *Winbank* cannot be automated if running it requires someone carefully placing the check in a specific location, or else the scanned in image will be incorrect, making the result of the entire program execution wrong.

Upon further examination, there are other deeper problems. Due to the dummy mode, incorrect check processing will occur when checks are of a different size than expected or when the courtesy amount is not at the expected location. The majority of the handwritten checks in the United States are of uniform size, but business checks, both typed and handwritten ones tend to be larger. Foreign checks come in all different sizes.

Another problem is the courtesy amount itself. Although the majority of U.S.

handwritten checks are of the same size, the location and size of their courtesy amounts may differ slightly. If there is too much deviation from the expected coordinates, the check will not be processed correctly. Foreign checks differ greatly in both where the courtesy amount is located and its appearance. So, by running the system in "dummy" mode, there is a very large fraction of checks that cannot be processed by the current *Winbank* system.

A courtesy amount locater, which would be implemented as a program that looks for the courtesy amount, could make it possible to process any check with a courtesy amount. This courtesy amount locater must be able to take in as input any kind of check. So, not many assumptions can be made on the check. For instance, it is not guaranteed that there is a box around the courtesy amount, nor is there always going to be a dollar sign in front of the amount, because foreign checks may not have either feature. Not making specific assumptions will result in a universal locater that would take the system many steps further in automating check processing.

4.2 Techniques and Approaches

There are two main approaches that can be used in the extraction of data, in this case numeral text, from a check: top-down and bottom up. In the top-down approach [2], as the name suggests, the system tries to first get a general idea for the structure of the check. Then it will break up the check into regions, then lines, and lastly words, maybe even characters. A system implemented with a bottom-up approach [[4], [11], [8]], on the other hand, first tries to locate the characters. From the characters, it forms words, then lines, and so on. A top-down approach is usually less computationally intensive, because they are used mainly for larger image blocks where there are fewer of them in number. The disadvantage of a top-down system, however, is that it is not as robust as a system implemented in a bottom-up approach. There are several existing techniques that have been developed for both top-down and bottom-up approaches.

4.2.1 Bottom-Up Techniques

Stroke Method [11] The stroke method is an algorithm used to find the location of characters. Each line of the image is examined. Whenever a black pixel follows a white pixel, all of the black pixels until the next white pixel are stored. By examining these runs of black pixels, it is possible to determine the probable average thickness of the characters. A certain number of these runs that are vertically connected to each other are combined into strokes. Connected or very adjacent strokes are then combined to form characters.

Connected Component Method [[4], [8]] The connected component is another way of finding characters. In this method, for each black pixel, an extension using depth first search and either 8-connectivity or 4-connectivity heuristics outputs an component that is connected to the first black pixel. 8-connectivity and 4-connectivity refers to looking in either the 8 or 4 directions immediately adjacent to the pixel in question. The output of this algorithm produces connected components that may be part of a character, many characters, or noise. By examining the physical attributes, such as the height-width ratio, black pixel density, and size, each component is either determined to be valid or thrown out.

After finding the characters in the image, the characters need to be grouped into words and then lines. The simplest way to do this is to look at the proximities between the characters and form the words based on the different proximities. Then using the location of the words, lines are formed from words that are essentially on the same horizontal coordinates. If there is skew in the scanned image, a Hough transform [8] can be performed on the centroid of the characters to obtain the words that lie on the same line.

4.2.2 Top-Down Techniques

The two most useful top-down techniques for extracting data are smearing and smoothing. Both of these techniques are solutions to finding blocks of text on an im-

age. Smearing [2] an image means extending each pixel a factor horizontally. Words, which are made up of black pixels, will generate blocks of black as a result of smearing.

Smoothing [2] is analogous to scanning at a low resolution. By smoothing, each pixel value represents the "average" of a larger region. This means that a high concentration of dark pixels will result in the pixel value in the smoothed image to be dark. The output of smoothing will contain areas of dark. By picking out the dark regions, it is possible to locate the characters, words, or lines from the image.

4.2.3 Feature Extraction Techniques

Filters Different filters can be used to accentuate changes in an image. For example, a simple filter is one that looks for horizontal changes in a greyscale image. The filter would be an array that is made up of 3 rows and a variable number of columns, preferably 3 or greater. The first row is comprised of all 1's; the second row is comprised of all 0's; and the third row is comprised of all -1's. When this filter is passed through an image, it acts like a frame on each pixel. So, say if the filter is 3×3, then the 3 pixels above the pixel in question are individually multiplied by 1; the two pixels to the right and left of the pixel in question and the pixel itself are multiplied by zero; and the 3 pixels below the pixel in question are multiplied by -1. These nine values are then added together to produce the value that would go in the coordinate of the pixel in question.

For example, if the filter was being passed through an area where there are no changes in the image, such as a solid color, with all the pixel values being the same, then passing the filter through that region means that the number of pixels of a value multiplied by 1 and the number of pixels of the same value multiplied by -1 would be the same. When all of these values are added together, the result wouldbe zero. In other words, for each pixel multiplied by 1, there is another pixel of the same value that is multiplied by -1, thus canceling each other out when added.

On the other hand, when passing the filter through a region that contains different pixel values, the result of the filter would not be 0. Furthermore, the greater the difference in the pixel values of the rows, the higher the result of the filtering. This

is how horizontal changes are detected, while the values give insight to how much change there is in the greyscale.

Hough Transform [9] Hough transforms are used to isolate different features on an image. There are two types of Hough transforms: classical and generalized. Due to the amount of time needed to perform a generalized Hough transform because of computational complexities, only the classical Hough transforms could be viable. The classical Hough transform is used to detect regular curves, such as lines and circles. It works in the following way:

In order to find a line in an image, it is necessary to make a graph of slope versus y-intercept. To generate this graph, the line that each point produces on the graph of slope versus y-intercept is plotted. (This is analogous to, given a slope and y-intercept, producing a line in the x and y domain.) What happens is that the pixels that lie on the same line in the image will produce lines on the slope versus y-intercept graph that form a cartwheel, where all the lines will pass through a point. The point that the most lines pass through represents the slope and y-intercept of the line in the image that the pixels lie on. Figure 4-1 is a histogram that shows the plot of the slope versus y-intercept for an image containing pixels that make up a straight line. The peak of the graph shows where the point that most of the lines passed through and is the coordinate of the slope and y-intercept of the line on the original image.

4.3 Background Research

The discussion in the previous section on some basic types of approaches and techniques provides a good foundation for analyzing some related research. The related research that has been done may give some insight to an approach to the problem of locating the courtesy amount. In addition to an earlier system to locate the courtesy amount, some areas where related research has been done are in forms processing and the locating of the zip code on envelopes.

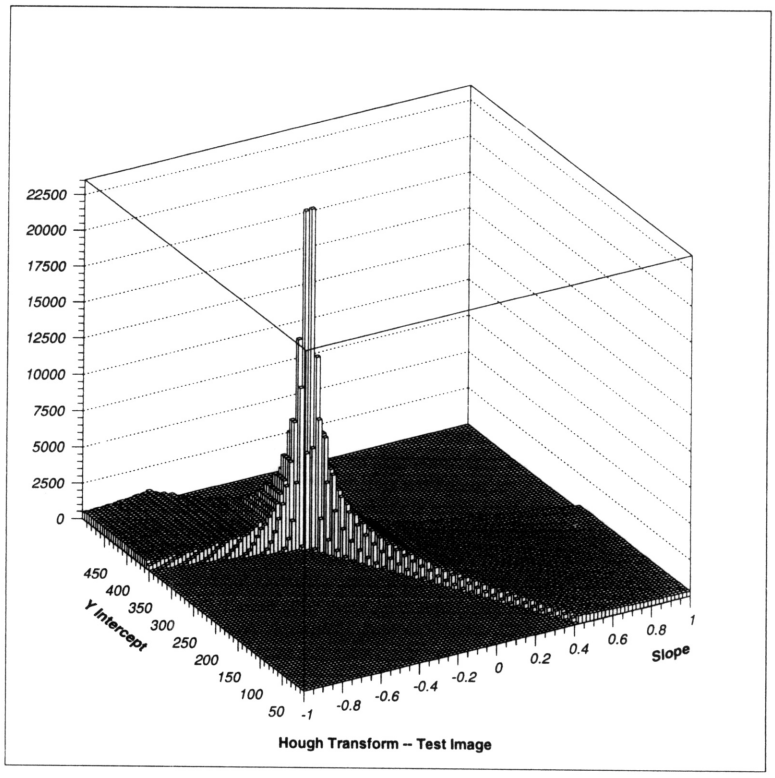

Figure 4-1: Hough transform of a test image consisting of a single straight line

4.3.1 Forms Processing

Forms processing is the processing of text from forms. In the paper on research that Richard Casey and David Ferguson [1] have done, much of the difficulties in automated forms processing are mentioned. First of all, there are many form formats. A similar can be found in trying to locate the courtesy amount, because both domestic and especially international checks come in all kinds of layouts and sizes. Other problems in forms processing include poor typing quality, many fonts, and positional errors in typing information onto the form. These problems all sound very similar to the problem that would be encountered by a courtesy amount locater: Handwritten checks are mostly written with a ball-point pen, which produce much unevenness and breaks or "skips" in the characters. Furthermore, handwritten checks are much more difficult to process in all respects than typed or printed checks. Since there are so many checks processed each year, there is naturally a wide variety of "fonts". Positional errors in the information occur as well, when the dollar amount is written exactly on or even running over the edge of the bottom line of the courtesy amount box/line.

The solution that Casey et al. gives to forms processing, something that is in many ways similar to the locater problem, deals with the difficulties mentioned above with some simple solutions. Based on the assumption that the form format has been prespecified, to deal with many form formats, it was necessary to use several methods, such as finding the form number or getting the form layout, to find out exactly which form, out of a list of known forms, it was. To deal with the positional errors in typing information, an algorithm was used to remove the line from the text, and the characters that intersected the line were reconstructed and filled in with the missing pixels. The solutions to the other problems are not as relevant to the locater problem since they deal more with what happens after getting the data from the form, in other words, the optical character recognition aspect.

Though the problem that Casey et al. solved sounds very similar to the courtesy amount locater problem, only certain aspects of the solution may be useful here. First, all the forms are prespecified. Accomplishing prespecification in a locater is

very difficult, mainly due to the fact that bank check formats are not standardized, making the range of possibilities varied. On the other hand, the solution to solving the positional errors in typing information could be very useful.

Another publication presents research on processing forms that are in a more unconstrained environment, where the format of the forms has not been prespecified, and, the forms are not necessarily typewritten [14]. The way of locating the information on the forms is done is using graphical methods. A form is preprocessed by segmenting the form, using connected components, and categorizing each component as a box, line segment, or text. The line segments could be part of the box, such as when small vertical lines separate the box making compartments for individual characters; or it could be part of the box or text. To obtain the data, it is assumed that information, which were categorized as text, is contained in the boxes.

The solution for unconstrained form processing is even less helpful in the courtesy amount location than before. Although segmenting and categorizing a check may be viable, it cannot be assumed that the courtesy amount will be located within a box. In U.S. checks, courtesy amounts are usually in a box, but for foreign checks, it is not always the case. So, looking for a box is not a dependable choice in courtesy amount location.

4.3.2 Zip Code Location

To locate the address block on envelopes, Jain et al. [6] use special filters called Gabor filters. An image of the envelope is passed through a set of eight Gabor filters. Depending on the input parameters for the angles, the output produces areas of black, which is the output for the background area, and white bars representing the text vertically skewed in directions that are specified by the input angles. At certain angles, the white bars show up very clearly on a sea of black. This filtered image is filtered once again, using a simplified filter. The final result consists of blotches of white on a sea of black. The areas that contain the blotches are then examined for the text. This method, though accurate, is quite time consuming due to its complexity.

Another method of locating address blocks is by smearing [2]. When a (binarized)

image is smeared, this means every pixel is extended horizontally by a factor. The result is, depending on the smearing factor, words or lines of text are smeared into regions of black. The horizontally smeared image is then vertically smeared, producing blocks of black. These blocks are then examined using heuristics on the dimensions and other physical attributes of the blocks, as well as the location of the blocks on the image. In the case of the courtesy amount locater, smearing a check will not work when there is a background image on the check. A background image would not only produce undesired blocks but will also distort true blocks of text.

The successful location of the address blocks is not enough to be beneficial, since the purpose of examining envelopes is to sort them. Locating and eventually recognizing the zip code is the most helpful step taken for initial sorting and therefore, most of the research has been concentrated in this area. One method of locating the zip code was presented by Edward Cohen et al. [3]. In this method, after determining the individual lines of text on the address block, each line is segmented into several possible word groupings. Each possibility is then categorized as text, city, state abbreviation, digits, digits-dash, ZIP code, barcode, P.O. Box, or noise. The confidence levels of the categorization plus the sensibility of the categories that together make up the line determine which word grouping to use. The "word" that is categorized as a ZIP code is recognized using OCR techniques. If there is some further ambiguity as to which of the "words" is the ZIP code, physical context is used, i.e. the most bottom line would most probably contain the ZIP code. Since the methods used here are mostly based on context and assumptions on physical locations of the ZIP code, the techniques can not applied to the case of the courtesy amount locater, where context does not exist, and assumptions of the location of the amount cannot be made.

Courtesy Amount Locater In addition to the related research described above, a courtesy amount locater for handwritten checks was implemented by Granowetter [5]. This locater took as input a check with black text on top of a white background. After the components, in this case, text, were extracted using a depth first search, strings were generated from these components. In order to find the correct component, sev-

eral tests were used. The first two tests determined the regularity of the components. Based on the assumption that handwritten text is less structured than typed text, which narrows down the possibilities of which string is the courtesy amount can be narrowed down. These strings are then tested using the assumptions that the courtesy amount should contain at least two characters, a decimal point may appear in a dollar amount, there are two digits after the decimal point in a dollar amount, some digits in the courtesy amount may be smaller than others, and finally, the courtesy amount is in a particular general location. The string that passes the most tests is considered to be the courtesy amount.

Though Granowetter also tries to implemeent a system to find the courtesy amount, the techniques that he uses are not particularly helpful. First of all, most checks contain some sort of background that complicate things greatly; therefore, it is not practical to assume the ideal input of a white check with black text. Also, it is not very effective to place so many restrictions on the courtesy amount. It is not guaranteed that the courtesy amount is located in a general region, nor are there always two digits after a decimal point. For example, the decimal point could be followed by a long dash to designate zero cents. The other heuristics used are even more uncertain. So, even though certain details of his system might be helpful in implementing this courtesy amount locater, the basic ideas are too specific to be effective.

Other Related Work In almost everything that has been discussed so far, the basic, almost unstated assumption was the fact that the image consists of black pixels, constructing the words, on a sea of white pixels. Though this assumption has been made in so many cases, it is not a practical assumption here for the reason that most checks, especially handwritten ones, have a background image that cannot be easily ignored. This problem brings up another area of research which may have been helpful in the location of the courtesy amount: the separation of text from graphics images. While this topic sounds like a research area that may provide some suggestions as to how to obtain a binarized check image, without the background pictures, it turns out that it was not. Several algorithms have been published for

extracting text from graphics [7], [4], but these algorithms were completely based on the assumption that the input are images which have text that do not touch the graphics. In the case of bank checks, text is written on the background pictures. So, research in the separation of text from graphics resulted in a dead end.

4.4 General Description of the Courtesy Amount Locater

Having examined different techniques and information on related research, it is time to proceed with a solution to finding the courtesy amount. In this section, a general solution, describing the basic ideas and motivations, will be presented. The locater uses modifications of several techniques presented earlier and can be described as a hybrid system that is both top-down and bottom-up.

After a check has been scanned in, the first objective is to get rid of any background pictures, in an attempt to separate out the text from the graphics. So, the image is thresholded on dark pixels. In other words, pixels above a certain value will be set to black, and pixels below the value will be set to white. This will get rid of much of the background image. A very beneficial side effect of this is that thresholding on high pixels will bring out all of the lines on the check, since they are dark colored. Unfortunately, the lines around the courtesy amount on handwritten checks will not be made prominent in this step, because they are usually of variable, lighter colors.

However, being able to bring out and locate most of the lines on a check will be very helpful in locating the text. So, the next step is to actually locate the lines. This is achieved by first performing a Hough transform on all of the dark pixels on the thresholded image. This Hough transform will be a simple one, merely looking for lines in the image. Since the check will contain, at most, only a minor amount of skew, the possibilities of lines going through a point are limited by the narrow range of possible slopes centered around zero, the slope for a perfectly horizontal line. The output of the Hough transform will be the slope and y-intercept of the lines on the

check. The pixels that served as input to the Hough transform are then plugged in to the different equations for the lines found on the check to see if they belong to one of the lines. This will help in determining where each line begins and ends.

Since most text is written on lines, the location of the lines can help to find where the text is. This is not a very straightforward procedure, though, because, in the original image, above the lines along with the text are the backgrounds and other noise. So, to find the text, it will be necessary to first find the range of pixel values for the ink. When this is found, the original image can be thresholded again on the ink pixel values to obtain an image with the text writing.

To find the ink pixel values, a certain amount of area above the lines is analyzed. Since the ink values and the line values often overlap due to the certain darkness of the writing, the thresholded image usually contains some, if not the majority, of the text. This means that the thresholded image can help greatly in narrowing down the possible ink ranges. The coordinates of the dark pixels above the lines in the thresholded image are referenced in the original image to obtain the actual pixel values. The number of pixels from each of the values is counted. This procedure is performed on each line, and the resulting values for each line are added up. The value with the most amount of pixels is considered to be in the range of the ink pixels. So now, the original image is thresholded around this value to emphasize the writing.

This second thresholded image accentuates the text, but since the pixel values of the ink and the lines overlap, lines will show up as well in this image. It is important to remove these lines from the image in order to get the individual components of the text, because writing is often touching the line. Therefore, characters that touch a line will all be considered as one connected component, thus throwing off the analyses of individual components.

So, once the lines have been removed, connected components on the image are found. These blobs are subjected to different tests based on physical attributes to try to filter out the blobs that are not text, namely parts of the background image that have passed the ink thresholding.

Once a final set of blobs have been determined, the blobs need to be grouped

together to form related lines of text. This text can define a general region to be sent through the Winbank system for insight to which of the lines of text is actually the courtesy amount. It is not very important at this point to obtain the exact components of the courtesy amount but only the coordinates of the region where the courtesy amount is located.

There are several advantages of sending a region, rather than the individual blobs, through *Winbank*. First of all, the region can be thresholded again to make a more precise thresholded image of the writing, since there is less noise to contend with. The resulting thresholded image from Winbank will be more accurate and will bring out more specific details of the writing that the previous thresholding missed. Also, by sending lines of text in rather than individual components, this prevents situations where a date or address written with words and numerals is mistaken as being the courtesy amount, because the numerals in this mixed text line were recognized with great confidence.

To find the related lines of text that can be used to find the regions obtained by finding related lines of text, meaning that words which lie on the same line and are close together. The regions of where these lines of text are are then sent through *Winbank*. The overall recognition rate of Winbank are considered to, at the best, produce results that point to only one possible courtesy amount, and the the worst, distinguish the numerals from the words.

A very likely case where there would be a problem is when the date is written in all numerals. There are two ways that a date could be written with all numerals: 5-28 or 5/28 followed by a default 19 and then the last two digits of the year. To distinguish between the date and the amount it is easier to first locate the date, because the date is more structured than the courtesy amount. If the line contains a typewritten 19 to start of the year, with the rest of the characters on the line of different font and size, then that line is probably the date. If the candidate contains a dash that appears around the midline of the height of the text (thus preventing the case of confusing a dash with the hat of a 5), followed by five or six digits, this candidate is probably the date.

If the candidate contains a blob that has a high aspect ratio (height to width), then there are four possible things to look at. The first is to see if the height of the possible slash is taller than the rest of the characters on the same line. The second thing to do is to send this blob through recognition, and if the result comes back as a 1, then it could be a slash. The third thing is to send all of the blobs on the same line through slant correction to see how much of an angle is needed to "unslant" the blobs. The possible slash would need more of a slant correction than the rest of the blobs/characters. The last thing to examine is the location of the candidate slash. As in the case of a dash, a slash in the courtesy amount would not be followed by five or six digits. So, using all of the heuristics above, by determining which is the date, the courtesy amount can be located and recognized.

4.5 Actual Implementation

With a general idea of how the locater workd, in this section the specific details of the actual implementation of the courtesy amount locater are presented. A check is scanned in, at 300 dots per inch, and due to the nature of the scanning program, this image is automatically saved as a TIFF file, which is a color format. For convenience of later handling, this file is converted to PGM format, which is a greyscale format with values ranging from 0 to 255. This PGM file is then fed in to the locater application.

The locater application takes in the name of the file, and reads the information about the size of the image in to variables, which will be needed in the rest of the program. Also, the image data from the file is read in to an array, and it is this array that will serve as the variable for the original bitmap.

The image array is then thresholded for darker pixels so that some lines will be picked out. This thresholded image is now used for finding the lines. The basic implementation principle is to make a two dimensional histogram based on the slope and the y-intercept so that it would be possible to find out the slope and y-intercept of the lines on the check. The dimensions of the histogram can be bounded on both

axes: Since horizontal lines on the check will have close to zero slope, the range of slopes is limited. This also limits the possible y-intercepts that the lines could take one to the height (which is the y-direction) of the check. The two dimensional Hough transform is necessary, though, because it is not always guaranteed that the lines on the check will have zero slope, since the check may be skewed, or even the lines on the check may not be perfect.

To find the horizontal lines on the check, for each point in the image, the line representing all the combinations of slope and y-intercept that would contain that point, with variables slope and y-intercept is found. The histogram values at the slope and y-intercept coordinates where the line would pass through are incremented by 1. Upon completion, peaks in the histogram are formed. Since these peaks are found in clusters, it is necessary to find values to represent each cluster to limit the amount of lines "found" on the check image, because all of the high values in the cluster contribute to only one line. To find a representative coordinate, all of the peak values, above a certain number, are sorted according to their histogram height. Then peaks that are within a certain distance from each other are packed together. This compounded array is a list of the peaks in the histogram. These peaks represent the slopes and y-intercepts of the lines in the image.

Now that the y-intercepts and slopes of all the lines on the image are known, each coordinate in the image is substituted in to the equation, having the form $y = mx + b$, where the m is the slope of the line, b is the y-intercept of the line, and the y and x are the coordinates that are being tried to see it they make the equation true, within a certain ϵ, to account for possible slight sitortions and rounding errors. By doing this, it is possible to find the extreme points of the lines in the image to determine where it starts and ends. Only the maximum length line is found for each slope, y-intercept point.

With information on the slope, y-intercept, and the starting and ending points of the line, it is now possible to find the pixel range values of the ink. A histogram of the different pixel values located within a certain distance above the line is made. Since the peak of the histogram is going to be mostly contributed by the ink, the

original image is thresholded on a range around this peak.

Before getting the connected components, the lines have to be removed from the image. Starting at the starting point of the line and following the equation of the line, looking two or three points above and below the calculated location of the line, it is possible to remove the line. Looking above and below the calculated line is necessary, because the output of the Hough transform may give a line that is slightly off, and also, a line is always more than one pixel wide. Since writing often times are right on the line, it is necessary to reconstruct the lettering if it runs through a line. This is done by looking a few more pixels above and below the line, for each x-coordinate, to see if there are "extensions" from the line. If there is an extension, then the entire thickness of the line for that x-coordinate is filled back in.

Now that the lines have been removed, it is easier to obtain the text. This can be done using the *getblob*() procedure (see Chapter 2.3) from segmentation that gets connected components. The *getblob*() was altered in many respects. First of all, since the ink thresholding may produce a picture where the writing has skips in it, the tolerance of the *getblob*() was changed to be much more loose. This means that pixels that were separated by as much as four white pixels are still considered connected. Also, since *getblob*(), which contains a recursive function, is performed on such a large area, a limit had to be placed on the number of levels of recursion, to prevent running out of memory and segmentation faulting.

In addition, *getblob*() was altered so that each blob, before saving it (and using up so much memory), was analyzed for its physical characteristics. The first test on the blob was its dark pixel weight. If the blob did not contain a certain number of dark pixels, it was considered as noise and would not be saved. Also, if the blob was too heavy to be text, it is thrown out as well. Then the blob's width to height ratio was found, and if this ratio was too big, the blob was thrown out. By measuring the aspect ratio, it was possible to throw out text written in cursive. The last test was on the volume/perimeter ratio. The volume is considered to be the number of dark pixels in the blob, and the perimeter is any dark pixel bordered by at least one white pixel. Since writing is basically made up of thin lines, it has a low volume/perimeter

ratio, but dark massive blobs, under the maximum weight limit, resulting from the background images that are in the range of the ink (and therefore had passed the ink thresholding) will have an extremely high volume/perimeter ratio. If a blob passes these tests, it will be saved as a blob. This is how an array of blobs is made.

The next step is to find which blobs are part of the same line of text. Starting from the first blob in the array, its four corners are stored as the region for the current line of text and is then compared with all of the other blobs. If the other blob and the current text line overlap in the vertical direction at all, then the horizontal distance between them is considered. It is important to note that they only have to overlap and not lie on the same horizontal line, because the possible skew of a check and other factors make it extremely plausible that the text do not lie on the same horizontal line.

If the current text line and the blob are within a certain distance from each other, then they are examined to see if they are located within two times the height of the current line of text, which is the height of the tallest blob in the current text line. The first proximity test takes care of the situation where the blob that sets the height of the current text line is much taller than the rest, making blobs that are very far away be considered as part of the same text line. A third test is performed to see if the blob or the text line completely envelopes the other. This test is useful in situations such as when there is a border around the checks. When a border makes it to this stage and is being considered for part of a line of text, everything within the border, meaning all the blobs, will be considered as one line, since all the blobs and the border "overlap." If a blob passes these proximity tests, it is considered part of the same line of text. The region for the current line of text is then altered to include this blob, the height of the text line is readjusted to reflect the height of the tallest blob in the text line, and the blob is considered as "done," so it will not be considered later on.

The last tests performed are on the height and length of the text line. If the height is below a certain number, it is thrown out. This test takes care of the smaller, typed text, already on the check, that had passed the ink thresholding. Limiting

the length of the text line helps to weed out text lines that are obviously too long to be the courtesy amount. Text lines that pass these tests are then candidates for the courtesy amount. The four corners of the text lines are then fed in to *Winbank*, where the region is then thresholded again, for a more precisely thresholded image to bring out the writing. Also, in *Winbank*, *getblob*() is performed again, on the new thresholded image and with a lower tolerance, which means that dark pixels can only be at most one white pixel apart for them to be considered as part of the same connected component. The results of *Winbank* are compared for each region, and if there is more than one good result, the heuristics described in the previous section are used to distinguish the numerals. The final result is the recognized courtesy amount.

4.6 Locater in Action

To further the understanding or conceptualization of the locater that has so far been described only in words, it may help to see an example of the locater used on a check. This section contains some images of a check at different stages of the locater. First, a check is scanned in as a greyscale image as in Figure 4-2.

The check is then thresholded for dark pixels, and the result is shown in Figure 4-3. Notice that parts of, but not all, the background picture has been thresholded away and that the lines and text are very prominent in this image.

After this stage, the lines on the check are found using Hough transforms. The histogram of the slope versus y-interecept that is used to find the lines is found in Figure 4-4. Due to so much noise and complexity in the check, this histogram does not contain the clean peaks that that the histogram, of a single line, in Figure 4-1 does.

The values of the ink is determined and then thresholded. Lines on the image are subsequently removed, and connected components are found using getblob. Text lines are formed from the resulting blobs. The text lines/regions that pass the height and length test are then sent as input to *Winbank*. Figure 4-5 shows the final text line candidates. The output of *Winbank* is used to help determine which one of these

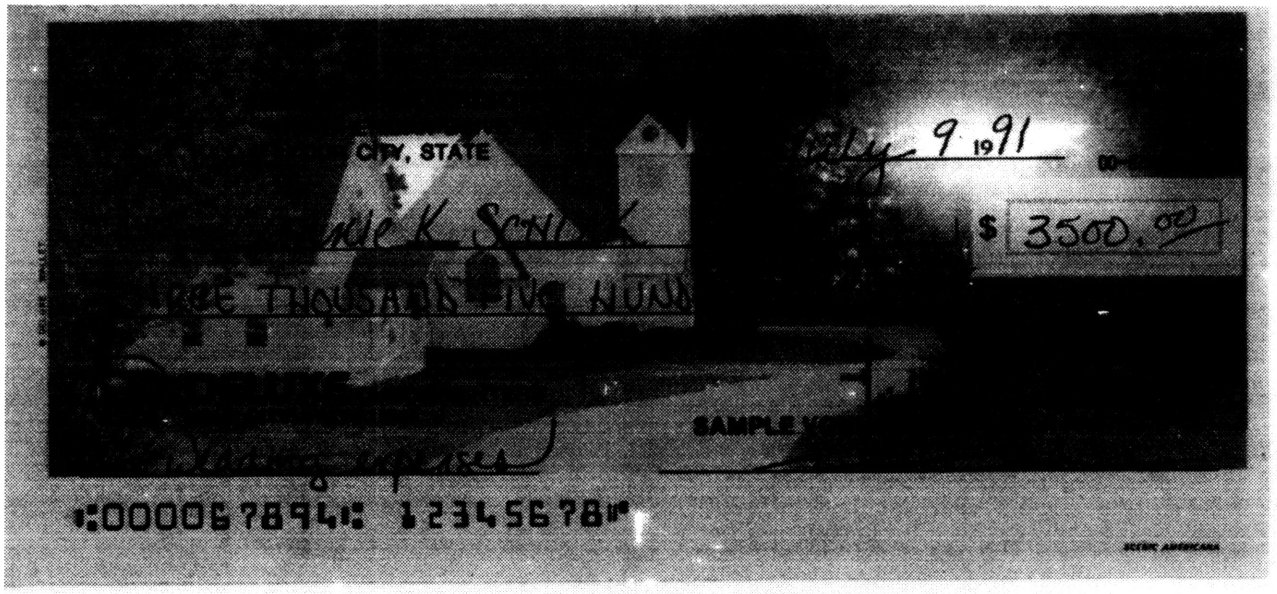

Figure 4-2: Raw grayscale image of check immediately prior to scanning

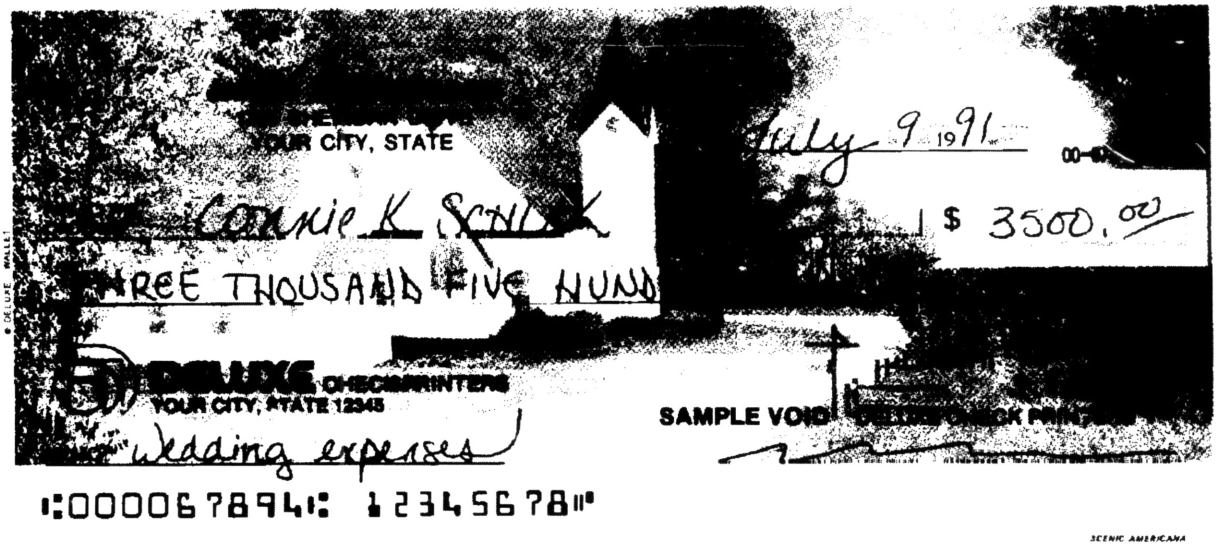

Figure 4-3: Thresholded image

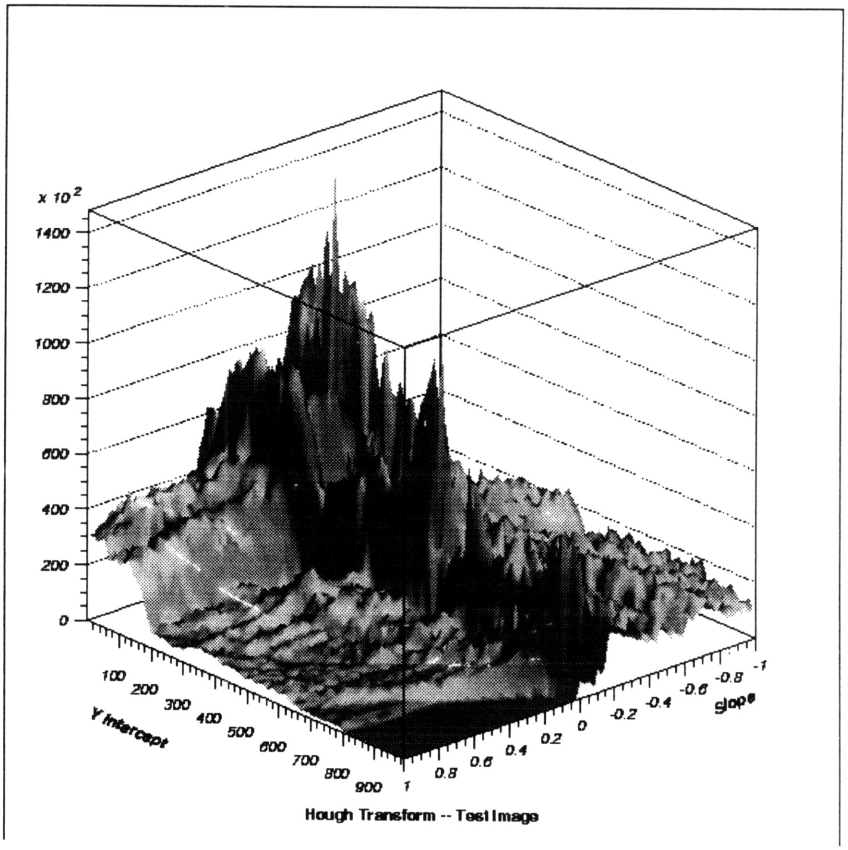

Figure 4-4: Hough transform of thresholded check image

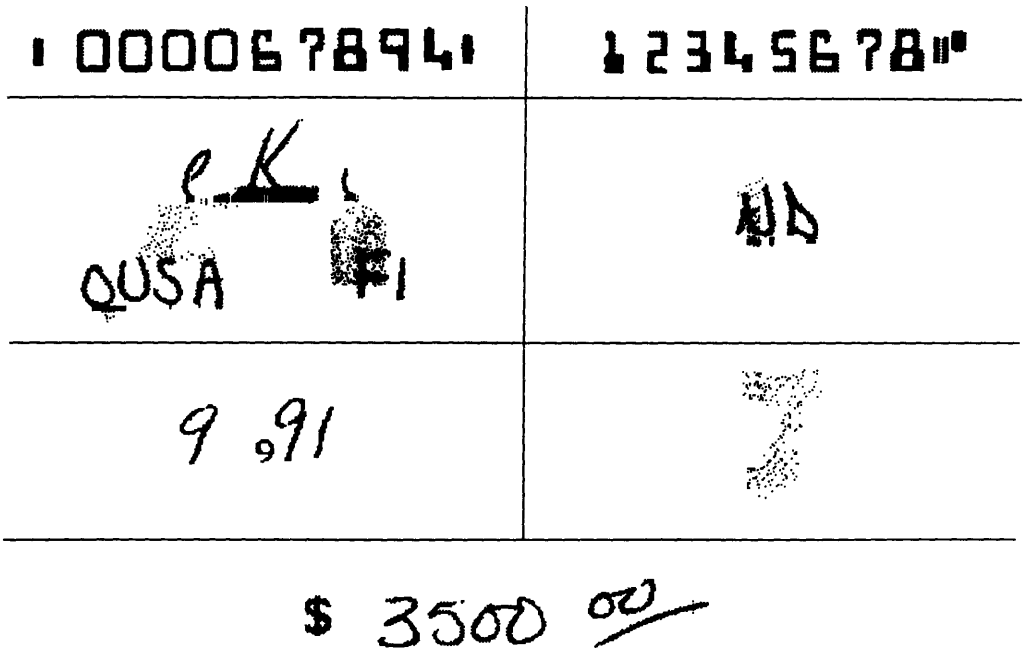

Figure 4-5: Locater output for input into recognition

candidates is the actual courtesy amount.

4.7 Testing

Testing the locater was done on various checks, both U.S. and foreign. The checks varied in size, location and appearance of the courtesy amount, types of background pictures, color of ink for both the lines on the check and the writing, and thickness of the writing. Due to time constraints, the integration of the locater and *Winbank* was not implemented, though since the accuracy of *Winbank* is high, non-numerals sent in through *Winbank* would have low recognition rates whereas numbers would have high recognition rates. The results presented are therefore a preliminary measure of the accuracy of having located the courtesy amount region, among a couple of other regions, as one to send in to *Winbank*.

For the U.S. checks, 100 different checks were used for the data. Out of this 100, the courtesy amount was correctly found to be a text line (and region) on 98 of the checks, thus producing an accuracy rate of 98%.

Foreign checks included checks mostly from England and the Isle of Man. Due to resource constraints, only fourteen of these checks were tested. Out of these fourteen, eleven of them had identified the courtesy amount as a text region, makeing the accuracy rate 79%.

4.8 Discussion

Thought these are not the final results, these preliminary results are indeed a good sign that the integration with *Winbank*, along with some minor adjustments, can produce the correct, definit location of the courtesy amount. If there were many, many lines of text found, then the true courtesy amount is only one of the many possible candidates to be sent in to *Winbank*; however, this was not the case. For most checks, less than five lines of text were found to be appropriate to be sent through Winbank. The other candidates were usually the text lines of the account number that is written in *MICR* ink at the bottom of the check, the check number, or parts of the background image that had managed to pass all of the tests. Improvements can be made to better filter out these recurring situations.

For the foreign checks, especially the English ones, the courtesy amount box contains the darkest and thickest line as its edges. So, sometimes, not all of the lines could be removed, and if the writing was touching the box, then the entire courtesy amount region would be considered as a blob, with the dimensions of the courtesy amount box, meaning, that this blob will not pass the aspect ratio test and will therefore be thrown out. This was the major source of error on the foreign checks.

The resuls for both the U.S. and foreign checks look very promising. With some improvements and fine-tuning, the accuracy of the system can be raised even more. This system is the result of trying out many ideas before coming up with ones what worked.

The first attempt to implement a locater was similar to the current locater, with the addition of some preliminary steps before performing the Hough transform: After reading in the image data, the check is smoothed. Since the check is scanned in at such a high resolution, there is many little details on the check that would clutter up the initial computations. So, it is necessary to smooth out the image, which has the same effect as if the check were scanned in at a lower resolution, thus producing an output containing only the major details on the check. The smoothing filter used is a 5×5 two dimensional Gaussian array. The smoothed image is then filtered to look for abrupt changes in the horizontal and vertical directions.

Due to the filtering, the filtered image could contain values of such variable ranges that make it difficult to work with. So, the values of the filtered image is then normalized to bring it back, to range from 0 to 255. This normalization is done first finding the most minimum and maximum values found in the image. These two values are subtracted and divided by 255 to find the normalizing factor. Then each of the pixel values are scaled by the normalizing factor. This normalized image is then adaptively thresholded to bring out the lines of the courtesy amount box. The thresholded image is then fed in to the Hough transform.

This first system, which was based more focused on just trying to find the courtesy amount, was not used mainly for the reason that trying to find the courtesy box or line is much more difficult than trying to find other lines due to the color variations. This made adaptive thresholding very difficult and sometimes impossible. Thresholding to find the other lines on the check, which were consistently darker, was more accurate and less computationally intensive. Since the thresholding in the final system was very straightforward, filtering was not necessary, because the dark lines were already there and accentuated due to their darkness. Filtering in the first attempt was to bring out the lighter colored courtesy amount box. Also, smoothing was not necessary in the final system, because the thresholding got rid of most of the background picture anyway.

In the first system, looking for vertical lines was almost as important as finding horizontal lines, but it did not prove to be very useful when using Hough transforms.

Since thresholding on dark pixels was done first on the image, courtesy amount boxes, which tend to be light in color are thresholded away. Also, because Hough transforms are based on looking at the "popularity" of a slope and y-intercept, the vertical lines of the courtesy amount box, if they existed, were too short to make their slope and y-intercept show up as a prominent peak. All of these reasons made this attempt not as successful as the final system.

4.9 Possible Improvements

There are numerous improvements that could be made to the current locater problem. One of the obvious improvements is to fine tune the constants for cutoff's, such as the ones for the height of the text lines, number of dark pixels in a blob, the horizontal distance to distinguish lines of text, the volume/perimeter ratio, and the aspect ratio. Another improvement would be to actually integrate the locater system with *Winbank*.

There are also some possible improvements that would take a little more time. One is to look for a courtesy amount box on foreign checks. Foreign checks tend to have very dark and heavy boxes around the courtesy amount. So, to make the system run faster for foreign checks, it may be good to perform a Hough transform on vertical lines after finding the horizontal lines. Finding vertical lines is possible in this case, because the boxes are indeed dark and will not be thresholded away. Also, since the slopes of the horizontal lines are known, the slopes of the the vertical lines would just be the reciprocal of the horizontal slopes, because the vertical lines will obviously lie perpendicular to the horizontal lines. Therefore, only a one dimensional histogram will be needed for the Hough transform. If finding the vertical lines from the entire check is not successful, a Hough transform can be performed on a small window of the check at a time. One downside of doing this is that it is very computationally intensive.

Since the outcome of the current locater system often times includes the text line for the account number that is written at the bottom of the check in *MICR* ink and/or the check number, the recognition rate of these account numbers would be very high.

So, in order to avoid this confusion, another neural net could be added on to *Winbank* to recognize the account numbers and check numbers from the other text lines. This is not a hard task, because these numbers ar pretty standard and regular.

Because the system currently thresholds only on darker pixels, checks with very light colored lines or checks that do not contain any lines but rather shaded regions for the blanks are very not successfully processed. One possible solution is to implement parts of the first system attempted, described in the earlier section, to detect edges. So, if there are no lines detected on the check, it would go to a second stage where edge detection is used and then the image with the accentuated edges and lines would then be fed back in to find lines.

Another enhancement would be to keep a list of checks' formats that have been encountered. This is a middle point between running in "dummy" mode and trying to blindly find the courtesy amount. Initially, a list of known formats can be entered in to the system. Then as new checks are encountered, they can be added to the list of known formats. Finally, as an addition to keeping a list of known formats would be a culling system to initially sort out the checks based on size.

Chapter 5

Discussion

5.1 General Improvements

Even with the discussed improvements to segmentation, preprocessing, and locating the courtesy amount, there are some general improvements that can be made to the entire system. One of the biggest improvement that can help in both the location of the courtesy amount and the accuracy of processing the courtesy amount is to use the legal (word) amount as a double check. Performing recognition on the legal amount can be done, as a parallel process to recognizing the courtesy amount, using one of the numerous systems already available to recognize handwritten words. The recognition of the legal amount is really not as difficult as if it were only word recognition, because the dictionary of words that comprise the legal amount is very limited. Furthermore, the words are not extremely similar. So, a very crude recognition of the legal amount can help make sure that the result of the recognition of the courtesy amount is correct. It can also help determine if the locater found the courtesy amount on the check correctly. Of course, the problem is that the legal amount must be found as well. The legal amount might be distinguished by the fact that it is written on one of the longer lines on the check, and it is most probably below the line of the name of the recipient. Even with the problem of finding the legal amount, it is possible for the courtesy amount locater and the legal amount locater to work together to ultimately locate and recognize both of the amounts.

Chapter 6

Conclusion

A bank check processing system that can automatically process checks accurately can help save banks millions of dollars a year. The improvements and ideas presented here are all steps to achieve a system that will be faster, better, and more accurate than a human operator. It may take many years before this is possible. By then, the use of paper checks may even be completely obsolete. In the mean time, however, human operators are still keying in the transactions. So, the opportunity for a bank check processing system, perhaps even an imperfect one, to make a debut is still there.

References

[1] R. G. Casey and D. R. Ferguson. Intelligent forms processing. *IBM Systems Journal*, 29(3):435–450, 1990.

[2] A. Downtown and C. Leedham. Preprocessing and presorting of envelope images for automatic sorting using ocr. *Pattern Recognition*, 23((3/4)):347–362, 1990.

[3] Jonathan J. Hull Edward Cohen and Sargur N. Srihari. Understanding handwritten text in a structured environment: determining zip codes from addresses. *International Journal of Pattern Recognition and Artificial Intelligence*, 5((1/2)):221–264, 1991.

[4] Lloyd Alan Fletcher and Rangachar Kasturi. A robust algorithm for text string separation from mixed text/graphics images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(6), November 1988.

[5] Len M. Granowetter. Locating courtesy amount blocks on handwritten bank checks. Bachelor's thesis, Massachusetts Institute of Technology, 1993.

[6] Anil K. Jain and Sushil K. Battacharjee. Addrss block location on envelopes using gabor filters. *Pattern Recognition*, 25(12):1459–1477, 1992.

[7] Chan Pyng Lai and Rangachar Kasturi. Detection of dimension sets in engineering drawings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(8), August 1994.

[8] P. Palumbo S. Srihari, C. Wang and J. Hull. Recognizing address blocks on mail pieces. *The AI Magazine*, 8:25–40, December 1987.

[9] simonpe. http://vision.dai.ed.ac.uk/simonpe/hipr/html/hough.htm. World Wide Web page.

[10] Peter J. Sparks. A hybrid method for segmenting numeric character strings. Bachelor's thesis, Massachusetts Institute of Technology, 1991.

[11] S. Srihari. *Feature extraction for location address block on mail pieces*, pages 261–275. banana, 1987.

[12] C.C. Tapper. Speed, accuracy, and flexibility trade-offs in on-line character recognition. In P.S.P. Wang, editor, *Character and Handwriting Recognition*, pages 79–96. World Scientific, 1991.

[13] C.Y. Suen Tappert, C.C. and T. Wakahara. On-line handwriting recognition- a survey. In *Proc. Int. Conf. on Pattern Recognition*, pages 1123–1132. orange you glad, 1988.

[14] Dacheng Wang and Sargur N. Srihari. Analysis of form images. *International Journal of Pattern Recognition and Artificial Intelligence*, 8(5):1031–1052, 1994.

Appendix A

Locater Source Code Listings

```
#include <stdlib.h>

void *mmalloc(size_t size)
{
  void *ptr;

  if ((ptr=malloc(size)) == NULL) {
    printf("Bad Malloc\n");
    exit(0);
  }
  return(ptr);
}


void *ccalloc(size_t nelem, size_t elsize)
{
  void *ptr;

  if ((ptr=calloc(nelem,elsize)) == NULL) {
    printf("Bad Calloc\n");
    exit(0);
  }
  return(ptr);
}


void *rrealloc(void *ptr, size_t size)
{
  void *ptr2;

  if ((ptr2=realloc(ptr, size)) == NULL) {
    printf("Bad Realloc\n");
    exit(0);
  }
  return(ptr2);
}

void ffree(void *ptr)
```

```
{
 if (ptr == NULL)                                                    40
   printf("Bad Free\n");
 else free(ptr);
}
```

```
#include "image.h"

void find_ink(int **image, int **thresheld_image, int threshold_range,
              int xdim, int ydim, tSORT *lines, int num_lines, int *starts,
              int *ends, int *main_color)
{
 int count, count1, count2;
 int y_start, y_end;
 int *histogram, color;
 int max_pixels, max_color;                                          10

 float slope;

 histogram = (int *)mmalloc((size_t)threshold_range * sizeof(int));
 for (count=0; count<num_lines; count++) {
   for (count1=starts[count]; count1<ends[count]; count1++) {
     slope = (float)(lines[count].slope-(HIST_YSIZE/2))/(float)SLOPE_DIVIDE;
     y_start = (int)((slope*(float)count1) + (float)lines[count].intercept);
     for (count2=y_start; count2<(y_start+LOOK_ABOVE); count2++) {
         if (thresheld_image[count2][count1] < BLACK) {              20
           color = image[count2][count1];
           histogram[color]++;
         }
     }
   }
 }
 max_pixels = 0;
 for (count=0; count<threshold_range; count++)
   if (histogram[count] > max_pixels) {
     max_pixels = histogram[count];                                  30
     max_color = count;
   }
 *main_color = max_color;
}
```

```
#include "image.h"


void get_line(int **original, int **image, int xdim, tSORT line,
              int threshold, int *start, int *end, float *average_ratio)
{
 int count, count1, count2;
 int **output;
 int line_point;
```

```
float slope;
int temp_start, temp_end;
int max_start, max_end;
int length, max_length = 0;
int gap_detector;
int thickness_start, thickness;
int fill_in_count;
int start_looking;
int volume=0;
int perimeter=0;

temp_start = 0;
temp_end = 0;
for (count1=0; count1<xdim; count1++) {
  slope = (float)(line.slope-(HIST_YSIZE/2))/(float)SLOPE_DIVIDE;
  line_point = (int)((slope*(float)count1) + (float)line.intercept);
  gap_detector = 0;
  thickness_start = line_point+(3*LINE_EPSILON);
  for (count2=(line_point-LINE_EPSILON);
        count2<=(line_point+LINE_EPSILON); count2++) {
    if (image[count2][count1] < threshold) /* is dark/black pixel */
        temp_end = count1;
    else
        gap_detector++;
    if (gap_detector > (2*LINE_EPSILON)) {
        if ((temp_end-temp_start) > 0) {
          length = find_length(line, temp_start, temp_end);
          if (length > max_length) {
            max_length = length;
            max_start = temp_start;
            max_end = temp_end;
          }
        }
        temp_start = count1;
    }
  }
}

*start = max_start;
*end = max_end;

for (count1=max_start; count1<max_end; count1++) {
  slope = (float)(line.slope-(HIST_YSIZE/2))/(float)SLOPE_DIVIDE;
  line_point = (int)((slope*(float)count1) + (float)line.intercept);
  thickness = 0;
  thickness_start = line_point+(3*LINE_EPSILON);
  for (count2=(line_point-LINE_EPSILON);
        count2<=(line_point+LINE_EPSILON); count2++) {
    if (image[count2][count1] < threshold) /* is dark/black pixel */ {
        volume++;
        thickness++;
        if (count2<thickness_start)
          thickness_start = count2;
        image[count2][count1] = 255;
```

```
            original[count2][count1] = 255;
        }
      }
      if (thickness > 0) {
        perimeter += 2;
        start_looking = thickness_start + thickness;
        if (!((image[start_looking+1][count1]) &&                              70
              (image[start_looking+2][count1])) ||
            !((image[thickness_start-1][count1]) &&
              (image[thickness_start-2][count1]))) {
            for (count=thickness_start; count<start_looking; count++) {
              image[count][count1] = 0;
              original[count][count1] = 0;
            }
        }
      }
    }                                                                          80
    *average_ratio = (float)volume/(float)perimeter;
}


int find_length(tSORT line, int start, int end)
{
    int y_start, y_end;
    int output;
    float slope;
                                                                               90
    slope = (float)(line.slope-(HIST_YSIZE/2))/(float)SLOPE_DIVIDE;
    y_start = (int)((slope*(float)start) + (float)line.intercept);
    y_end = (int)((slope*(float)end) + (float)line.intercept);
    output = (int)sqrt((double)(((y_end-y_start)*(y_end-y_start)) +
                                ((end-start)*(end-start))));
    return output;
}
```

```
#include "image.h"
#include "utils.h"


tBITMAP convert_image(int **image, int xdim, int ydim, int threshold)
{
    int count1, count2;
    tBITMAP output;

    output = BitmapCreate(xdim, ydim);                                         10

    for (count1=0; count1<ydim; count1++)
      for (count2=0; count2<xdim; count2++) {
        if (image[count1][count2] < threshold)
            output[count1][count2] = (char)1;
        else
            output[count1][count2] = (char)0;
      }
```

```
  return output;
}

void FindBlobInfo(tBITMAP bitmap, int xdim, int ydim, int *row_start,
                  int *row_end, int *col_start, int *col_end, int *num_pixels)
{
  int count1, count2;

  *num_pixels = 0;
  *row_start = ydim;
  *row_end = 0;
  *col_start = xdim;
  *col_end = 0;
  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++) {
      if (bitmap[count1][count2]) {
          (*num_pixels)++;
          if (count2<*col_start)
            *col_start = count2;
          if (count2>*col_end)
            *col_end = count2;
          if (count1<*row_start)
            *row_start = count1;
          if (count1>*row_end)
            *row_end = count1;
      }
    }
}

tNEW_BLOBTYPE *GetBlobs(tBITMAP bitmap, int xdim, int ydim, float vol_per,
                        int *num_blobs, int **col_starts, int **col_ends,
                        int **row_starts, int **row_ends)
{
  int count1,count2;
  tBITMAP copy;
  tBITMAP temp;
  char filename[100];
  tNEW_BLOBTYPE *output;
  int num_alloc=200;
  int row_start, row_end, col_start, col_end, num_pixels;
  int counter = 0;
  float temp_per_vol;

  *num_blobs = 0;
  copy = BitmapCreate(xdim, ydim);

  for(count1=0;count1<xdim;count1++) {
    for(count2=0;count2<ydim;count2++) {
      copy[count2][count1] = bitmap[count2][count1];
    }
  }
```

```
  output = (tNEW_BLOBTYPE *)mmalloc((size_t)num_alloc * sizeof(tNEW_BLOBTYPE));
  (*col_starts) = (int *)ccalloc(num_alloc, sizeof(int));
  (*col_ends) = (int *)ccalloc(num_alloc, sizeof(int));
  (*row_starts) = (int *)ccalloc(num_alloc, sizeof(int));
  (*row_ends) = (int *)ccalloc(num_alloc, sizeof(int));
  while(1) {
    temp=NULL;
    GetBlob(copy, &temp, -1, -1, xdim, ydim, 0);                                    80
    if( temp==NULL ) break;
    FindBlobInfo(temp, xdim, ydim, &row_start, &row_end, &col_start, &col_end,
                 &num_pixels);
    if (num_pixels > MIN_BLOB_THRESHOLD && num_pixels < MAX_BLOB_THRESHOLD) {
      if (((float)(col_end-col_start)/(float)(row_end-row_start)) <
          BLOB_ASPECT_CUTOFF) {
        temp_per_vol = perimeter_volume(temp, xdim, ydim);
        if (temp_per_vol < PERIMETER_VOLUME) {
          output[(*num_blobs)] = ConvertBlobs(temp, xdim, ydim);
          (*col_starts)[(*num_blobs)] = col_start;                                  90
          (*col_ends)[(*num_blobs)] = col_end;
          (*row_starts)[(*num_blobs)] = row_start;
          (*row_ends)[(*num_blobs)] = row_end;
          (*num_blobs)++;
        }
      }
    }
    counter++;
    if (*num_blobs == num_alloc) {
      output = rrealloc(output, (num_alloc+5)*sizeof(tNEW_BLOBTYPE));              100
      (*col_starts) = rrealloc((*col_starts), (num_alloc+5)*sizeof(int));
      (*col_ends) = rrealloc((*col_ends), (num_alloc+5)*sizeof(int));
      (*row_starts) = rrealloc((*row_starts), (num_alloc+5)*sizeof(int));
      (*row_ends) = rrealloc((*row_ends), (num_alloc+5)*sizeof(int));
    }
    num_alloc += 5;
    BitmapDestroy(temp, ydim);
  }
  BitmapDestroy(copy, ydim);
  return output;                                                                    110
}



void GetBlob(tBITMAP input, tBITMAP *output, int start_row, int start_col,
             int xdim, int ydim, int recurse_count)
{
  int row = start_row;
  int col = start_col;
  int rcount;                                                                       120
  int ccount;

  if(recurse_count>50000) return;

  if( (start_row==-1) && (start_col==-1) ) {   /* If we just started */
    /* No start point, so brute force find any dark pixel. */
```

```
  /* The simplest robust way to do this is simply starting from one */
  /* corner of the bitmap and scanning. */

  for(col=0;col<xdim;col++) {                                          130
    for(row=0;row<ydim;row++) {
      if( input[row][col] ) break;
    }
    if( row!=ydim ) break;
  }
 }

 if( col==xdim && row==ydim ) {  /* We found no dark pixels */
   return;
 }                                                                     140

 /* If output is null, we need to malloc up the space to put our */
 /* bitmap. */
 if((*output) == NULL) (*output)=BitmapCreate(xdim, ydim);


 /* Look in a TOLERANCE neighborhood of row, col for dark pixels. */
 /* Recursively call self, at each dark pixel on the edge of that */
 /* neightborhood. */
                                                                       150
 for(ccount=col-TOLERANCE;ccount<=col+TOLERANCE;ccount++) {
   for(rcount=row-TOLERANCE;rcount<=row+TOLERANCE;rcount++) {
     if(ccount>=0 && ccount<xdim && rcount>=0 && rcount<ydim) {
       if( input[rcount][ccount] ) {
         input[rcount][ccount] = (char)0;
         (*output)[rcount][ccount] = (char)1;
         GetBlob(input, output, rcount, ccount, xdim, ydim, recurse_count+1);
       }
     }
   }                                                                   160
 }
}


tBITMAP BitmapCreate(int xdim, int ydim)
{
 int count;
 tBITMAP temp;
 temp = (tBITMAP)ccalloc(ydim, sizeof(tBITMAP_ROW));
 for(count=0;count<ydim;count++) {                                     170
   temp[count] = (tBITMAP_ROW)ccalloc(xdim, sizeof(char));
 }
 return temp;
}

void BitmapDestroy(tBITMAP bitmap, int ydim)
{
 int count;
 for (count=0; count<ydim; count++)
   ffree(bitmap[count]);                                               180
```

```
  ffree(bitmap);
}


tNEW_BLOBTYPE NewBlobCreate(int ydim)
{
  return (tNEW_BLOBTYPE)ccalloc((size_t)ydim, sizeof(struct tBOUNDARY));
}

                                                                        190
tNEW_BLOBTYPE ConvertBlobs (tBITMAP blob, int xdim, int ydim)
{
  tNEW_BLOBTYPE temp_blob;
  int count1, count2;
  int min, max;

  temp_blob = NewBlobCreate(ydim);
  for (count1=0; count1<ydim; count1++) {
    min = xdim;
    max = 0;                                                            200
    for (count2=0; count2<xdim; count2++) {
      if (blob[count1][count2]) {
        if (count2<min)
          min=count2;
        if (count2>max)
          max=count2;
      }
    }
    temp_blob[count1].start_blob = min;
    temp_blob[count1].end_blob = max;                                   210
  }
  return temp_blob;
}

tBITMAP NewBlob2OldBlob(tBITMAP bitmap, tNEW_BLOBTYPE input, int xdim,
                        int ydim)
{
  tBITMAP temp_blob;
  int count1, count2;
                                                                        220
  temp_blob = BitmapCreate(xdim, ydim);
  for (count1=0; count1<ydim; count1++)
    for (count2=input[count1].start_blob; count2<=input[count1].end_blob;
         count2++)
      if (bitmap[count1][count2])
        temp_blob[count1][count2] = 1;
  return temp_blob;
}

                                                                        230

int fwrite_bitmap(char *filename, tBITMAP bitmap, int height, int width)
{
```

```
  int count1, count2, count3;
  FILE *outfile;
  unsigned char outval;
  unsigned char bitmap_mask[] = {0x01, 0x02, 0x04, 0x08,
                                 0x10, 0x20, 0x40, 0x80};
                                                                          240
  int div_eight;

  if( (outfile = fopen(filename, "w")) == NULL) {
    printf("fwrite_bitmap:    Couldn't open file %s for writing\n", filename);
    exit(0);
  }

  fprintf(outfile, "#define %s_width %d\n", filename, width);
  fprintf(outfile, "#define %s_height %d\n", filename, height);
  fprintf(outfile, "static unsigned char %s_bits[] = {\n", filename);     250

  div_eight = (int)(ceil((double)((double)width/(double)8)));
  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<div_eight;count2++) {
      outval = 0;
      for(count3=0;count3<8;count3++) {
        if( (count2*8+count3) < width) {
          if ( bitmap[count1][count2*8+count3] ) {
            outval += bitmap_mask[count3];
          }                                                               260
        }
      }
      if(count1==(height-1) && count2==(div_eight-1)) {
        fprintf(outfile, "%s};\n", uchar2hexstring(outval));
      }
      else {
        fprintf(outfile, "%s,\n", uchar2hexstring(outval));
      }
    }
  }                                                                       270
  fclose(outfile);
}


char *uchar2hexstring(unsigned char bits) {
  switch (bits)
    {
    case 0:
      return "0x00";
    case 1:                                                               280
      return "0x01";
    case 2:
      return "0x02";
    case 3:
      return "0x03";
    case 4:
      return "0x04";
    case 5:
```

```
  return "0x05";
case 6:                                                    290
  return "0x06";
case 7:
  return "0x07";
case 8:
  return "0x08";
case 9:
  return "0x09";
case 10:
  return "0x0a";
case 11:                                                   300
  return "0x0b";
case 12:
  return "0x0c";
case 13:
  return "0x0d";
case 14:
  return "0x0e";
case 15:
  return "0x0f";
case 16:                                                   310
  return "0x10";
case 17:
  return "0x11";
case 18:
  return "0x12";
case 19:
  return "0x13";
case 20:
  return "0x14";
case 21:                                                   320
  return "0x15";
case 22:
  return "0x16";
case 23:
  return "0x17";
case 24:
  return "0x18";
case 25:
  return "0x19";
case 26:                                                   330
  return "0x1a";
case 27:
  return "0x1b";
case 28:
  return "0x1c";
case 29:
  return "0x1d";
case 30:
  return "0x1e";
case 31:                                                   340
  return "0x1f";
case 32:
```

```
  return "0x20";
case 33:
  return "0x21";
case 34:
  return "0x22";
case 35:
  return "0x23";
case 36:
  return "0x24";
case 37:
  return "0x25";
case 38:
  return "0x26";
case 39:
  return "0x27";
case 40:
  return "0x28";
case 41:
  return "0x29";
case 42:
  return "0x2a";
case 43:
  return "0x2b";
case 44:
  return "0x2c";
case 45:
  return "0x2d";
case 46:
  return "0x2e";
case 47:
  return "0x2f";
case 48:
  return "0x30";
case 49:
  return "0x31";
case 50:
  return "0x32";
case 51:
  return "0x33";
case 52:
  return "0x34";
case 53:
  return "0x35";
case 54:
  return "0x36";
case 55:
  return "0x37";
case 56:
  return "0x38";
case 57:
  return "0x39";
case 58:
  return "0x3a";
case 59:
```

```
  return "0x3b";
case 60:
  return "0x3c";
case 61:
  return "0x3d";
case 62:
  return "0x3e";
case 63:
  return "0x3f";
case 64:
  return "0x40";
case 65:
  return "0x41";
case 66:
  return "0x42";
case 67:
  return "0x43";
case 68:
  return "0x44";
case 69:
  return "0x45";
case 70:
  return "0x46";
case 71:
  return "0x47";
case 72:
  return "0x48";
case 73:
  return "0x49";
case 74:
  return "0x4a";
case 75:
  return "0x4b";
case 76:
  return "0x4c";
case 77:
  return "0x4d";
case 78:
  return "0x4e";
case 79:
  return "0x4f";
case 80:
  return "0x50";
case 81:
  return "0x51";
case 82:
  return "0x52";
case 83:
  return "0x53";
case 84:
  return "0x54";
case 85:
  return "0x55";
case 86:
```

```
  return "0x56";
 case 87:
  return "0x57";
 case 88:
  return "0x58";
 case 89:
  return "0x59";
 case 90:
  return "0x5a";
 case 91:
  return "0x5b";
 case 92:
  return "0x5c";
 case 93:
  return "0x5d";
 case 94:
  return "0x5e";
 case 95:
  return "0x5f";
 case 96:
  return "0x60";
 case 97:
  return "0x61";
 case 98:
  return "0x62";
 case 99:
  return "0x63";
 case 100:
  return "0x64";
 case 101:
  return "0x65";
 case 102:
  return "0x66";
 case 103:
  return "0x67";
 case 104:
  return "0x68";
 case 105:
  return "0x69";
 case 106:
  return "0x6a";
 case 107:
  return "0x6b";
 case 108:
  return "0x6c";
 case 109:
  return "0x6d";
 case 110:
  return "0x6e";
 case 111:
  return "0x6f";
 case 112:
  return "0x70";
 case 113:
```

```
  return "0x71";
 case 114:
  return "0x72";
 case 115:
  return "0x73";
 case 116:
  return "0x74";
 case 117:
  return "0x75";
 case 118:
  return "0x76";
 case 119:
  return "0x77";
 case 120:
  return "0x78";
 case 121:
  return "0x79";
 case 122:
  return "0x7a";
 case 123:
  return "0x7b";
 case 124:
  return "0x7c";
 case 125:
  return "0x7d";
 case 126:
  return "0x7e";
 case 127:
  return "0x7f";
 case 128:
  return "0x80";
 case 129:
  return "0x81";
 case 130:
  return "0x82";
 case 131:
  return "0x83";
 case 132:
  return "0x84";
 case 133:
  return "0x85";
 case 134:
  return "0x86";
 case 135:
  return "0x87";
 case 136:
  return "0x88";
 case 137:
  return "0x89";
 case 138:
  return "0x8a";
 case 139:
  return "0x8b";
 case 140:
```

```
    return "0x8c";
  case 141:
    return "0x8d";
  case 142:
    return "0x8e";
  case 143:
    return "0x8f";
  case 144:
    return "0x90";
  case 145:
    return "0x91";
  case 146:
    return "0x92";
  case 147:
    return "0x93";
  case 148:
    return "0x94";
  case 149:
    return "0x95";
  case 150:
    return "0x96";
  case 151:
    return "0x97";
  case 152:
    return "0x98";
  case 153:
    return "0x99";
  case 154:
    return "0x9a";
  case 155:
    return "0x9b";
  case 156:
    return "0x9c";
  case 157:
    return "0x9d";
  case 158:
    return "0x9e";
  case 159:
    return "0x9f";
  case 160:
    return "0xa0";
  case 161:
    return "0xa1";
  case 162:
    return "0xa2";
  case 163:
    return "0xa3";
  case 164:
    return "0xa4";
  case 165:
    return "0xa5";
  case 166:
    return "0xa6";
  case 167:
```

```
    return "0xa7";
  case 168:
    return "0xa8";
  case 169:
    return "0xa9";
  case 170:
    return "0xaa";
  case 171:
    return "0xab";
  case 172:
    return "0xac";
  case 173:
    return "0xad";
  case 174:
    return "0xae";
  case 175:
    return "0xaf";
  case 176:
    return "0xb0";
  case 177:
    return "0xb1";
  case 178:
    return "0xb2";
  case 179:
    return "0xb3";
  case 180:
    return "0xb4";
  case 181:
    return "0xb5";
  case 182:
    return "0xb6";
  case 183:
    return "0xb7";
  case 184:
    return "0xb8";
  case 185:
    return "0xb9";
  case 186:
    return "0xba";
  case 187:
    return "0xbb";
  case 188:
    return "0xbc";
  case 189:
    return "0xbd";
  case 190:
    return "0xbe";
  case 191:
    return "0xbf";
  case 192:
    return "0xc0";
  case 193:
    return "0xc1";
  case 194:
```

```
    return "0xc2";
  case 195:
    return "0xc3";
  case 196:
    return "0xc4";
  case 197:
    return "0xc5";
  case 198:
    return "0xc6";
  case 199:
    return "0xc7";
  case 200:
    return "0xc8";
  case 201:
    return "0xc9";
  case 202:
    return "0xca";
  case 203:
    return "0xcb";
  case 204:
    return "0xcc";
  case 205:
    return "0xcd";
  case 206:
    return "0xce";
  case 207:
    return "0xcf";
  case 208:
    return "0xd0";
  case 209:
    return "0xd1";
  case 210:
    return "0xd2";
  case 211:
    return "0xd3";
  case 212:
    return "0xd4";
  case 213:
    return "0xd5";
  case 214:
    return "0xd6";
  case 215:
    return "0xd7";
  case 216:
    return "0xd8";
  case 217:
    return "0xd9";
  case 218:
    return "0xda";
  case 219:
    return "0xdb";
  case 220:
    return "0xdc";
  case 221:
```

```
  return "0xdd";
 case 222:
  return "0xde";
 case 223:
  return "0xdf";
 case 224:
  return "0xe0";
 case 225:
  return "0xe1";
 case 226:
  return "0xe2";
 case 227:
  return "0xe3";
 case 228:
  return "0xe4";
 case 229:
  return "0xe5";
 case 230:
  return "0xe6";
 case 231:
  return "0xe7";
 case 232:
  return "0xe8";
 case 233:
  return "0xe9";
 case 234:
  return "0xea";
 case 235:
  return "0xeb";
 case 236:
  return "0xec";
 case 237:
  return "0xed";
 case 238:
  return "0xee";
 case 239:
  return "0xef";
 case 240:
  return "0xf0";
 case 241:
  return "0xf1";
 case 242:
  return "0xf2";
 case 243:
  return "0xf3";
 case 244:
  return "0xf4";
 case 245:
  return "0xf5";
 case 246:
  return "0xf6";
 case 247:
  return "0xf7";
 case 248:
```

```
      return "0xf8";
    case 249:
      return "0xf9";
    case 250:
      return "0xfa";
    case 251:
      return "0xfb";
    case 252:
      return "0xfc";
    case 253:
      return "0xfd";
    case 254:
      return "0xfe";
    case 255:
      return "0xff";
    }
}


/*
void clean_image(int **image, int xdim, int ydim)
{
  int count1, count2;
  tBITMAP bitmap;

  bitmap = convert_image(image, xdim, ydim);
  GetBlobs(&bitmap, xdim, ydim);
  image = unconvert_bitmap(bitmap, xdim, ydim);
}


int **unconvert_bitmap(tBITMAP bitmap, int xdim, int ydim)
{
  int count1, count2;
  int **output;

  output = (int **)ccalloc(ydim, sizeof(int *));
  for (count1=0; count1<ydim; count1++)
    output[count1] = (int *)ccalloc(xdim, sizeof(int));

  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++) {
      if (bitmap[count1][count2])
          output[count1][count2] = 0;
      else
          output[count1][count2] = 255;
    }
  return output;
}

int FindBlobWeight(tBITMAP bitmap, int xdim, int ydim)
{
  int count1, count2;
  int weight=0;
```

```
  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++) {
      if (bitmap[count1][count2])
          weight++;
    }
  return weight;
}

*/
```

```
#include "image.h"
#include "string.h"


void main(int argc, char *argv[])
{
  char *input_filename = "/var/tmp/pictures/default_input.pgm";
  char *output_filename = "/var/tmp/pictures/default_output.pgm";
  char input_filename_copy[1024];
  int input_filename_number;
  int **image_array;
  int xdim, ydim;
  int count, count1, count2, count3, count4;
  int **thresheld_image;
  int low_thresh_val, high_thresh_val;
  tSORT *lines;
  int num_lines;
  int *starts, *ends;
  float slope;
  int main_color;
  int **ink_image;
  tBITMAP bitmap;
  float *line_ratios;
  int num_blobs;
  tNEW_BLOBTYPE *blobs;
  tBITMAP temp_bitmap;
  float total_ratio_count = 0.0;
  float avg_line_ratio;
  char filename[100];
  tREGION *text_lines;
  int *col_starts, *col_ends, *row_starts, *row_ends;
  int orig_num_text_lines, num_text_lines;
  int *blob_association;
  tBITMAP *textline_bitmaps;
  int next_available_slot = 0;
  tREGION *current_ptr;

  if(argc>1) input_filename = argv[1];
  if(argc>2) output_filename = argv[2];

  low_thresh_val = 125;
  high_thresh_val = 255;
```

```
image_array = pgm2array(input_filename, &xdim, &ydim);

thresheld_image = threshold(image_array, xdim, ydim, low_thresh_val,
                            high_thresh_val);

lines = horizontal_hough(thresheld_image, xdim, ydim, &num_lines);

starts = (int *)ccalloc(num_lines, sizeof(int));
ends = (int *)ccalloc(num_lines, sizeof(int));
line_ratios = (float *)ccalloc(num_lines, sizeof(float));
for (count=0; count<num_lines; count++) {
  get_line(image_array, thresheld_image, xdim, lines[count], BLACK,
           &starts[count], &ends[count], &line_ratios[count]);
  slope = (float)(lines[count].slope-(HIST_YSIZE/2))/(float)1000;
  printf("bin count:  %d   ", lines[count].histogram_height);
  printf("slope:  %1.4f   y_intercept:  %d   start:  %d   end:  %d\n",
         slope, lines[count].intercept, starts[count], ends[count]);
}

find_ink(image_array, thresheld_image, (high_thresh_val-low_thresh_val),
         xdim, ydim, lines, num_lines, starts, ends, &main_color);
ink_image = ink_threshold(image_array, xdim, ydim, (main_color-INK_RANGE),
                          (main_color+INK_RANGE));

printf("starting blob work\n");
bitmap = convert_image(thresheld_image, xdim, ydim, BLACK);
blobs = GetBlobs(bitmap, xdim, ydim, avg_line_ratio, &num_blobs, &col_starts,
                 &col_ends, &row_starts, &row_ends);

printf("number of blobs:  %d\n", num_blobs);

blob_association = (int *)ccalloc(num_blobs, sizeof(int *));
text_lines = find_text_lines(col_starts, col_ends, row_starts, row_ends,
                             num_blobs, &num_text_lines, blob_association);

orig_num_text_lines = num_text_lines;
for (count1=0; count1<orig_num_text_lines; count1++) {
  printf("text line %d  height:  %d  row:  %d--%d  col:  %d--%d\n", count1,
         (text_lines[count1].stop_row - text_lines[count1].start_row),
         text_lines[count1].start_row, text_lines[count1].stop_row,
         text_lines[count1].start_col, text_lines[count1].stop_col);
  if (
      text_lines[count1].stop_row-text_lines[count1].start_row>MIN_HEIGHT
      && text_lines[count1].stop_row-text_lines[count1].start_row<MAX_HEIGHT
      && text_lines[count1].stop_col-text_lines[count1].start_col>MIN_LENGTH
      && text_lines[count1].stop_col-text_lines[count1].start_col<MAX_LENGTH
      ) {
    text_lines[next_available_slot] = text_lines[count1];
    for (count=0; count<num_blobs; count++)
      if (blob_association[count] == next_available_slot)
        blob_association[count] = -1;
    for (count=0; count<num_blobs; count++)
      if (blob_association[count] == count1)
```

```
            blob_association[count] = next_available_slot;
      next_available_slot++;
    }
    else (num_text_lines)--;
  }

  textline_bitmaps =(tBITMAP *)mmalloc((size_t)num_text_lines*sizeof(tBITMAP));
  for (count=0; count<num_text_lines; count++)
    textline_bitmaps[count] = BitmapCreate(xdim, ydim);
  printf("number of text lines found:  %d\n", num_text_lines);
  for (count1=0; count1<num_text_lines; count1++) {
    for (count2=0; count2<num_blobs; count2++) {
      if (blob_association[count2] == count1) {
          for (count3=row_starts[count2]; count3<row_ends[count2]; count3++) {
            for (count4=blobs[count2][count3].start_blob;
                 count4<=blobs[count2][count3].end_blob; count4++)
              (textline_bitmaps[count1])[count3][count4] =bitmap[count3][count4];
          }
      }
    }
    strcpy(input_filename_copy, input_filename);
    *(strstr(input_filename_copy, ".tif")) = '\0';
    input_filename_number = atoi(input_filename_copy);
    sprintf(filename, "output/%d-bl-%d", input_filename_number, count1);
    fwrite_bitmap(filename, textline_bitmaps[count1], ydim, xdim);
    printf("done with writing text line %d\n", count1);
  }


}
```

```
#include "image.h"
#include "utils.h"

tSORT *horizontal_hough(int **input, int xdim, int ydim, int *number_lines)
{
  int count1, count2, count3;
  int **histogram;
  float slope;
  int slope_bin, y_intercept;
  int num_lines;
  tSORT *lines;

  histogram = (int **)ccalloc(ydim, sizeof(int *));
  for (count1=0; count1<ydim; count1++)
    histogram[count1] = (int *)ccalloc(HIST_YSIZE, sizeof(int));

  for (count1=10; count1<(ydim-10); count1++)
    for (count2=10; count2<(xdim-10); count2++) {
      if (input[count1][count2] < 20) {
          for (count3=0; count3<HIST_YSIZE; count3++) {
            slope = (float)(count3-(HIST_YSIZE/2))/(float)SLOPE_DIVIDE;
```

```
            y_intercept = (int)((float)count1-(slope*(float)count2));
            if ((y_intercept > 0) && (y_intercept < ydim))
              histogram[y_intercept][count3]++;
          }
      }
    }

  lines = find_peaks(histogram, HIST_YSIZE, ydim, &num_lines);
  *number_lines = num_lines;
  return (lines);
}


int **rotate(int **input, int xdim, int ydim)
{
  int count1, count2;
  int **output;

  output = (int **)ccalloc(xdim, sizeof(int *));
  for (count1=0; count1<xdim; count1++)
    output[count1] = (int *)ccalloc(ydim, sizeof(int));
  for (count1=0; count1<xdim; count1++)
    for (count2=0; count2<ydim; count2++)
      output[count1][count2] = input[count2][count1];
  return (output);
}


tSORT *find_peaks(int **histogram, int size, int ydim, int *num_lines)
{
  int count1, count2, count;
  int max_peak=0, test_slope, test_intercept;
  int istat;
  int total_bin_count=0;
  int count_bins = 0;
  int average;
  int replace_index;
  int maxpeak=0;
  tSORT *sorting_array;
  int num_alloc=100;
  int num_items=0;
  int weighted_slope;
  int weighted_intercept;
  int slope_denom;
  int intercept_denom;
  float weighted_average;
  float histogram_cutoff;

  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<size; count2++)
      if (histogram[count1][count2] > 0) {
          total_bin_count += histogram[count1][count2];
          count_bins++;
          if (histogram[count1][count2] > maxpeak) {
```

```
            maxpeak = histogram[count1][count2];
            test_slope = count2;
            test_intercept = count1;
          }
      }

  average = total_bin_count/count_bins;

  printf("height in max peak:  %d\n", maxpeak);
  printf("total number of bins:  %d\n", count_bins);
  printf("max's slope:  %d\n", test_slope);
  printf("max's intercept:  %d\n", test_intercept);
  printf("average:  %d\n",average);

  sorting_array = (tSORT *)mmalloc((size_t)num_alloc * sizeof(tSORT));
  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<size; count2++)
      if (((float)histogram[count1][count2]/(float)maxpeak)>0.75) {
          if (num_items == num_alloc) {
            sorting_array = rrealloc(sorting_array,
                                     (num_alloc+100)*sizeof(tSORT));
            num_alloc+=100;
          }
          sorting_array[num_items].histogram_height = histogram[count1][count2];
          sorting_array[num_items].slope = count2;
          sorting_array[num_items].intercept = count1;
          num_items++;
      }

  printf("made sorted array\n");

#define DECREASING(a,b) ((a).histogram_height > (b).histogram_height) \
  HPSORT_ARRAY(sorting_array, num_items, DECREASING);
#undef DECREASING

  printf("number of items in sorting array:  %d\n", num_items);

  for (count=0; count<num_items; count++) {
    for (count1=(count+1); count1<num_items; count1++)
      if (sorting_array[count].intercept != -1) {
          if ((abs(sorting_array[count].intercept -
                   sorting_array[count1].intercept) <= PEAK_THRESH) &&
              (abs(sorting_array[count].slope -
                   sorting_array[count1].slope) <= PEAK_THRESH))
            sorting_array[count].intercept = -1;
      }
  }
  pack_array(sorting_array, &num_items);


  *num_lines = num_items;
  printf ("number of lines in find_peaks:  %d\n", num_items);
```

```
  return(sorting_array);
}

void pack_array(tSORT *sorting_array, int *num_items)
{
  int count;
  int constant_num;
  int next_available_slot=0;
  tSORT *current_ptr;

  constant_num = *num_items;
  for (count=0; count<constant_num; count++) {
    if (sorting_array[count].intercept != -1) {
      sorting_array[next_available_slot] = sorting_array[count];
      next_available_slot++;
    }
    else (*num_items)--;
  }
}


/* when calling this function, decrement item_number by 1 every time */
void delete_item(tSORT *sorting_array, int num_items, int item_number)
{
  int count;

  for (count=item_number; count<(num_items-1); count++)
    sorting_array[count] = sorting_array[count-1];
}

int element_exists(tSORT *sorting_array, int num_items, int slope,
                   int intercept)
{
  int count;

  for (count=0; count<num_items; count++)
     if ((sorting_array[count].slope = slope) &&
          (sorting_array[count].intercept = intercept))
        return(count+1);
  return(0);
}
```

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


/***** main.c *****/
#define FILTER_WIDTH 3
#define FILTER_LENGTH 5
#define FILTER2_WIDTH 5
#define FILTER2_LENGTH 3
```

```
#define SMOOTHING_WIDTH 5
#define SMOOTHING_LENGTH 5
#define INK_RANGE 20

/***** hough_transform.c *****/
#define HIST_YSIZE 200
#define PEAK_HALF_WIDTH 3
#define PEAK_THRESH 2
#define SLOPE_DIVIDE 1000
/*  Sorts (clobbering) an array, where                                    20
    if operation(array[i], array[j]) is true, then i precedes j
    This is from Numerical Recipes in C
    */

#define HPSORT_ARRAY(array, length, operation) \
{ \
   while(1) { \
     typeof(&(*array)) hpsort_internal_ptr; \
     typeof(*array) hpsort_internal_item; \
     unsigned long hpsort_internal_i, hpsort_internal_ir; \              30
     unsigned long hpsort_internal_j, hpsort_internal_l; \
     hpsort_internal_ptr = (array) - 1; \
     if ((length) < 2) break;  \
     hpsort_internal_l=((length) >> 1)+1; \
     hpsort_internal_ir=(length); \
     for (;;) { \
       if (hpsort_internal_l > 1) { \
         hpsort_internal_item=hpsort_internal_ptr[--hpsort_internal_l]; \
       } else { \
         hpsort_internal_item=hpsort_internal_ptr[hpsort_internal_ir]; \  40
         hpsort_internal_ptr[hpsort_internal_ir]=hpsort_internal_ptr[1]; \
         if (--hpsort_internal_ir == 1) { \
           hpsort_internal_ptr[1]=hpsort_internal_item; \
           break; \
         } \
       } \
       hpsort_internal_i=hpsort_internal_l; \
       hpsort_internal_j=hpsort_internal_l+hpsort_internal_l; \
       while (hpsort_internal_j <= hpsort_internal_ir) { \
         if (hpsort_internal_j < hpsort_internal_ir && \                  50
                  hpsort_internal_ptr[hpsort_internal_j+1])) \
           hpsort_internal_j++; \
           if (operation(hpsort_internal_item, \
                    hpsort_internal_ptr[hpsort_internal_j])) { \
             hpsort_internal_ptr[hpsort_internal_i] = \
               hpsort_internal_ptr[hpsort_internal_j]; \
               hpsort_internal_i=hpsort_internal_j; \
               hpsort_internal_j <<= 1; \
           } else hpsort_internal_j=hpsort_internal_ir+1; \
       } \                                                               60
       hpsort_internal_ptr[hpsort_internal_i]=hpsort_internal_item; \
     } \
     break; \
   } \
```

```
}


/***** hough_test.c *****/
#define HEIGHT 100
#define WIDTH 300

/***** get_line.c *****/
#define LINE_INK_THRESH 20  /* for finding black ink only */
#define LINE_EPSILON 2
#define BLACK 20 /* for finding only black on a thresholded image */
#define RECONSTRUCT_LINE_THRESH 3

/***** get_blob.c *****/
#define MIN_BLOB_THRESHOLD 75
#define MAX_BLOB_THRESHOLD 15000
#define TOLERANCE 4
#define BLOB_ASPECT_CUTOFF 3.5
#define PERIMETER_VOLUME 4.0


/***** find_ink.c *****/
#define LOOK_ABOVE 100
#define LINES_SEPARATION 25

/***** text_line.c *****/
#define MIN_HEIGHT 30
#define MAX_HEIGHT 300
#define MIN_LENGTH 60
#define MAX_LENGTH 600

#define MAX_HORIZ_DIST 200
#define MIN_FRAC_OVERLAP 0.5
#define MIN_OVERLAP 15

/*************************************************************************/
/***** hough_transform.c *****/
struct tLINE {
  int slope, intercept;
};
typedef struct tLINE tLINE;

struct tSORT {
  int histogram_height;
int slope, intercept;
};
typedef struct tSORT tSORT;

/***** get_blob.c *****/
typedef char *tBITMAP_ROW;
typedef tBITMAP_ROW *tBITMAP;
struct tBOUNDARY {
  int start_blob, end_blob;
};
```

```
typedef struct tBOUNDARY *tNEW_BLOBTYPE;

/***** find_ink.c *****/
struct tINK {
  int color, num_pixels;
};
typedef struct tINK tINK;


/***** text_line.c *****/
struct tREGION {
  int start_row, stop_row, start_col, stop_col;
};
typedef struct tREGION tREGION;

/*************************************************************************/
/***** pgmlib.c *****/
int array2pgm(unsigned char **carray, int xdim, int ydim, char *filename);
int **pgm2array(char *filename, int *xdim, int *ydim);

/***** filter.c *****/
int **filter(int **carray, int **filter_array, int xdim, int ydim, int width,
             int length);

/***** normalize.c *****/
int **normalize(int **input, int xdim, int ydim);
void find_extremes(int **input, int xdim, int ydim, int *min, int *max);

/***** generate_filter.c *****/
int **generate_filter();

/***** threshold.c *****/
int **threshold(int **input, int xdim, int ydim, int low_thresh_val,
                int high_thresh_val);
int **ink_threshold(int **input, int xdim, int ydim, int low_thresh_val,
                    int high_thresh_val);

/***** convert.c *****/
unsigned char **int2char(int **input, int xdim, int ydim);

/***** hough_transform.c *****/
tSORT *horizontal_hough(int **input, int xdim, int ydim, int *num_lines);
int ** rotate(int **input, int xdim, int ydim);
void vertical_hough(int **input, int xdim, int ydim);
int throw_out(int **histogram, tLINE *outlines);
tSORT *find_peaks(int **histogram, int size, int ydim, int *num_lines);
void delete_item(tSORT *sorting_array, int num_items, int item_number);
void pack_array(tSORT *sorting_array, int *num_items);

/***** hough_test.c *****/
void bitmap_to_int_array(unsigned char *xbitmap, int **image_array, int height,
                         int width);
```

```
/***** find_lines.c *****/
void find_lines(int **threshold_image, int xdim, int ydim, tSORT *lines,
                int num_lines, int threshold, int *starts, int *ends);


/***** get_blob.c *****/
int **unconvert_bitmap(tBITMAP bitmap, int xdim, int ydim);
tBITMAP convert_image(int **image, int xdim, int ydim, int threshold);
void FindBlobInfo(tBITMAP bitmap, int xdim, int ydim, int *row_start,
                  int *row_end, int *col_start, int *col_end, int *num_pixels);
tNEW_BLOBTYPE *GetBlobs(tBITMAP bitmap, int xdim, int ydim, float vol_per,
                        int *num_blobs, int **col_starts, int **col_ends,
                        int **row_start, int **row_ends);
void GetBlob(tBITMAP input, tBITMAP *output, int start_row, int start_col,
             int xdim, int ydim, int recurse_depth);
tBITMAP BitmapCreate(int xdim, int ydim);
void BitmapDestroy(tBITMAP bitmap, int ydim);
tNEW_BLOBTYPE NewBlobCreate(int ydim);
tNEW_BLOBTYPE ConvertBlobs(tBITMAP blob, int xdim, int ydim);
tBITMAP NewBlob2OldBlob(tBITMAP bitmap, tNEW_BLOBTYPE input,int xdim,int ydim);
int fwrite_bitmap(char *filename, tBITMAP bitmap, int height, int width);
char *uchar2hexstring(unsigned char bits);

/***** get_line.c *****/
void get_line(int **original, int **image, int xdim, tSORT line,
              int threshold, int *start, int *end, float *average_ratio);
int find_length(tSORT line, int start, int end);

/***** find_ink.c *****/
void find_ink(int **image, int **thresheid_image, int threshold_range,
              int xdim, int ydim, tSORT *lines, int num_lines, int *starts,
              int *ends, int *main_color);

/***** perimeter.c *****/
float perimeter_volume(tBITMAP bitmap, int xdim, int ydim);


/***** text_line.c *****/
tREGION *find_text_lines(int *col_starts, int *col_ends, int *row_starts,
                         int *row_ends, int num_blobs, int *num_text_lines,
                         int *blob_association);
tBITMAP make_region_bitmap(tBITMAP bitmap, int xdim, int ydim, tREGION region);
```

```
#include "image.h"


float perimeter_volume(tBITMAP bitmap, int xdim, int ydim)
{
  int count1, count2;
  int perimeter = 0;
  int volume = 0;
  float output;
```

```
  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++) {
      if (bitmap[count1][count2]) {
          volume++;
          if ((count1 == 0) || (count1 == (ydim-1)) ||
             (count2 == 0) || (count2 == (xdim-1)))
            perimeter++;
          else {
            if (!bitmap[count1-1][count2] || !bitmap[count1+1][count2] ||
                !bitmap[count1][count2-1] || !bitmap[count1][count2+1] ||
                !bitmap[count1-1][count2-1] || !bitmap[count1+1][count2+1] ||
                !bitmap[count1-1][count2+1] || !bitmap[count1+1][count2-1])
              perimeter++;
          }
      }
    }
  output = (float)volume/(float)perimeter;
  return output;
}
```

```
#include "image.h"


int array2pgm(unsigned char **carray, int xdim, int ydim, char *filename)
{
  FILE *fileout;
  int count1, count2;

  fileout = fopen(filename, "w");
  if (fileout  == NULL) {
    fprintf(stderr, "Couldn't open file\n");
    return(0);
  }
  fprintf(fileout, "P5\n#Dump Raw PGM\n#CREATOR: Patricia Liu\n");
  fprintf(fileout, "%d %d\n255\n", xdim, ydim);
  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++)
      if (fwrite(&(carray[count1][count2]), (size_t)1,
                 (size_t) 1, fileout)
          != 1) {
          /* or use fputc((int)carray[count2][count1], fileout) */
          fprintf(stderr,"Had trouble writing\n");
          return(0);
      }
  fclose(fileout);
  return(1);
}


int **pgm2array(char *filename, int *xdim, int *ydim)
{
  int **output;
  FILE *filein;
```

```
  int count1, count2;
  char *s;

  filein = fopen(filename, "r");
  if (filein == NULL) {
    fprintf(stderr, "Could not open file\n");
    return(0);
  }
  s = (char *)ccalloc(80, sizeof(char));
  fgets(s, 80, filein);
  while (cont(s)) {
    fgets(s, 80, filein);
  }
  sscanf(s, "%d %d", xdim, ydim);
  fgets(s, 80, filein);
  output = (int **)ccalloc(*ydim, sizeof(int *));
  for (count1=0; count1<*ydim; count1++)
    output[count1] = (int *)ccalloc(*xdim, sizeof(int));
  for (count1=0; count1<*ydim; count1++)
    for (count2=0; count2<*xdim; count2++)
      output[count1][count2] = (int)fgetc(filein);
  printf("Got this far\n");
  return(output);
}


int cont(char *s)
{
  int count=0;

  while (s[count] == ' ')
    count++;
  if ((s[count] == 'P') || (s[count] == '#'))
    return(1);
  else return(0);
}
```

```
#include "image.h"
#include "utils.h"


tREGION *find_text_lines(int *col_starts, int *col_ends, int *row_starts,
                         int *row_ends, int num_blobs, int *num_text_lines,
                         int *blob_association)
{
  int count, count1, count2;
  tREGION *output, temp_line;
  int num_alloc = 4;
  int num_items = 0;
  int temp_height;
  int one_blob;

  for (count=0; count<num_blobs; count++)
```

```
  blob_association[count] = -1;
output = (tREGION *)mmalloc((size_t)num_alloc * sizeof(tREGION));
for (count1=0; count1<num_blobs; count1++) {
  if (blob_association[count1] == -1){
    temp_line.start_row = row_starts[count1];
    temp_line.stop_row = row_ends[count1];
    temp_line.start_col = col_starts[count1];
    temp_line.stop_col = col_ends[count1];
    temp_height = row_ends[count1] - row_starts[count1];
    blob_association[count1] = num_items;
    for (count2=(count1+1); count2<num_blobs; count2++) {
        for(;;) {

          /* Ignore assosciated blobs */
          if (blob_association[count2] != -1)
            break;

          /* Ignore already busted blobs */
          if(row_ends[count1]-row_starts[count1]>MAX_HEIGHT
            || col_ends[count1]-col_starts[count1]>MAX_LENGTH)
            break;
          if(row_ends[count2]-row_starts[count2]>MAX_HEIGHT
            || col_ends[count2]-col_starts[count2]>MAX_LENGTH)
            break;

          /* If there is no vertical overlap */
          if ((temp_line.stop_row <= row_starts[count2]) ||
             (temp_line.start_row >= row_ends[count2]))
            break;

          /* If the horizontal gap is more than some tolerance */
          {
            int hcut;
            hcut = temp_height * 2;
            if(hcut > MAX_HORIZ_DIST)
              hcut = MAX_HORIZ_DIST;
            if( ! (
                    ((temp_line.start_col < col_starts[count2]) &&
                     (temp_line.stop_col > col_ends[count2])) ||
                    ((temp_line.start_col < col_starts[count2]) &&
                     ((col_starts[count2]-temp_line.stop_col)<(hcut))) ||
                    ((temp_line.start_col > col_starts[count2]) &&
                     ((temp_line.start_col-col_ends[count2]) < (hcut)))
                    )
              ) break;
          }

          /* If one surrounds the other */
          if(temp_line.start_row <= row_starts[count2] &&
             temp_line.stop_row >= row_ends[count2] &&
             temp_line.start_col <= col_starts[count2] &&
             temp_line.stop_col >= col_ends[count2])
            break;
          if(temp_line.start_row >= row_starts[count2] &&
```

```
            temp_line.stop_row <= row_ends[count2] &&
            temp_line.start_col >= col_starts[count2] &&
            temp_line.stop_col <= col_ends[count2])
           break;


         /* If the vertical overlap is less than some tolerance */
         {
          int overlap, smaller_height;
          overlap =                                                          80
           ((row_ends[count2] < temp_line.stop_row) ?
            row_ends[count2] : temp_line.stop_row) -
                 ((temp_line.start_row > row_starts[count2]) ?
                  temp_line.start_row : row_starts[count2]);
          smaller_height =
           temp_height > row_ends[count2] - row_starts[count2] ?
                row_ends[count2] - row_starts[count2]
                 : temp_height;
          if( (float)overlap/(float)smaller_height < MIN_FRAC_OVERLAP)
           break;                                                            90
          if( overlap < MIN_OVERLAP)
           break;
         }

         if (row_starts[count2] < temp_line.start_row)
          temp_line.start_row = row_starts[count2];
         if (row_ends[count2] > temp_line.stop_row)
          temp_line.stop_row = row_ends[count2];
         if (col_starts[count2] < temp_line.start_col)
          temp_line.start_col = col_starts[count2];                          100
         if (col_ends[count2] > temp_line.stop_col)
          temp_line.stop_col = col_ends[count2];
         if ((row_ends[count2]-row_starts[count2]) > temp_height)
          temp_height = row_ends[count2] - row_starts[count2];
         one_blob = 0;
         blob_association[count2] = num_items;
         break;
        }
    }
    output[num_items] = temp_line;                                           110
    num_items++;
    if (num_items == num_alloc) {
        output = rrealloc(output, (num_alloc+2)*sizeof(tREGION));
        num_alloc += 2;
    }
  }
 }
 *num_text_lines = num_items;
 return (output);
}                                                                            120


tBITMAP make_region_bitmap(tBITMAP bitmap, int xdim, int ydim, tREGION region)
{
```

```
  int count1, count2;
  tBITMAP output;

  output = BitmapCreate(xdim, ydim);
  for (count1=region.start_row; count1<=region.stop_row; count1++)
    for (count2=region.start_col; count2<=region.stop_col; count2++)
      output[count1][count2] = bitmap[count1][count2];
  return (output);
}
```

```
#include "image.h"

int **threshold(int **input, int xdim, int ydim, int low_thresh_val,
                int high_thresh_val)
{
  int count1, count2;
  int **output;

  output = (int **)ccalloc(ydim, sizeof(int *));
  for (count1=0; count1<ydim; count1++)
    output[count1] = (int *)ccalloc(xdim, sizeof(int));

  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++) {
      if ((input[count1][count2] > low_thresh_val) &&
          (input[count1][count2] < high_thresh_val))
/* make white */
        output[count1][count2] = 255;
      else output[count1][count2] = 0;
    }
  return (output);
}


int **ink_threshold(int **input, int xdim, int ydim, int low_thresh_val,
                    int high_thresh_val)
{
  int count1, count2;
  int **output;

  output = (int **)ccalloc(ydim, sizeof(int *));
  for (count1=0; count1<ydim; count1++)
    output[count1] = (int *)ccalloc(xdim, sizeof(int));

  for (count1=0; count1<ydim; count1++)
    for (count2=0; count2<xdim; count2++) {
      if ((input[count1][count2] > low_thresh_val) &&
          (input[count1][count2] < high_thresh_val))
/* make black */
        output[count1][count2] = 0;
      else output[count1][count2] = 255;
    }
  return (output);
```

```
}
```

```
#ifndef _UTILS_H_
#define _UTILS_H_

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>


void *mmalloc(size_t);
void *ccalloc(size_t, size_t);
void *rrealloc(void *, size_t);
void ffree(void *);

int fatal(char *, char *, char *, ...);
int warning(char *, char *, char *, ...);

int message(char *, ...);

#define SETUP_DEBUG_MSG(var) static int * _DebugLevel_holder_ = &(var)
#define DEBUG_MSG(level, msg, args...) \
if(*(_DebugLevel_holder_) >= level) message(msg, ## args)

#define FATAL(msg, args...) fatal(_FILE_, _FUNCTION_, msg, ## args)
#define WARNING(msg, args...) warning(_FILE_, _FUNCTION_, msg, ## args)

#define FATAL_IF(flag, msg, args...) \
if(flag) fatal(_FILE_, _FUNCTION_, msg, ## args)

#define WARNING_IF(flag, msg, args...) \
if(flag) warning(_FILE_, _FUNCTION_, msg, ## args);

#define FFREE(ptr) ffree(ptr)
#define MMALLOC(size) mmalloc(size)
#define RREALLOC(ptr, size) rrealloc(ptr, size)
#define CCALLOC(nmemb, size) ccalloc(nmemb, size)


#endif /* _UTILS_H_ */
```

Appendix B

OCR Source Code Listings

```
#include "prototypes.h"

BOOL Bordered(tBITMAP, tCOORD);
BOOL LeftOf(tBITMAP, tCOORD, tCOORD);
BOOL RightOf(tBITMAP, tCOORD, tCOORD);
BOOL InLine(tBITMAP, tCOORD, int (* )());

/*************************************************/
/*              HITNDEFLECT.C                    */
/*-----------------------------------------------*/
/* This function tries to find a segmentation    */
/* path using the "Hit and Deflect" algorithm    */
/* without violating dimension constraints on    */
/* the resulting segments.  If such a path is    */
/* found, a pointer to the path is returned.     */
/* otherwise the NULL pointer is returned.       */
/*************************************************/

tNEW_BLOBTYPE *HitNDeflect(tBITMAP bitmap, tNEW_BLOBTYPE input, tCOORD min,
                           tCOORD max, DIRECTION dir)
{
  int count, count1, count2;
  int endrow;
  int (*Advance)();
  tNEW_BLOBTYPE *output;
  tCOORD point, prev_point, prev2_point;

  /* initialize only prev2_point, since prev_point set immediately */
  prev2_point.row = 0;
  prev2_point.col = 0;

  output = (tNEW_BLOBTYPE *)calloc((size_t)2, sizeof(tNEW_BLOBTYPE));
  output[0] = NewBlobCreate();
  output[1] = NewBlobCreate();
  for (count=0; count<HEIGHT; count++) {
    output[0][count].start_blob = input[count].start_blob;
    output[1][count].end_blob = input[count].end_blob;
  }
```

```
if (dir == UP) {
  point.row = max.row;
  point.col = max.col;
  endrow = 0;
  Advance = Up;
  for (count=max.row; count<HEIGHT; count++) {
    output[0][count].end_blob = max.col;
    output[1][count].start_blob = max.col;
  }
}
else {
  point.row = min.row;
  point.col = min.col;
  endrow = HEIGHT-1;
  Advance = Down;
  for (count=0; count<min.row; count++) {
    output[0][count].end_blob = min.col;
    output[1][count].start_blob = min.col;
  }
}

while (point.row != endrow) {
  if (!Bordered(bitmap, point)) {
    output[0][point.row].end_blob = point.col;
    output[1][point.row].start_blob = point.col;
    point.row = (*Advance)(point.row);
  }
  else
    if (!InLine(bitmap, point, Advance)) {
        output[0][point.row].end_blob = point.col;
        output[1][point.row].start_blob = point.col;
        point.row = (*Advance)(point.row);
    }
    else
        if (!LeftOf(bitmap, point, prev2_point))
          if (!RightOf(bitmap, point, prev2_point)) {
            output[0][point.row].end_blob = point.col;
            output[1][point.row].start_blob = point.col;
            point.row = (*Advance)(point.row);
          }
          else {
            --point.col;
            output[0][point.row].end_blob = point.col;
            output[1][point.row].start_blob = point.col;
          }
        else
          if (!RightOf(bitmap, point, prev_point)) {
            ++point.col;
            output[0][point.row].end_blob = point.col;
            output[1][point.row].start_blob = point.col;
          }
          else {
            output[0][point.row].end_blob = point.col;
```

```
            output[1][point.row].start_bl ] = point.col;
            point.row = (*Advance)(point.row);
          }

    prev_point.row = point.row;
    prev_point.col = point.col;
  }
                                                                          100
  if (BlobDimensionsOK(bitmap, input, output[0])
      && BlobDimensionsOK(bitmap, input, output[1]))
    return output;
  else {
    free(output[0]);
    free(output[1]);
    free(output);
    return(NULL);
  }
}                                                                         110


/*********************************************/
/*              BORDERED.C                   */
/*-------------------------------------------*/
/* This function simply checks to see if a   */
/* given point is bordered by a whitespace or */
/* is up against a border in                 */
/* the given map and returns TRUE if it is,  */
/* FALSE otherwise.                          */                          120
/*********************************************/

BOOL Bordered(tBITMAP map, tCOORD point)
{
  if(point.row > 0) {
    if(!map[point.row-1][point.col])
      return(TRUE);
    if(point.col < WIDTH-1)
      if(!map[point.row-1][point.col+1])
          return(TRUE);                                                   130
  }
  else return(TRUE);              /* up against top edge */

  if(point.col < WIDTH-1) {
    if(!map[point.row][point.col+1])
      return(TRUE);
    if(point.row < HEIGHT-1)
      if(!map[point.row+1][point.col+1])
          return(TRUE);
  }                                                                       140
  else return(TRUE);              /* up against right edge */

  if(point.row < HEIGHT-1) {
    if(!map[point.row+1][point.col])
      return(TRUE);
    if(point.col > 0)
```

```
    if(map[point.row+1][point.col-1])
        return(TRUE);
  }
  else return(TRUE);              /* up against bottom edge */

  if(point.col > 0) {
    if(!map[point.row][point.col-1])
      return(TRUE);
    if(point.row > 0)
      if(!map[point.row-1][point.col-1])
        return(TRUE);
  }
  else return(TRUE);              /* up against left edge */

  /* if trickled down, then no neighbors, return false */
  return(FALSE);
}


/*************************************************/
/*            DIRECTIONS.C                       */
/*-----------------------------------------------*/
/* These functions are used for checking if a    */
/* pixel of a bitmap (stored as a char array)    */
/* is bordered in a certain direction by a '1'   */
/* or the character representing a segmentation  */
/* path.                                         */
/*                                               */
/* LeftOf, RightOf, and InLine are used for      */
/* actual checking of the value of a bordering   */
/* pixel.                                        */
/******* ***************************************/


BOOL LeftOf(tBITMAP blobmap, tCOORD point, tCOORD prev2_point)
{
  return ((point.col <= 0) || (blobmap[point.row][point.col-1])
          || ((prev2_point.row == point.row) &&
              (prev2_point.col == point.col)));
}


BOOL RightOf(tBITMAP blobmap, tCOORD point, tCOORD prev2_point)
{
  return ((point.col >= WIDTH-1) || (blobmap[point.row][point.col+1])
          || ((prev2_point.row == point.row) &&
              (prev2_point.col == point.col)));
}


BOOL InLine(tBITMAP blobmap, tCOORD point, int (*forward)())
{
  return (blobmap[(*forward)(point.row)][point.col]);
}
```

```
#include "prototypes.h"


tBITMAP BitmapCreate()
{
 int count;
 tBITMAP temp;
 temp = (tBITMAP)calloc(HEIGHT, sizeof(tBITMAP_ROW));
 for(count=0;count<HEIGHT;count++) {
  temp[count] = (tBITMAP_ROW)calloc(WIDTH, sizeof(char));
 }
 return temp;
}


void BitmapDestroy(tBITMAP bitmap)
{
 int count;
 for (count=0; count<HEIGHT; count++)
  free(bitmap[count]);
 free(bitmap);
}


tINTMAP IntmapCreate(int height, int width)
{
 int count, count2;
 tINTMAP temp;
 temp = (tINTMAP)calloc(height, sizeof(tINTMAP_ROW));
 for(count=0;count<height;count++) {
  temp[count] = (tINTMAP_ROW)calloc(width, sizeof(int));
 }
 return temp;
}


tNEW_BLOBTYPE NewBlobCreate()
{
 return (tNEW_BLOBTYPE)calloc((size_t)HEIGHT, sizeof(struct tBOUNDARY));
}
```

```
#include "prototypes.h"

tNEW_BLOBTYPE *Bridge(tBITMAP bitmap, tNEW_BLOBTYPE input, tCOORD min,
                     tCOORD max)
{
 tNEW_BLOBTYPE *output;
 int count;
 int crossings;
```

```
BOOL on_contour = FALSE;
float slope;
int temp_col;  /* for convenience only */

if (min.row >= max.row)
  return(NULL);

/* This assumes that the blob is cut into only two parts.  So, the */
/* starting point of the input is copied into the first (left) blob, */
/* and the ending point of the input is copied into the second blob. */
output = (tNEW_BLOBTYPE *)cailoc((size_t)2, sizeof(tNEW_BLOBTYPE));
output[0] = NewBlobCreate();
output[1] = NewBlobCreate();
for (count=0; count<HEIGHT; count++) {
  output[0][count].start_blob = input[count].start_blob;
  output[1][count].end_blob = input[count].end_blob;
}

if ((max.col-min.col) != 0)
  slope = (float)(max.row-min.row)/(float)(max.col-min.col);

for (count=min.row; count<=max.row; count++) {
  if (crossings > MAX_CONTOUR_CROSSINGS)
    return(NULL);
  if ((max.col-min.col) != 0)
    (double)output[0][count].end_blob =
        rint((double)(1/slope)*(count-min.row+(slope*min.col)));
  else output[0][count].end_blob = max.col;
  output[1][count].start_blob = output[0][count].end_blob;
  temp_col = output[0][count].end_blob;
  if (bitmap[count][temp_col])
        on_contour = TRUE;
  else {
    if (on_contour == TRUE) {
        ++crossings;
        on_contour = FALSE;
    }
  }
}

for (count=0; count<min.row; count++) {
  output[0][count].end_blob = min.col;
  output[1][count].start_blob = max.col;
}

for (count=max.row; count<=HEIGHT; count++) {
  output[0][count].end_blob = max.col;
  output[1][count].start_blob = max.col;
}

if (BlobDimensionsOK(bitmap, input, output[0])
    && BlobDimensionsOK(bitmap, input, output[1]))
  return output;
else {
```

```
    free(output[0]);
    free(output[1]);
    free(output);
    return(NULL);
  }
}

                                                                    70
int BlobDimensionsOK(tBITMAP bitmap, tNEW_BLOBTYPE original_blob,
                     tNEW_BLOBTYPE test_blob)
{
  int okay;
  int weight;
  int original_startr, original_endr, test_startr, test_endr;

  weight = FindBlobWeight(bitmap, test_blob);
  original_startr = FindExtremePoint(bitmap, original_blob, ROW, START);
  original_endr = FindExtremePoint(bitmap, original_blob, ROW, END);        80
  test_startr = FindExtremePoint(bitmap, test_blob, ROW, START);
  test_endr = FindExtremePoint(bitmap, test_blob, ROW, END);

  return
    (((float)weight >= (float)((float)weight*(float)WT_RATIO_THRESHOLD))
     &&
     ((float)(original_endr-original_startr) >=
      (float)((float)(test_endr-test_startr)*(float)HT_RATIO_THRESHOLD)));
}
```

```
#include "prototypes.h"


char Categorize(tBITMAP bitmap, tNEW_BLOBTYPE input, int height_range,
                int midline)
{
  int quarter_height, half_height;
  int startr, stopr, startc, stopc;

  quarter_height = .25;                                             10
  half_height = .40;

  startr = FindExtremePoint(bitmap, input, ROW, START);
  stopr = FindExtremePoint(bitmap, input, ROW, END);
  startc = FindExtremePoint(bitmap, input, COL, START);
  stopc = FindExtremePoint(bitmap, input, COL, END);

  if (startr > midline) {
     if((stopr - startr) < (quarter_height * height_range)){
        if (((float)(stopr-startr)/(float)(stopc-startc)) < .3){        20
          return ('-');
        }
        else {
          return ('.');
        }
```

```
    }
    else {
        return (',');
    }
  }
  else {
    if((stopr - startr) < (half_height * height_range)) {
        return ('G');
    }
  }
 return ('D');
}
```

```
#include<stdlib.h>
#include<stdio.h>

#include"getblob.h"

void main(int argc, char *argv[])
{
 FILE *infile;
 FILE *outfile;

 unsigned char byte, data[8], bits;
 int count1, count2, count3;

 char buffer[HEIGHT][WIDTH];

 char widthstring[100];
 char heightstring[100];
 char declstring[100];


 if(argc!=3) {
  printf("Usage:  %s <check_file> <bitmap_file>\n", argv[0]);
  exit(0);
 }

 if( (infile=fopen(argv[1], "r")) == NULL) {
  printf("Couldn't open file %s for reading\n", argv[1]);
  exit(0);
 }

 if( (outfile=fopen(argv[2], "w")) == NULL) {
  printf("Couldn't open file %s for writing\n", argv[2]);
  exit(0);
 }
```

```
  sprintf(widthstring, "#define %s_width 300\n",argv[2]);
  sprintf(heightstring, "#define %s_height 100\n", argv[2]);
  sprintf(declstring, "static unsigned char %s_bits[] = {\n", argv[2]);

  fprintf(outfile, widthstring);
  fprintf(outfile, heightstring);
  fprintf(outfile, declstring);

  for(count2=0;count2<300;count2++) {
    for(count1=0;count1<100;count1++) {
      if( fscanf(infile, "%c", &(buffer[count1][count2])) == 0 ) {
          printf("Error in check file %s\n", argv[1]);
          exit(0);
      }
    }
  }

  for(count1=0;count1<100;count1++) {
    for(count2=0;count2<300;count2++) {
      data[count2%8] = (buffer[count1][count2] > (unsigned char)64);
      if( (!(count2%8) && (count2!=0)) || (count2==299) ) {
          for(count3=0;count3<8;count3++) {
            if( data[count3] ) bits += (unsigned char)
                                    pow((double)2, (double)count3);
          }
          if( count1==99 && count2==299) {
            fprintf(outfile, "%s};\n", uchar2hexstring(bits));
          }
          else {
            fprintf(outfile, "%s,", uchar2hexstring(bits));
          }
      }
    }
  }

  fclose(infile);
  fclose(outfile);
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <assert.h>

/***** getblob.c *****/
#define TOLERANCE 2
#define HEIGHT 100
#define WIDTH 300
```

```
/***** recognize.c *****/
#define nInputNodes     256
#define nHiddenNodes    40
#define nOutputNodes    10
#define nHiddenLayer    1
#define INT_HEIGHT      16
#define INT_WIDTH       16
#define NUM_OF_OUTPUTS 10   /* pjliu: taken from datatype.h */

/* pjliu: types were float huge, instead of just float */
/* activation nodes */
float iNode[nInputNodes];
float hNode[nHiddenNodes];
float oNode[nOutputNodes];
float inputHNode[nHiddenNodes];
float inputONode[nOutputNodes];

/* synaptic weights and delta weights and bias */
float wt1[nHiddenNodes][nInputNodes];
float wt2[nOutputNodes][nHiddenNodes];
float bias1[nHiddenNodes];
float bias2[nOutputNodes];

float wt3[nHiddenNodes][nInputNodes];
float wt4[nOutputNodes][nHiddenNodes];
float bias3[nHiddenNodes];
float bias4[nOutputNodes];

/***** segmentation.c *****/
#define INFINITE -1
#define BLOB_THRESHOLD 15
#define ASPECT_CUTOFF .75
#define PROFILE_SIZE 3
#define NUM_PAIRS 9    /* is PROFILE_SIZE*PROFILE_SIZE */

#define GET_ASPECT(asp, bdims) \
((bdims.start_x == bdims.end_x) ? (asp = INFINITE) : \
 (asp = ((float)(((float)(bdims.end_y - bdims.start_y))/ \
                 ((float)(bdims.end_x - bdims.start_x))))))

/* #define NOT_FRAG(sdims,bdims) */

/***** profile.c *****/
#define BORDER_DEVIATION .33 /* for making the 2 side borders when profiling */
#define TRUE 1
#define FALSE 0
typedef int BOOL;

/***** bridge.c *****/
#define MAX_CONTOUR_CROSSINGS 1
#define WT_RATIO_THRESHOLD .20
#define HT_RATIO_THRESHOLD .50

/***** thinning.c and thicken.c *****/
```

```
#define PADDED_X 18
#define PADDED_Y 18

/***** slant.c *****/
#define SLANTED_HEIGHT 16
#define SLANTED_WIDTH 72

/*****************************************************************************/
/***** getblob.c *****/
typedef char *tBITMAP_ROW;
typedef tBITMAP_ROW *tBITMAP;

typedef int *tINTMAP_ROW;
typedef tINTMAP_ROW *tINTMAP;

struct tBOUNDARY {
  int start_blob, end_blob;
};
typedef struct tBOUNDARY *tNEW_BLOBTYPE;

typedef enum {START, END} START_END;
typedef enum {ROW, COL} ROW_COL;

typedef enum {COMPUTE_HEIGHT, MIDLINE} HEIGHT_MIDLINE;

/***** for the dimensions of the blob *****/
/*
  struct tDIMENSION{
  int weight;
  int start_x, start_y, end_x, end_y;
};

struct BLOBTYPE {
  tBITMAP blob_bitmap;
  tDIMENSION blob_dimension;
};
*/

/***** for segmentation information *****/
typedef struct tCOORD tCOORD;
struct tCOORD {
  int row, col;
};


typedef struct tPATHTYPE tPATHTYPE;
struct tPATHTYPE {
 tCOORD path_point;
 struct tPATHTYPE *next;
};

typedef struct tSEGMENT tSEGMENT;
struct tSEGMENT {
  int *split;
```

```
  tPATHTYPE *split_paths;
};

/***** profile.c *****/
typedef enum {TOP, BOTTOM} VIEW;

/***** HDS.c *****/
typedef enum {UP, DOWN} DIRECTION;
```

```
/* This file contains not only procedures to get the blobs, but also some */
/* other basic blob utilities, such as determining the blob weight */

#include "prototypes.h"


/* Iterative wrapper for GetBlob.  Makes copy of input bitmap, */
/* because GetBlob clobbers it (the copy).  Returns the number of */
/* blobs found. */

int GetBlobs(tBITMAP input, tBITMAP **outblobs)
{
  int count1,count2;
  tBITMAP copy;
  tBITMAP temp;
  int num_blobs=0;

  copy = BitmapCreate();

  for(count1=0;count1<WIDTH;count1++) {
    for(count2=0;count2<HEIGHT;count2++) {
      copy[count2][count1] = input[count2][count1];
    }
  }

  while(1) {
    temp=NULL;
    GetBlob(copy, &temp, -1, -1);
    if( temp==NULL ) break;
    /* Test for valid blob before wasting memory here. */
    /* Free blob if not valid. */
    (*outblobs)[num_blobs] = temp;
    num_blobs++;
  }

  BitmapDestroy(copy);
  return num_blobs;
}


void GetBlob(tBITMAP input, tBITMAP *output, int start_row, int start_col)
{
  int row = start_row;
  int col = start_col;
```

```
  int rcount;
  int ccount;

  if( (start_row==-1) && (start_col==-1) ) {   /* If we just started  */
    /* No start point, so brute force find any dark pixel. */
    /* The simplest robust way to do this is simply starting from one  */
    /* corner of the bitmap and scanning.  */

    for(col=0;col<WIDTH;col++) {
      for(row=0;row<HEIGHT;row++) {
          if( input[row][col] ) break;
      }
      if( row!=HEIGHT ) break;
    }
  }

  if( col==WIDTH && row==HEIGHT ) {   /* We found no dark pixels  */
    return;
  }


  /* If output is null, we need to malloc up the space to put our */
  /* bitmap.  */
  if((*output) == NULL) (*output)=BitmapCreate();


  /* Look in a TOLERANCE neighborhood of row, col for dark pixels.  */
  /* Recursively call self, at each dark pixel on the edge of that  */
  /* neightborhood.  */

  for(ccount=col-TOLERANCE;ccount<=col+TOLERANCE;ccount++) {
    for(rcount=row-TOLERANCE;rcount<=row+TOLERANCE;rcount++) {
      if(ccount>=0 && ccount<WIDTH && rcount>=0 && rcount<HEIGHT) {
          if( input[rcount][ccount] ) {
            input[rcount][ccount] = (char)0;
            (*output)[rcount][ccount] = (char)1;
            GetBlob(input, output, rcount, ccount);
          }
      }
    }
  }
}


tNEW_BLOBTYPE ConvertBlobs (tBITMAP blob)
{
  tNEW_BLOBTYPE temp_blob;
  int count1, count2;
  int min, max;

  temp_blob = NewBlobCreate();
  for (count1=0; count1<HEIGHT; count1++) {
    min = WIDTH;
    max = 0;
```

```
    for (count2=0; count2<WIDTH; count2++) {
      if (blob[count1][count2]) {
            if (count2<min)
              min=count2;
            if (count2>max)
              max=count2;
      }
    }
    temp_blob[count1].start_blob = min;
    temp_blob[count1].end_blob = max;
  }
  return temp_blob;
}


tBITMAP NewBlob2OldBlob(tBITMAP bitmap, tNEW_BLOBTYPE input)
{
  tBITMAP temp_blob;
  int count1, count2;

  temp_blob = BitmapCreate(HEIGHT, WIDTH);
  for (count1=0; count1<HEIGHT; count1++)
    for (count2=input[count1].start_blob; count2<=input[count1].end_blob;
          count2++)
      if (bitmap[count1][count2])
          temp_blob[count1][count2] = 1;
  return temp_blob;
}


int FindBlobWeight(tBITMAP bitmap, tNEW_BLOBTYPE blob)
{
  int temp_weight = 0;
  int count1, count2;

  for (count1=0; count1<HEIGHT; count1++)
    for (count2=blob[count1].start_blob;count2<=blob[count1].end_blob;
          count2++)
      if (bitmap[count1][count2])
          temp_weight++;
  return temp_weight;
}


int FindExtremePoint(tBITMAP bitmap, tNEW_BLOBTYPE blob, ROW_COL row_col,
                     START_END start_end)
{
  int count1, count2;
  int min, max;

  max = 0;
  min = WIDTH;
  for (count1=0; count1<HEIGHT; count1++)
    for (count2=blob[count1].start_blob;count2<=blob[count1].end_blob; count2++)
```

```
    if (bitmap[count1][count2]) {
        if (row_col == COL) {
          if (start_end == START) {
            if (min > count2) {
              min = count2;
            }
          }
          else
            if (max < count2) {
              max = count2;
            }
        }
        else
          if (start_end == START) {
            if (min > count1) {
              min = count1;
            }
          }
          else
            if (max < count1) {
              max = count1;
            }
    }
  if (start_end == START)
    return min;
  else return max;
}

float GetAspect(int startc, int stopc, int startr, int stopr)
{
  float aspect;

  if (startc == stopc)
    aspect = INFINITE;
  else
    aspect = (float)(((float)(stopr-startr)) / ((float)(stopc-startc)));
  return aspect;
}


int HeightAndMidline(tBITMAP bitmap, tNEW_BLOBTYPE *input, int num_blobs,
                     HEIGHT_MIDLINE compute_item)
{
  int count;
  int max, min;
  int temp_range;
  int startr, stopr;

  /* initialize */
  max = 0;
  min = 100;

  for (count=0; count<num_blobs; count++) {
```

```
    startr = FindExtremePoint(bitmap, input[count], ROW, START);
    if (startr < min)
      min = startr;
    stopr = FindExtremePoint(bitmap, input[count], ROW, END);
    if (stopr > max)
      max = stopr;
  }
  if (compute_item == COMPUTE_HEIGHT)
    return (max-min);
  else return ((int)floor(((float)(max-min)/2)+.5)+min);
}
```

```
#include "prototypes.h"

extern unsigned char *test_bitmap;

void main()
{
  int count, num_blobs;
  int count1;
  tBITMAP bitmap;
  tBITMAP *blobs;
  tNEW_BLOBTYPE *new_blobs, *temp_blobs;

  char filename[100];
  char *result;
  char *segment_result;
  int valid_blobs;
  int midline, height_range;
  int total_chars = 0;
  int num_chars;
  int comma_count, cent_count;

  char *recognizer_file = "/mit/pjliu/winbank/hand.wts";
  ReadWeights(recognizer_file, 0);

  result = (char *)malloc((size_t)20 * sizeof(char));

  /* Convert array of unsigned char (the bitmap) into our bitmap type */
  /* not needed for real winbank -- for testing purposes only */
  bitmap = BitmapCreate();
  bitmap_to_char_array(test_bitmap, bitmap, HEIGHT, WIDTH);

  blobs = (tBITMAP *)malloc((size_t)30 * sizeof(tBITMAP));
  num_blobs = GetBlobs(bitmap, &blobs);

  temp_blobs =(tNEW_BLOBTYPE *)malloc((size_t)num_blobs*sizeof(tNEW_BLOBTYPE));
  new_blobs = (tNEW_BLOBTYPE *)malloc((size_t)num_blobs*sizeof(tNEW_BLOBTYPE));
  valid_blobs = 0;
  for (count=0; count<num_blobs; count++) {
    temp_blobs[count] = ConvertBlobs(blobs[count]);
    if (FindBlobWeight(bitmap, temp_blobs[count]) > BLOB_THRESHOLD) {
```

```
    new_blobs[valid_blobs] = temp_blobs[count];
    valid_blobs++;
   }
  }

  num_blobs = valid_blobs;

  for(count=0;count<num_blobs;count++) {
   sprintf(filename, "blob-%dbmp", count);
   fwrite_bitmap(filename, (NewBlob2OldBlob(bitmap, new_blobs[count])),
                 HEIGHT, WIDTH);
  }


  height_range = HeightAndMidline(bitmap, new_blobs, num_blobs,
                                  COMPLETE_HEIGHT);
  midline = HeightAndMidline(bitmap, new_blobs, num_blobs, MIDLINE);

  for (count=0; count<num_blobs; count++) {
   segment_result = Segment(bitmap, new_blobs[count], height_range, midline,
                            &num_chars, count);
   for (count1=total_chars; count1<=(total_chars+num_chars); count1++)
    result[count1] = segment_result[count1-total_chars];
   total_chars += num_chars;
   printf("%s\n", segment_result);
  }

  /* basic testing for confusion with periods and commas */
  for (count=0; count<total_chars; count++) {
   if (result[count] == ',') {
    comma_count = 0;
    for (count1 = count; count1<(count+3); ++count1)
        if ((result[count1] >= 48) && (result[count1] <= 57))
          comma_count++;
    if (comma_count == 2)
        result[count] = '.';
   }
   if (result[count] == '.') {
    cent_count = 0;
    for (count1 = count; count1<(count+3); ++count1)
        if ((result[count1] >= 48) && (result[count1] <= 57))
          cent_count++;
    if (cent_count == 3)
        result[count] = ',';
   }
  }
  result[total_chars] = 0;
  printf("%s\n", result);
 }
```

```
#include "prototypes.h"
```

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
#define MAX(x,y) ((x) > (y) ? (x) : (y))


tINTMAP Normalize(tBITMAP bitmap, tNEW_BLOBTYPE input)
{
 int startr, stopr, startc, stopc;
 tINTMAP output;
 float total;
 int count,count2,count3,count4;
 float rowscale, colscale;
 float rowfrac, colfrac;
 float block_startr, block_stopr, block_startc, block_stopc;

 output=IntmapCreate(INT_HEIGHT, INT_WIDTH);

 startr = FindExtremePoint(bitmap, input, ROW, START);
 stopr = FindExtremePoint(bitmap, input, ROW, END);
 startc = FindExtremePoint(bitmap, input, COL, START);
 stopc = FindExtremePoint(bitmap, input, COL, END);

 rowscale = (float)(stopr-startr+1.0)/16.0;
 colscale = (float)(stopc-startc+1.0)/16.0;

 for(count=0;count<16;count++) {
  for(count2=0;count2<16;count2++) {
   total = 0.0;

   block_startr = (float)count*rowscale + (float)startr;
   block_stopr = block_startr+rowscale;
   if (block_stopr >= HEIGHT)
       block_stopr--;

   block_startc = (float)count2*colscale + (float)startc;
   block_stopc = block_startc + colscale;
   if (block_stopc >= WIDTH)
       block_stopc--;

   for(count3=(int)floor((double)block_startr);
       count3<=(int)floor((double)block_stopr); count3++) {
      for(count4=(int)floor((double)block_startc);
        count4<=(int)floor((double)block_stopc); count4++) {
       if( !((rowfrac=block_stopr-block_startr) < 1.0) ) {
        rowfrac = MIN( (float)count3+1.0-block_startr, 1.0);
        rowfrac = MIN( rowfrac, block_stopr-(float)count3);
       }
       if( !((colfrac=block_stopc-block_startc) < 1.0) ) {
        colfrac = MIN( (float)count4+1.0-block_startc, 1.0);
        colfrac = MIN( colfrac, block_stopc-(float)count4);
       }

       total += rowfrac * colfrac *
        (float)(bitmap[count3][count4] &&
                (count4>=input[count3].start_blob) &&
```

```
                    (count4<=input[count3].end_blob)).
            }
        }
        if (total > colscale*rowscale/3.0) output[count][count2] = 1;
    }
  }
  return output;
}
```

```
#include "prototypes.h"

int PostProcess(tINTMAP bit, float *act_val)
{
  int i ,j;
  int bits[INT_HEIGHT*INT_WIDTH];
  int ind_maxact1 = 0, ind_maxact2 = 0 ;
  float maxact1 = - 1.0 , maxact2 = -1.0 ;
  int index=0, result;

  for(i=0;i < INT_HEIGHT;i++) {
    for(j=0;j < INT_WIDTH;j++) {
      bits[index]=bit[i][j];
      index++;
    }
  }
  for(i = 0; i<10; i++) {
    if(maxact1 <  act_val[i]) {
      maxact1 = act_val[i] ;
      ind_maxact1 = i ;
    }
  }
  for(i=0; i<10; i++) {
    if(i != ind_maxact1) {
      if(maxact2 < act_val[i]) {
          maxact2 = act_val[i] ;
          ind_maxact2 = i ;
      }
    }
  }

  if((ind_maxact1 == 8 && ind_maxact2 == 9) ||
     (ind_maxact1 == 9 && ind_maxact2 == 8))
    return(check8and9(bits)) ;
  else if((ind_maxact1 == 3 && ind_maxact2 == 5) ||
          (ind_maxact1 == 5 && ind_maxact2 == 3))
    return(check3and5(bits)) ;
  else if((ind_maxact1 == 5 && ind_maxact2 == 8) ||
          (ind_maxact1 == 8 && ind_maxact2 == 5))
    return(check5and8(bits)) ;
  else if((ind_maxact1 == 4 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 4))
    return(check4and9(bits)) ;
  else if((ind_maxact1 == 2 && ind_maxact2 == 6) ||
```

```
        (ind_maxact1 == 6 && ind_maxact2 == 2))
  return(check2and6(bits)) ;
else if((ind_maxact1 == 1 && ind_maxact2 == 6) ||
        (ind_maxact1 == 6 && ind_maxact2 == 1))
  return(check1and6(bits)) ;
else if((ind_maxact1 == 7 && ind_maxact2 == 9) ||
        (ind_maxact1 == 9 && ind_maxact2 == 7))
  return(check7and9(bits)) ;
else if((ind_maxact1 == 5 && ind_maxact2 == 6) ||
        (ind_maxact1 == 6 && ind_maxact2 == 5))
  return(check5and6(bits)) ;
else if((ind_maxact1 == 5 && ind_maxact2 == 9) ||
        (ind_maxact1 == 9 && ind_maxact2 == 5))
  return(check5and9(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 8) ||
        (ind_maxact1 == 8 && ind_maxact2 == 3))
  return(check3and8(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 9) ||
        (ind_maxact1 == 9 && ind_maxact2 == 3))
  return(check3and9(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 6) ||
        (ind_maxact1 == 6 && ind_maxact2 == 3))
  return(check3and6(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 1) ||
        (ind_maxact1 == 1 && ind_maxact2 == 0))
  return(check0and1(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 2) ||
        (ind_maxact1 == 2 && ind_maxact2 == 0))
  return(check0and2(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 3) ||
        (ind_maxact1 == 3 && ind_maxact2 == 0))
  return(check0and3(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 4) ||
        (ind_maxact1 == 4 && ind_maxact2 == 0))
  return(check0and4(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 5) ||
        (ind_maxact1 == 5 && ind_maxact2 == 0))
  return(check0and5(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 6) ||
        (ind_maxact1 == 6 && ind_maxact2 == 0))
  return(check0and6(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 7) ||
        (ind_maxact1 == 7 && ind_maxact2 == 0))
  return(check0and7(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 9) ||
        (ind_maxact1 == 9 && ind_maxact2 == 0))
  return(check0and9(bits)) ;
else if((ind_maxact1 == 1 && ind_maxact2 == 8) ||
        (ind_maxact1 == 8 && ind_maxact2 == 1))
  return(check1and8(bits)) ;
else if((ind_maxact1 == 1 && ind_maxact2 == 9) ||
        (ind_maxact1 == 9 && ind_maxact2 == 1))
  return(check1and9(bits)) ;
else return(ind_maxact1) ;
```

```
}

int check1and2(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      total++ ;
      if((j>=0 && j<5) || (j>10 && j<16))
           if(bits[total] == 1)
             mid_ones++ ;
    }
  if(mid_ones > 2)
    return(2);
  else
    return(1);
}


int  check1and3(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      total++ ;
      if((j>=0 && j<5) || (j>10 && j<16))
           if(bits[total] == 1)
             mid_ones++ ;
    }
  if(mid_ones > 2)
    return(3) ;
  else
     return(1) ;
}
```

```
int  checkland4(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      total++ ;
      if((j>=0 && j<5) || (j>10 && j<16))
           if(bits[total] == 1)
             mid_ones++ ;
    }
  if(mid_ones > 2)
    return(4) ;
  else
    return(1) ;
}


int  checkland5(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      total++ ;
      if((j>=0 && j<5) || (j>10 && j<16))
           if(bits[total] == 1)
             mid_ones++ ;
    }
  if(mid_ones > 2)
    return(5) ;
  else
    return(1) ;
}


int  checkland7(int * bits)
```

```
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;                                                    210
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      total++ ;                                                   220
      if((j>=0 && j<5) || (j>10 && j<16))
          if(bits[total] == 1)
            mid_ones++ ;
    }
  if(mid_ones > 2)
    return(7) ;
  else
    return(1) ;
}
                                                                  230

int check0and1(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
  for (i = 0; i < 16; i++) {                                      240
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i > 9 && i < 16) {
          pres_bit = bits[total];
          if (pres_bit != orig) {
            changes++:
            if (changes > 2)                                      250
              second_changes++;
          }
          orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(1) ;
  else
    return(0) ;                                                   260
```

```
}


int check0and2(int *bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int change_matrix[20];
 int *origout2, i, j, total = 0  ;
 int trials = 0;
 int second_changes = 0;                                                    270
 int pres_bit, orig, changes;

 for (i = 0; i < 16; i++) {
  orig = 0;
  pres_bit = 0;
  changes = 0;
  for (j = 0; j < 16; j++) {
   total++ ;
   if (i > 9 && i < 16) {
        pres_bit = bits[total];                                             280
        if (pres_bit != orig) {
          changes++;
          if (changes > 2)
            second_changes++;
        }
        orig = pres_bit;
   }
  }
 }
 if (second_changes < 2)                                                    290
  return(2) ;
 else return(0) ;
}


int  check0and3(int *bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int change_matrix[20];
 int *origout2, i, j, total = 0  ;                                          300
 int trials = 0;
 int second_changes = 0;
 int pres_bit, orig, changes;

 for (i = 0; i < 16; i++) {
  orig = 0;
  pres_bit = 0;
  changes = 0;
  for (j = 0; j < 16; j++) {
   total++ ;                                                                310
   if (i > 9 && i < 16) {
        pres_bit = bits[total];
        if (pres_bit != orig) {
          changes++;
```

```
            if (changes > 2)
              second_changes++;
            }
            orig = pres_bit;
        }
      }
    }
  }
  if (second_changes < 2)
    return(3) ;
  else return(0) ;
}


int  check0and4(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int which_one = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(4) ;
  else return(0) ;
}


int  check0and5(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
```

```
  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(5) ;
  else return(0) ;
}


int check0and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i >= 0 && i < 9) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(6) ;
  else return(0) ;
```

```
}


int  check0and7(int * bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int change_matrix[20];
 int *origout2, i, j, total = 0  ;                                    430
 int trials = 0;
 int second_changes = 0;
 int pres_bit, orig, changes;

 for (i = 0; i < 16; i++) {
  orig = 0;
  pres_bit = 0;
  changes = 0;
  for (j = 0; j < 16; j++) {
   total++ ;                                                          440
   if (i > 9 && i < 16) {
        pres_bit = bits[total];
        if (pres_bit != orig) {
          changes++;
          if (changes > 2)
            second_changes++;
        }
        orig = pres_bit;
    }
  }                                                                   450
 }
 if (second_changes < 2)
  return(7) ;
 else return(0) ;
}


int check0and8(int * bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];           460
 int total = 0 ;
 int changed;
 int *origout2, i, j;
 int trials = 0;
 int rejected = 0;
 int mid_ones = 0 ;
 int wrong_ones = 0;
 int orig  pres_bit;

 for (i = 0; i < 16; i++)                                            470
  for (j = 0; j < 16; j++) {
   total++ ;
   if(i>=6 && i<9 && j>=6 && j<9)
        if(bits[total])
          mid_ones++ ;
  }
```

```
  if(mid_ones > 2)
    return(8) ;
  else return(0) ;
}

int  check0and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int which_one = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit; }
    }
  }
  if (second_changes < 2)
    return(9) ;
  else return(0) ;
}

int  check1and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j;
  int trials = 0;
  int which_one = 0;
  int total_number = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
  int index = 0 ;

  for (i = 0; i < 20; i++)
    change_matrix[i] = 0;

  changes = 0;
```

```
  total_number++;
  second_changes = 0;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      index++ ;
      if (i > 9 && i < 16) {                                          540
            pres_bit = bits[index] ;
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit;
      }
    }
  }                                                                   550
  if (second_changes < 2)
    return(1) ;
  else return(6) ;
}


int  check1and8(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];                                              560
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int which_one = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;                                                      570
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit;                                          580
      }
    }
  }
  if (second_changes < 2)
```

```
    return(1) ;
  else return(8) ;
}


int  check1and9(int * bits)                                          590
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;                                                    600
    changes = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i >= 0 && i < 9) {
           pres_bit = bits[total];
           if (pres_bit != orig) {
             changes++;
             if (changes > 2)
               second_changes++;
           }                                                         610
           orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(1) ;
  else return(9) ;
}

                                                                     620
int  check2and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int zone1_ones, zone2_ones, zone1a_ones, zone2a_ones;
  int zone1b_ones, zone2b_ones, zone1c_ones, zone2c_ones;             630
  int wrong_ones = 0;
  int orig, pres_bit;

  zone1_ones = 0;
  zone2_ones = 0;
  zone1a_ones = 0;
  zone2a_ones = 0;
  zone1b_ones = 0;
```

```
  zone2b_ones = 0;
  zone1c_ones = 0;
  zone2c_ones = 0;

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    for (j = 0; j < 16; j++) {
      total++ ;
      if (i > 3 && i < 7) {
          pres_bit = bits[total] ;
          if (pres_bit != 0) {
            if (j < 6 && i != 4)
              zone1_ones++;
            if (j < 7 && i != 4)
              zone1a_ones++;
            if (j < 8 && i != 4)
              zone1b_ones++;
            if (j < 8)
              zone1c_ones++;
            if (j > 7)
              zone2c_ones++;
            if (j > 7 && i != 6)
              zone2b_ones++;
            if (j > 8 && i != 6)
              zone2a_ones++;
            if (j > 9 && i != 6)
              zone2_ones++;
          }
      }
    }
  }
  if (zone1_ones == zone2_ones) {
    if (zone1a_ones == zone2a_ones) {
      if (zone1b_ones == zone2b_ones) {
          zone1_ones = zone1c_ones;
          zone2_ones = zone2c_ones;
      }
      else {
          zone1_ones = zone1b_ones;
          zone2_ones = zone2b_ones;
      }
    }
    else {
      zone1_ones = zone1a_ones;
      zone2_ones = zone2a_ones;
    }
  }
  if (zone1_ones > zone2_ones)
    return(6) ;
  if (zone2_ones > zone1_ones)
    return(2) ;
  if (zone1_ones == zone2_ones)
    printf("cannot make a decision between 2 ard 6\n") ;
}
```

```
int  check3and5(int * bits)
{
  int i, j , sum = 0, total = 0 .
  double average ;
  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)                                          700
      {
          total++ ;
          if (i >= 3 && i <= 7)
            if (bits[total] == 1)
              sum = sum+j ;
      }
  average = sum/16.0 ;
  if(average >= 8.0)
    return(3) ;
  else                                                                710
    return(5) ;
}


int  check3and6(int * bits)
{
  int actual, *pnumeral, *origout. *origin, matrix[16][16];
  int changed, firsts[16], lasts[16]. changed2;
  int *origout2, i, j;
  int first_max, second_max;                                          720
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;
  int wrong_ones = 0;
  int rejected = 0;                                                   730
  int total_number = 0;
  int index = 0, index2 = 0  ;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;
  }
                                                                      740
  first_max = 0;
  second_max = 0;

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
```

```
  }

  how_many++;

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    first_sum = 0;
    changed = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
      index++ ;
      pres_bit = bits[index] ;
      if (pres_bit != 0 && changed == 0) {
            firsts[i] = j;
            changed = 1;
      }
    }
    for (j = 15; j >= 0; j = j - 1) {
      index2 = i*16 + j ;
      pres_bit = bits[index2] ;
      if (pres_bit != 0 && changed2 == 0) {
            lasts[i] = j;
            changed2 = 1;
      }
    }

    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i];
  }

  for (j = 8; j < 11; j++) {
    if (firsts[j] > first_max) {
      second_max = first_max;
      first_max = firsts[j];
    }
    else
      if (firsts[j] > second_max)
            second_max = firsts[j];
    first_sum = first_sum + firsts[j];
  }

  first_avg = ((double) (first_max + second_max)) / 2.0;

  if (first_avg > 5.01)
    return(3) ;
  else return(6) ;
}


int  check3and8(int * bits)
{
  int index = 0 ;
  int index2  = 0 ;
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
```

```
int changed, firsts[16], lasts[16], changed2;
int *origout2, i, j;
int zone1_ones, zone2_ones;
int pres_bit, total[16], total2[16];
double avg_position[16], avg2_position[16];
int max, min;
int first_sum, last_sum, diff_sum;
int first_pos[16], last_pos[16], first_hits[16];
double first_avg, last_avg, diff_avg, first_hit_avg[16];
int how_many = 0;
int wrong_ones = 0;
int rejected = 0;
int total_number = 0;

for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  first_hits[i] = 0;
  last_pos[i] = 0;
  total[i] = 0;
  total2[i] = 0;
}

for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0;
}

how_many++;

for (i = 0; i < 16; i++) {
  pres_bit = 0;
  first_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
    index++ ;
    pres_bit = bits[index] ;
    if (pres_bit != 0 && changed == 0) {
        firsts[i] = j;
        changed = 1;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;
        changed2 = 1;
    }
  }
  total[i] = total[i] + firsts[i];
  total2[i] = total2[i] + lasts[i];
}
```

```
  for (i = 0, max = lasts[i], min = lasts[i]; i < 16; i++)
    if ((i > 2 && i < 5) || (i > 8 && i < 11))
      first_sum = first_sum + firsts[i];

  first_avg = ((double) first_sum) / 4.0;
                                                                    860
  if (first_avg > 5.49)
    return(3) ;
  else return(8) ;
}


int  check3and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;                     870
  int *origout2, i, j;
  int first_max, second_max;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;                                                 880
  int wrong_ones = 0;
  int rejected = 0;
  int total_number = 0;
  int index = 0 , index2 = 0 ;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;                                                  890
  }

  first_max = 0;
  second_max = 0;

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
  }
                                                                    900
  how_many++;

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    first_sum = 0;
    changed = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
```

```
      index++ ;
      pres_bit =  bits[index] ;
      if (pres_bit != 0 && changed == 0) {
           firsts[i] = j;
           changed = 1;
      }
     }
     for (j = 15; j >= 0; j = j - 1) {
      index2 = i*16 + j ;
      pres_bit = bits[index2] ;
      if (pres_bit != 0 && changed2 == 0) {
           lasts[i] = j;
           changed2 = 1;
      }
     }

     total[i] = total[i] + firsts[i];
     total2[i] = total2[i] + lasts[i];
   }

   for (j = 3; j < 5; j++)
     first_sum = first_sum + firsts[j];

   first_avg = ((double) first_sum) / 2.0;

   if (first_avg > 4.99)
     return(3) ;
   else  return(9) ;
}


int  check4and9(int * bits)
{
   int actual, *pnumeral, *origout, *origin, matrix[16][16];
   int changed, firsts[16], lasts[16], changed2;
   int *origout2, i, j;
   int first_max, second_max;
   int zone1_ones, zone2_ones;
   int pres_bit, total[16], total2[16];
   double avg_position[16], avg2_position[16];
   int max, min;
   int first_sum, last_sum, diff_sum;
   int first_pos[16], last_pos[16];
   double first_avg, last_avg, diff_avg;
   int how_many = 0;
   int wrong_ones = 0;
   int rejected = 0;
   int total_number = 0;
   int index = 0 ;

   pnumeral = &matrix[0][0];
   /* origout = p1; */
   /* origout2 = p2; */
   origin = pnumeral;
```

```
  for (i = 0; i < 16; i+ -) {
    first_pos[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;
  }

  first_max = 0;
  second_max = 0;

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
  }

  how_many++;

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    first_sum = 0;
    changed  = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
      index = j*16 + i ;
      pres_bit = bits[index] ;
      if (pres_bit != 0 && changed == 0) {
           firsts[i] = j;
           changed = 1;
      }
    }
    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i];
  }

  for (j = 6; j < 10; j++)
    first_sum = first_sum + firsts[j];

  total_number++;

  first_avg = ((double) first_sum) / 4.0;

  if (first_avg < 1.51)
    return(9) ;
  else return(4) ;
}


int  check5and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int total_number = 0;
```

```
int rejected = 0;
int wrong_ones = 0;
int zone1_ones, zone2_ones;
int pres_bit  total[16], total2[16];
double avg_position[16], avg2_position[16];
int first_sum, last_sum, max;
int first_pos[16], last_pos[16];
double first_avg, last_avg;
int how_many = 0;
int index = 0 , index2 = 0 ;

for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  last_pos[i] = 0;
  total[i] = 0;
  total2[i] = 0;
}

for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0;
}

how_many++;

for (i = 0; i < 16; i++) {
  pres_bit = 0;
  first_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
    index++ ;
    pres_bit = bits[index] ;
    if (pres_bit != 0 && changed == 0) {
        firsts[i] = j;
        changed = 1;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;
        changed2 = 1;
    }
  }

  total[i] = total[i] + firsts[i];
  total2[i] = total2[i] + lasts[i];
}

for (i = 8, max = firsts[i]; i < 13; i++) {
  if (firsts[i] > max)
    max = firsts[i];
```

```
    first_sum = first_sum + firsts[i];
   }

   first_avg = ((double) first_sum) / 5.0;

   first_pos[(int) first_avg]++;

   if (first_avg < 2.0)
     return(6) ;
   else if (first_avg > 4.0)                                          1080
     return(5) ;
   else if (max > 5)
     return(5) ;
   else if (max < 6)
     return(6) ;
   else
     rejected++;
}

                                                                      1090
int  check5and8(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int first_sum, last_sum;
  int first_pos[16], last_pos[16];                                    1100
  double first_avg, last_avg;
  int how_many = 0;
  int total_number = 0;
  int rejected = 0;
  int wrong_ones = 0;
  int index = 0, index2 = 0 ;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;                                                  1110
    total[i] = 0;
    total2[i] = 0;
   }

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
   }

  how_many++;                                                         1120

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    first_sum = 0;
```

```
    last_sum = 0;
    changed = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
      index++ ;
      pres_bit = bits[index];                                          1130
      it (pres_bit != 0 && changed == 0) {
           firsts[i] = j;
           changed = 1;
      }
    }
    for (j = 15; j >= 0; j = j - 1) {
      index2 = i*16 + j ;
      pres_bit = bits[index2] ;
      if (pres_bit != 0 && changed2 == 0) {
           lasts[i] = j;                                               1140
           changed2 = 1;
      }
    }

    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i]; }

  total_number++;

  for (i = 8; i < 12; i++)                                             1150
    first_sum = first_sum + firsts[i];
  first_avg = ((double) first_sum) / 4.0;
  first_pos[(int) first_avg]++;

  for (i = 3; i < 6; i++)
    last_sum = last_sum + lasts[i];

  last_avg = ((double) last_sum) / 3.0;
  last_pos[(int) last_avg]++;
                                                                       1160
  if (last_avg - first_avg < 2.01)
    return(5) ;
  else if (last_avg - first_avg > 5.99)
    return(8) ;
  else if (last_avg < 5.99)
    return(5) ;
  else if (last_avg > 9.01)
    return(8) ;
  else if (first_avg > 5.99)
    return(5) ;                                                        1170
  else if (first_avg < 5.01)
    return(8) ;
  else
    rejected++;
}


int  check5and9(int * bits)
```

```
{
 int actual, *pnumeral, *origout, *origin , matrix[16][16];
 int changed, firsts[16], lasts[16], changed2;
 int *origout2, i, j;
 int zone1_ones, zone2_ones;
 int pres_bit, total[16], total2[16];
 double avg_position[16], avg2_position[16];
 int max, min, sevens, threes;
 int first_sum, last_sum, diff_sum;
 int first_pos[16], last_pos[16];
 double first_avg, last_avg, diff_avg;
 int how_many = 0;
 int wrong_ones = 0;
 int rejected = 0;
 int total_number = 0;
 int index = 0,  i  dex2 = 0 ;

 for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  last_pos[i] = 0;
  total[i] = 0;
  total2[i] = 0;
 }

 for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0; }

 how_many++;
 sevens = 0;
 threes = 0;

 for (i = 0; i < 16; i++) {
  pres_bit = 0;
  last_sum = 0;
  diff_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
   index++ ;
   pres_bit = bits[index] ;;
   if (pres_bit != 0 && changed == 0) {
        firsts[i] = j;
        changed = 1;
   }
  }
  for (j = 15; j >= 0; j = j - 1) {
   index2 = i*16 + j ;
   pres_bit = bits[index2] ;
   if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;
        changed2 = 1;
   }
  }
```

```
    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i]; }

  for (i = 2  max = lasts[i], min = lasts[i]; i < 7; i++) {
    if (lasts[i] > max)
      max = lasts[i];
    if (lasts[i] < min)                                          1240
      min = lasts[i];
    if (lasts[i] < 8)
      sevens++;
    if (lasts[i] < 4)
      threes++;
    last_sum = last_sum + lasts[i];
    diff_sum = diff_sum + (lasts[i] - firsts[i]);
  }

  last_avg = ((double) last_sum) / 5.0;                          1250
  diff_avg = ((double) diff_sum) / 5.0;

  last_pos[(int) last_avg]++;

  if (last_avg < 7.0)
    return(5) ;
  else if (last_avg >= 10.0)
    return(9) ;
  else if (diff_avg < 5.01)
    return(5) ;                                                  1260
  else if (diff_avg > 6.21)
    return(9) ;
  else if (sevens >= 3)
    return(5) ;
  else if (threes >= 2)
    return(5) ;
  else
    return(9) ;
}
                                                                 1270

int  check7and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int ones = 0;
  int first_max, second_max;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];                           1280
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;
```

```
int wrong_ones = 0;
int rejected = 0;
int total_number = 0;
int first_hit_sum;                                                    1290
int index = 0 , index2 = 0, index3 = 0 ;

pnumeral = &matrix[0][0];
origin = pnumeral;

for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  last_pos[i] = 0;
  total[i] = 0;
  total2[i] = 0;                                                      1300
}

first_max = 0;
second_max = 0;

for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0;
}
                                                                      1310
first_hit_sum = 0;
how_many++;

ones = 0;

for (i = 0; i < 16; i++) {
  pres_bit = 0;
  first_sum = 0;
  changed = 0;
  changed2 = 0;                                                       1320
  for (j = 0; j < 16; j++) {
    index++ ;
    pres_bit = bits[index] ;
    if (pres_bit != 0 && changed == 0) {
        firsts[i] = j;
        changed = 1;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;                                               1330
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;
        changed2 = 1;
    }
  }
  if (i == 9 || i == 10 || i == 11) {
    changed = 0;
    for (j = 0; j < (i + 1)  ; j++) {
        index3 = (i-j) * 16 + j ;                                     1340
```

```
            pres_bit = bits[index3] :
            if (pres_bit != 0 && changed == 0) {
              first_hit_sum = first_hit_sum + j;
              changed = 1;
            }
        }
      }

      total[i] = total[i] + firsts[i];
      total2[i] = total2[i] + lasts[i]; }                                    1350

    if (first_hit_sum > 16)
      return(7) ;
    else if (first_hit_sum < 17)
      return(9) ;
    else rejected++;
    total_number++;
}

                                                                             1360
int  check8and9(int * bits)
{
    int actual, *pnumeral, *origout, *origin, matrix[16][16];
    int change_matrix[20];
    int *origout2, i, j, total = 0  ;
    int trials = 0;
    int which_one = 0;
    int second_changes = 0;
    int pres_bit, orig, changes;

                                                                             1370
    for (i = 0; i < 16; i++) {
      orig = 0;
      pres_bit = 0;
      changes = 0;
      for (j = 0; j < 16; j++) {
        total++ ;
        if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;                                                     1380
              if (changes > 2)
                second_changes++;
            }
            orig = pres_bit;
        }
      }
    }
    if (second_changes < 2)
      return(9) ;
    else return(8) ;                                                         1390
}
```

```
/*************************************************/
/*              PROFILE.C                        */
/*-----------------------------------------------*/
/* This function scans the top/bottom profile    */
/* of a blob and returns the coords of the       */
/* min/max point encountered.  The profile is    */
/* only scanned up to a certain deviation        */
/* limit from the center of the blob towards     */
/* either side.                                  */
/*                                               */
/* NOTE:                                         */
/* When a character's contour runs along the     */
/* top or bottom edge of the input bitmap array  */
/* for the entire horizontal scanning region,    */
/* the min/max point returned will exceed the    */
/* the bitmap array's top/bottom boundaries by   */
/* 1 in the vertical direction.  It's horizon-   */
/* tal coordinate will be assigned to the start  */
/* of the scanning region by default.            */
/*************************************************/

#include "prototypes.h"

int FindMax (tCOORD *);
int FindMin (tCOORD *);


tCOORD *Profile(tBITMAP bitmap, tNEW_BLOBTYPE input, VIEW top_or_bot)
{
  tCOORD *output_array;
  int temp_min, temp_max;
  int    bloblen, loop_variable;
  BOOL   inbounds;
  static int row,col,startrow,stoprow, startcol, stopcol,
  (*Forw)(),(*Back)();

  output_array = (tCOORD *)calloc((size_t)PROFILE_SIZE, sizeof(tCOORD));

  startcol = FindExtremePoint(bitmap, input, COL, START);
  stopcol = FindExtremePoint(bitmap, input, COL, END);

  /* set parameters for top or bottom viewpoint */
  if(top_or_bot == TOP) {
    Forw = Down;
    Back = Up;
    startrow = 0;
    stoprow = HEIGHT - 1;
    }
  else
```

```
  {
   Forw = Up;
   Back = Down;
   startrow = HEIGHT - 1;
   stoprow = 0;
  }

/* set column limits of profile scan */
bloblen = stopcol - startcol;
startcol = (int)(startcol + (bloblen * BORDER_DEVIATION));
stopcol = (int)(stopcol - (bloblen * BORDER_DEVIATION));

/* initialize the array to be starting coordinates as min/max */
for (loop_variable = 0; loop_variable < PROFILE_SIZE; loop_variable++)
  {
   output_array[loop_variable].col = startcol;
   output_array[loop_variable].row = startrow;
  }

/* scan profile between limits, updating min/max found */
inbounds = FALSE; /* assume min/max may be top/bottom edges of map */
for(col = startcol; col <= stopcol; ++col)
  {
   row = (*Back)(startrow); /* min/max may be the top/bottom edges of map */
   while((abs(row-stoprow) >= 1) && bitmap[(*Forw)(row)][col])
        row = (*Forw)(row);

   /* flag that min/max is within bitmap array bounds */
   if(row != ((*Back)(startrow)))
        inbounds = TRUE;

   if(top_or_bot == TOP)
        {
         temp_min = find_min(output_array);
         if (row > output_array[temp_min].row)
           {
            output_array[temp_min].row = row;
            output_array[temp_min].col = col;
           }
        }
   else
        {
         temp_max = find_max(output_array);
         if (row < output_array[temp_max].row)
           {
            output_array[temp_max].row = row;
            output_array[temp_max].col = col;
           }
        }
  }
/* check for case of min/max being top or bottom edges */
/* if so, assign min/max to middle of top/bottom row.  */
if(!inbounds)
  {
```

```
    for (loop_variable = 0; loop_variable < PROFILE_SIZE; loop_variable++)
        {
         output_array[loop_variable].row = startrow;
         output_array[loop_variable].col = startcol + ((stopcol-startcol)/2);
        }
   }
 return(output_array);
}


int find_max (tCOORD *input_array)
{
 int max, index, loop_variable;

 max = input_array[0].row;
 for (loop_variable = 1; loop_variable < PROFILE_SIZE; loop_variable++)
   {
    if (max < input_array[loop_variable].row)
        {
         max = input_array[loop_variable].row;
         index = loop_variable;
        }
   }
 return index;
}


int find_min (tCOORD *input_array)
{
 int min, index, loop_variable;

 min = input_array[0].row;
 for (loop_variable = 1; loop_variable < PROFILE_SIZE; loop_variable++)
   {
    if (min > input_array[loop_variable].row)
        {
         min = input_array[loop_variable].row;
         index = loop_variable;
        }
   }
 return index;
}


int Up(int row)
{
 return(--row);
}


int Down(int row)
{
 return(++row);
}
```

```
#include "defines.h"


/***** utils.c ******/
int hsfbits_to_char_array(unsigned char *hsf_data, tBITMAP bitmap,
                          int height, int width);
int fwrite_bitmap(char *, tBITMAP, int, int);
int fwrite_intmap(char *, tINTMAP, int, int);
int get_next_bit(unsigned char *data);
char *uchar2hexstring(unsigned char bits);
void bitmap_to_char_array(unsigned char *, tBITMAP, int, int);


#define INITIALIZE (void *)-1


/***** getblob.c *****/
void GetBlob(tBITMAP, tBITMAP *, int, int);
int GetBlobs(tBITMAP, tBITMAP **);
tNEW_BLOBTYPE ConvertBlobs(tBITMAP);
tBITMAP NewBlob2OldBlob(tBITMAP, tNEW_BLOBTYPE);
int FindBlobWeight(tBITMAP, tNEW_BLOBTYPE);
int FindExtremePoint(tBITMAP, tNEW_BLOBTYPE, ROW_COL, START_END);
float GetAspect(int, int, int, int);

/***** allocation.c *****/
tBITMAP BitmapCreate();
void BitmapDestroy(tBITMAP);
tINTMAP IntmapCreate(int, int);
tNEW_BLOBTYPE NewBlobCreate();

/***** recognize.c *****/
/* BackPropagation Core function prototypes */

char Recognition(tINTMAP, float *);
int InitNet(void);
float Logistic(float);
float *Recognize_Hand(tINTMAP, int);
int FirstPropagate(int x);
int FreeLearn(void);

/* IO functions */
int ReadWeights(char *, int);


/***** segmentation.c *****/
char *Segment(tBITMAP, tNEW_BLOBTYPE, int, int, int *, int);

/***** profile.c *****/
int Up(int);
int Down(int);
tCOORD *Profile(tBITMAP, tNEW_BLOBTYPE, VIEW);
```

```
/***** bridge.c *****/
int BlobDimensionsOK(tBITMAP, tNEW_BLOBTYPE, tNEW_BLOBTYPE);
tNEW_BLOBTYPE *Bridge(tBITMAP, tNEW_BLOBTYPE, tCOORD, tCOORD);

/*** * HDS.c *****/
tNEW_BLOBTYPE *HitNDeflect(tBITMAP, tNEW_BLOBTYPE, tCOORD, tCOORD,
                           DIRECTION);

/***** categorize.c *****/
char Categorize(tBITMAP, tNEW_BLOBTYPE, int, int);

/***** normalize.c *****/
tINTMAP Normalize(tBITMAP, tNEW_BLOBTYPE);

/***** slant.c *****/
int GetIntmapWidth(tINTMAP, int *, int *);
tINTMAP rotate(tINTMAP, double);
tINTMAP Slant (tINTMAP);

/***** thinning.c *****/
tINTMAP Thinning (tINTMAP);

/***** thicken.c *****/
tINTMAP Thicken (tINTMAP);

/***** postproc.c *****/
int PostProcess(tINTMAP, float*);
int check8and9(int *) ;
int check3and5(int *) ;
int check2and6(int *) ;
int check3and8(int *) ;
int check5and8(int *) ;
int check5and9(int *) ;
int check5and6(int *) ;
int check4and9(int *) ;
int check7and9(int *) ;
int check1and6(int *) ;
int check3and9(int *) ;
int check3and6(int *) ;
int check0and1(int *) ;
int check0and2(int *) ;
int check0and3(int *) ;
int check0and4(int *) ;
int check0and5(int *) ;
int check0and6(int *) ;
int check0and7(int *) ;
int check0and9(int *) ;
int check1and8(int *) ;
int check1and9(int *) ;
```

```
#include "prototypes.h"
```

```
char Recognition(tINTMAP intmap, float *percentage)
{
  int output;
  float *percents;
  int max_index, max_index2;
  int count;
  int temp;                                                          10

  max_index = 0;
  max_index2 = 0;

  percents = (float *)calloc((size_t)10, sizeof(float));
  percents = Recognize_Hand(intmap, 0);

  for (count=0; count<10; count++)
    if (percents[count] > percents[max_index])
      max_index = count;                                             20
    else
      if (percents[count] > percents[max_index2])
          max_index2 = count;

  if ((percents[max_index] < 0.7) || ((percents[max_index] > 0.7) &&
                                      (percents[max_index2] > 0.25))) {
    temp = PostProcess(intmap, percents);
    if (temp != max_index)
      return('*');
    else                                                             30
      if (percents[max_index] < 0.4)
          return('*');
  }
  *percentage = percents[max_index];
  return((char)(max_index+48));
}



float *Recognize_Hand(tINTMAP map, int NET)                          40
{
  register int i, j;
  int cnt = 0;
  char szInfo[30];

  /* pjliu: added result variable */
  float *result;

  result = (float *)malloc((size_t)10 * sizeof(float));
                                                                     50
  /* fill the inputNodes */
  for (i = 0; i < 16; ++i) {
    for (j= 0; j < 16; ++j) {
      iNode[cnt++] = map[i][j];
    }
  }
```

```
  if (NET == 0)
    FirstPropagate(0);
  else
    FirstPropagate(1);

  for(i=0;i<10;i++)
    result[i] = oNode[i];

  return result;
}


/* weight file format is outlined in backprop.doc */
int ReadWeights(char *wInfileName, int x)
{
  static char szAppName[] = "PROTO2";
  char szInfo[30];
  float temp;
  float wt[nOutputNodes][nHiddenNodes];
  float bias[nOutputNodes];
  float wts[nOutputNodes][nHiddenNodes];
  float biass[nOutputNodes];
  FILE *wInfile;
  short y, z;

  if ((wInfile = fopen(wInfileName, "r")) == NULL)
    return 1;
  {
    /* current implementation has only 1 Hidden layer */
    register int i, j;

    for (i = 0; i < nHiddenNodes; ++i)
      {
          fscanf(wInfile, "%f", &temp);
          if (x)
            bias3[i] = temp;
          else
            bias1[i] = temp;
      }


    for (i = 0; i < nHiddenNodes; ++i)
      {
          for (j = 0; j < nInputNodes; ++j)
            {
              fscanf(wInfile, "%f", &temp);
              if (x)
                wt3[i][j] = temp;
              else
                wt1[i][j] = temp;
            }

      }
```

```
    for (i = 0; i < nOutputNodes; ++i)
     {
          fscanf(wInfile, "%f", &temp);
          if (x)
            bias4[i] = temp;
          else
            bias2[i] = temp;
     }
                                                                          120
    for (i = 0; i < nOutputNodes; ++i)
     {
          for (j = 0; j < nHiddenNodes; ++j)
            {
              fscanf(wInfile, "%f", &temp);
              if (x)
                wt4[i][j] = temp;
              else
                wt2[i][j] = temp;
            }                                                             130
     }

  }

  fclose(wInfile);
  return 0;
}

/*-----------------------------------------*/
/* BackPropagation Routines are below here */                             140

float Logistic(float num)
{
  if (num > 10.0)
    return(0.99999);
  else if (num < -10.0)
    return(0.00001);
  else
    return(1.0 / (1.0 + (float) exp( (double) ((-1.0) * num))));
}                                                                         150


int FirstPropagate(int x)
{
  register int i, j;
  float wt[nOutputNodes][nHiddenNodes];
  float bias[nOutputNodes];
  float wts[nOutputNodes][nHiddenNodes];
  float biass[nOutputNodes];
  short y, z;                                                             160

  /* forward propagation */
  for (i = 0; i < nHiddenNodes; ++i)
    {
```

```
    if (x)
        inputHNode[i] = bias3[i];
    else
        inputHNode[i] = bias1[i];

    for (j = 0; j < nInputNodes; ++j)                                    170
        {
         if (x)
           inputHNode[i] += wt3[i][j] * iNode[j];
         else
           inputHNode[i] += wt1[i][j] * iNode[j];
        }

    /* inputHNode[i] /= nInputNodes + 1; */

    hNode[i] = (float) Logistic(inputHNode[i]);                          180
  }

 for (i = 0; i < nOutputNodes; ++i)
  {
   if (x)
        inputONode[i] = bias4[i];
   else
        inputONode[i] = bias2[i];

   for (j = 0; j < nHiddenNodes; ++j)                                    190
        {
         if (x)
           inputONode[i] += wt4[i][j] * hNode[j];
         else
           inputONode[i] += wt2[i][j] * hNode[j];
        }

   /* inputONode[i] /= nHiddenNodes + 1; */

   oNode[i] = (float) Logistic(inputONode[i]);                           200
  }

 return 0;
}
```

```
                                                                         210
```

```
#include "prototypes.h"

char *Segment(tBITMAP bitmap, tNEW_BLOBTYPE input, int height_range,
```

```
            int midline, int *num_chars, int blob_num)
{
 int count, count1, count2, min_count, max_count;
 float aspect; /* is being static necessary? */
 tCOORD *min, *max;
 DIRECTION dir;                                                          10
 char *result;
 char *candidate;
 float *percentage;
 float *temp_percent;
 int startr, endr, startc, endc;
 int num_segs = 1;
 int num_digits;
 float average, max_average;
 tINTMAP intmap;
 tINTMAP normalized_intmap, slanted_intmap, thinned_intmap, thickened_intmap;   20
 tNEW_BLOBTYPE *output;
 char filename[100];
 tBITMAP converted_bitmap;

 /* initialize */
 max_average = 0.0;
 result = (char *)calloc((size_t)3, sizeof(char));
 percentage = (float *)calloc((size_t)2, sizeof(float));
 candidate = (char *)calloc((size_t)2, sizeof(char));
 temp_percent = (float *)calloc((size_t)2, sizeof(float));               30

 startr = FindExtremePoint(bitmap, input, ROW, START);
 endr = FindExtremePoint(bitmap, input, ROW, END);
 startc = FindExtremePoint(bitmap, input, COL, START);
 endc = FindExtremePoint(bitmap, input, COL, END);

 aspect = GetAspect(startc, endc, startr, endr);
 if ((aspect < ASPECT_CUTOFF) && (aspect != INFINITE)) {
  min = (tCOORD *)malloc((size_t)PROFILE_SIZE * sizeof(tCOORD));
  max = (tCOORD *)malloc((size_t)PROFILE_SIZE * sizeof(tCOORD));         40
  min = Profile(bitmap, input, TOP);
  max = Profile(bitmap, input, BOTTOM);
  for (min_count=0; min_count<PROFILE_SIZE; min_count++) {
   for (max_count=0; max_count<PROFILE_SIZE; max_count++) {
        output = Bridge(bitmap, input, min[min_count], max[max_count]);
        if (output == NULL) {
          ((HEIGHT-max[max_count].row) >= min[min_count].row)
            ? (dir = UP) : (dir = DOWN);
          output = HitNDeflect(bitmap,input,min[min_count],max[max_count],
                               dir);                                     50
        }
        if (output != NULL) {
          num_digits = 0;
          *num_chars = 2;
          average = 0.0;
          for (count=0; count<2; count++) { /* at most 2 segments in a blob */
            converted_bitmap = NewBlob2OldBlob(bitmap, output[count]);
            candidate[count] = Categorize(bitmap, output[count], height_range,
```

```
                                        midline);
            if (candidate[count] == 'D') {                                    60
              num_digits++;

              normalized_intmap = Normalize(bitmap_output[count]);
              sprintf(filename, "normal-%d%dbmp", blob_num, count);
              fwrite_intmap(filename, normalized_intmap, INT_HEIGHT,INT_WIDTH);

              slanted_intmap = Slant(normalized_intmap);
              sprintf(filename, "slant-%d%dbmp", blob_num, count);
              fwrite_intmap(filename, slanted_intmap, INT_HEIGHT, INT_WIDTH);
                                                                              70
              thinned_intmap = Thinning(slanted_intmap);
              sprintf(filename, "thin-%d%dbmp", blob_num, count);
              fwrite_intmap(filename, thinned_intmap, INT_HEIGHT, INT_WIDTH);

              thickened_intmap = Thicken(thinned_intmap);
              sprintf(filename, "thicken-%d%dbmp", blob_num, count);
              fwrite_intmap(filename, thickened_intmap, INT_HEIGHT, INT_WIDTH);

              candidate[count] = Recognition(thickened_intmap,
                                             &temp_percent[count]);           80
              average += (temp_percent[count]*temp_percent[count]);
            }
          }
          if (num_digits != 0)
            average = average/(float)num_digits;
          if (average >= max_average) {
            result = candidate;
            percentage = temp_percent;
          }
        }                                                                     90
    }
  }
}
else {
  *num_chars = 1;
  result[0] = Categorize(bitmap, input, height_range, midline);
  if (result[0] == 'D') {
    *num_chars = 1;

    normalized_intmap = Normalize(bitmap, input);                             100
    sprintf(filename, "normal-%dbmp", blob_num);
    fwrite_intmap(filename, normalized_intmap, INT_HEIGHT, INT_WIDTH);

    slanted_intmap = Slant(normalized_intmap);
    sprintf(filename, "slant-%dbmp", blob_num);
    fwrite_intmap(filename, slanted_intmap, INT_HEIGHT, INT_WIDTH);

    thinned_intmap = Thinning(normalized_intmap);
    sprintf(filename, "thin-%dbmp", blob_num);
    fwrite_intmap(filename, thinned_intmap, INT_HEIGHT, INT_WIDTH);           110

    thickened_intmap = Thicken(thinned_intmap);
```

```
    sprintf(filename, "thicken-%dbmp", blob_num);
    fwrite_intmap(filename, thickened_intmap, INT_HEIGHT, INT_WIDTH);

    result[0] = Recognition(thickened_intmap, &percentage[0]);
   }
  }
  result[*num_chars] = 0;
  return result;                                                    120
}
```

```
/*************************************************************************
 * This file contains the slant correction algorithm originally written  *
 * by Nagi.  The algorithm rotates individual bits through several angles *
 * in a horizontal plane.  The slant corrected image chosen by finding the *
 * rotated angle at which the image width is smallest                    *
 *************************************************************************/
#include "prototypes.h"


int GetIntmapWidth (tINTMAP input, int *min, int *max)              10
{
  int count1, count2;

  *min = SLANTED_WIDTH;
  *max = 0;
  for (count1=0; count1<SLANTED_HEIGHT; count1++)
    for (count2=0; count2<SLANTED_WIDTH; count2++) {
      if (input[count1][count2]) {
          if (count2 < *min)
            *min = count2;                                          20
          if (count2 > *max)
            *max = count2;
      }
    }
  return (*max-*min);
}


tINTMAP rotate(tINTMAP input, double angle)
{                                                                   30
  int count1, count2;
  tINTMAP output;
  int rotated_x, rotated_y;

  output = IntmapCreate(SLANTED_HEIGHT, SLANTED_WIDTH);

  for (count1=0; count1<SLANTED_HEIGHT; count1++)
    for (count2=0; count2<SLANTED_WIDTH; count2++) {
      if (input[count1][count2]) {
          rotated_x = count1;                                       40
          rotated_y = (int)(((double)count2) - count1*tan(-angle));
```

```
            output[rotated_x][rotated_y] = 1;
        }
      }
    return output;
}


tINTMAP Slant(tINTMAP input)
{                                                                    50
  int count1, count2;
  tINTMAP output;
  tINTMAP working_intmap, temp_intmap;
  double angle_count, best_angle;
  int min_width, temp_width;
  int startc, stopc;
  int copy_counter, start_copy;

  output = IntmapCreate(INT_HEIGHT, INT_WIDTH);
  working_intmap = IntmapCreate(SLANTED_HEIGHT, SLANTED_WIDTH);      60

  for (count1=0; count1<SLANTED_HEIGHT; count1++)
    for (count2=28; count2<SLANTED_WIDTH; ++count2)
      if (input[count1][count2-28])
            working_intmap[count1][count2] = 1;

  best_angle = 0.0;
  min_width = SLANTED_WIDTH;

  for (angle_count=-0.8; angle_count<=0.9; angle_count+=0.1) {       70
    temp_intmap = rotate(working_intmap, angle_count);
    temp_width = GetIntmapWidth(temp_intmap, &startc, &stopc);
    if (temp_width < min_width) {
      min_width = temp_width;
      best_angle = angle_count;
    }
    for (count1=0; count1<SLANTED_HEIGHT; count1++)
      for (count2=0; count2<SLANTED_WIDTH; count2++)
            temp_intmap[count1][count2] = 0;
  }                                                                  80
  temp_intmap = rotate(working_intmap, best_angle);
  min_width = GetIntmapWidth(temp_intmap, &startc, &stopc);
  start_copy = copy_counter = (int) (stopc + startc + 1) / 2 - 8;
  for (count1=0; count1<INT_HEIGHT; count1++, copy_counter=start_copy)
    for (count2=0; count2<INT_WIDTH; count2++, copy_counter++)
      if (temp_intmap[count1][copy_counter])
            output[count1][count2] = 1;
  return output;
}
```

```
#include "prototypes.h"

struct array_place{
  int x;
```

```
  int y;
};

struct array_place cell[8] = {
  -1, -1,
  -1, 0,                                                          10
  0, -1,
  0, 0
  }; /* These are the relative displacements from the eight directions */


tINTMAP Thicken (tINTMAP intmap)
{
  tINTMAP input, temp_data, output;
  int count, count1, count2;
                                                                  20
  input = IntmapCreate(PADDED_X, PADDED_Y);
  temp_data = IntmapCreate(PADDED_X, PADDED_Y);
  output = IntmapCreate(INT_HEIGHT, INT_WIDTH);

  for (count1=1; count1<(PADDED_X-1); count1++)
    for (count2=1; count2<(PADDED_Y-1); count2++)
      input[count1][count2]=intmap[count1-1][count2-1];

  for (count1=1; count1<(PADDED_X-1); ++count1)
    for (count2=1; count2<(PADDED_Y-1); ++count2)                 30
      if (input[count1][count2])
          for (count=0; count<4; ++count)
            temp_data[count1+cell[count].x][count2+cell[count].y] = 1;

  for (count1=1; count1<(PADDED_X-1); count1++)
    for (count2=1; count2<(PADDED_Y-1); count2++)
      output[count1-1][count2-1]=temp_data[count1][count2];

  return output;
}                                                                 40
```

```
#include "prototypes.h"

int IsBorder (tINTMAP, int, int);
int RightConditions (tINTMAP, int, int);


struct thinning_array_place
{
  int x;
  int y;                                                          10
};

struct thinning_array_place thinning_cell[8] = {
  -1,  0,
  -1, +1,
  0, +1,
```

```
  +1, +1,
  +1,  0,
  +1, −1,
  0, −1,
  −1, −1
  };


int IsBorder (tINTMAP input, int i, int j)
{
  int count, temp_x, temp_y;

  count = 0;
  if (input[i][j])
    for (count=0; count<8; count++) {
      temp_x = i + thinning_cell[count].x;
      temp_y = j + thinning_cell[count].y;
      if (!input[temp_x][temp_y])
          return 1;
      else ++count;
    }
  return 0;
}


int RightConditions (tINTMAP input, int i, int j)
{
  int count, temp_x, temp_y, cond1, cond2, cond3;
  int u, v, color, p, w_b, t1, t2;

  /* initialize */
  count = 0;
  cond1 = 0;
  cond2 = 0;
  cond3 = 0;

  for (u = 0; u <= 7; ++u) {
    temp_x = thinning_cell[u].x + i;
    temp_y = thinning_cell[u].y + j;
    if (input[temp_x][temp_y])
      ++count;
  }
  if ((count > 1) && (count < 7))
    cond1 = 1;

  w_b = 0;
  color = input[i−1][j];
  for (u = 1; u <= 8; ++u) {
    if (u == 8)
      v = 0;
    else v = u;
    temp_x = thinning_cell[v].x + i;
    temp_y = thinning_cell[v].y + j;
    if (!input[temp_x][temp_y]) {
```

```
      if (color) {
           ++ w_b;
           color = 0;
      }
     }
     else color = 1;
   }
   if (w_b == 1)
     cond2 = 1;
   t1 = input[i-1][j]+input[i-1][j+1]+input[i][j+1]+input[i+1][j-1];                    80
   t2 = input[i][j+1]+input[i+1][j+1]+input[i+1][j]+input[i-1][j-1];
   if (((! t1) && input[i+1][j]  && input[i][j-1]) ||
      ((! t2) && input[i][j-1] && input[i-1][j]))
     cond3 = 1;
   if (cond1 && (cond2 || cond3))
     return 1;
   else return 0;
}


                                                                                       90
tINTMAP Thinning (tINTMAP intmap)
{
   tINTMAP input, temp_data, output;
   int count1, count2;
   int more, g_switch;

   input = IntmapCreate(PADDED_X, PADDED_Y);
   temp_data = IntmapCreate(PADDED_X, PADDED_Y);
   output = IntmapCreate(INT_HEIGHT, INT_WIDTH);
                                                                                      100
   for (count1=1; count1<(PADDED_X-1); count1++)
     for (count2=1; count2<(PADDED_Y-1); count2++)
       input[count1][count2]=intmap[count1-1][count2-1];

   more = 1;
   g_switch = 1;

   while (more) {
     g_switch = !g_switch;
     more = 0;                                                                        110
     for (count1=0; count1<PADDED_X; ++count1)
       for (count2=0; count2<PADDED_Y; ++count2)
           if (IsBorder(input, count1, count2)) {
             if (RightConditions(input, count1, count2)) {
               if (g_switch) {
                 if (!((input[count1-1][count2] + input[count1][count2-1])
                         * input[count1][count2+1] * input[count1+1][count2])) {
                     temp_data[count1][count2] = 0;
                     more = 1;
                 }                                                                    120
                 else temp_data[count1][count2] = 1;
               }
               else {
                 if (!((input[count1][count2+1] + input[count1+1][count2])
```

```
                    * input[count1-1][count2] * input[count1][count2-1])) {
                temp_data[count1][count2] = 0;
                more = 1;
              }
              else temp_data[count1][count2] = 1;
            }
          }
          else temp_data[count1][count2] = input[count1][count2];
        }
        else temp_data[count1][count2] = input[count1][count2];

    for (count1=1; count1<PADDED_X; count1++)
      for (count2=1; count2<PADDED_Y; count2++)
          input[count1][count2]=temp_data[count1][count2];
  }
  for (count1=1; count1<(PADDED_X-1); count1++)
    for (count2=1; count2<(PADDED_Y-1); count2++)
      output[count1-1][count2-1]=input[count1][count2];
  return output;
}
```

```
#include "prototypes.h"

void bitmap_to_char_array(unsigned char *xbitmap, tBITMAP bitmap,
                          int height, int width)
{
  int count, count2;

  unsigned char bitmap_mask[] = {0x01, 0x02, 0x04, 0x08,
                                 0x10, 0x20, 0x40, 0x80};

  for(count=0; count<height; count++) {
    for(count2=0;count2<width;count2++) {
      bitmap[count][count2] =
          xbitmap[count*((width+8-width%8)/8)+((count2-(count2%8))/8)] &
            bitmap_mask[count2%8];
    }
  }
}


int hsfbits_to_char_array(unsigned char *hsf_data, tBITMAP bitmap,
                          int height, int width)
{
  int count1, count2;

  get_next_bit(INITIALIZE);
  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<width;count2++) {
      if(get_next_bit(hsf_data))
          bitmap[count1][count2] = (char)1;
      else bitmap[count1][count2] = (char)0;
```

```
    }
  }
}

int fwrite_bitmap(char *filename, tBITMAP bitmap, int height, int width)
{

  int count1, count2, count3;
  FILE *outfile;                                                          40
  unsigned char outval;
  unsigned char bitmap_mask[] = {0x01, 0x02, 0x04, 0x08,
                                 0x10, 0x20, 0x40, 0x80};

  int div_eight;

  if( (outfile = fopen(filename, "w")) == NULL) {
    printf("fwrite_bitmap:    Couldn't oper file %s for writing\n", filename);
    exit(0);
  }                                                                       50

  fprintf(outfile, "#define %s_width %d\n", filename, width);
  fprintf(outfile, "#define %s_height %d\n", filename, height);
  fprintf(outfile, "static unsigned char %s_bits[] = {\n", filename);

  div_eight = (int)(ceil((double)((double)width/(double)8)));
  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<div_eight;count2++) {
      outval = 0;
      for(count3=0;count3<8;count3++) {                                   60
          if( (count2*8+count3) < width) {
            if ( bitmap[count1][count2*8+count3] ) {
              outval += bitmap_mask[count3];
            }
          }
      }
      if(count1==(height-1) && count2==(div_eight-1)) {
          fprintf(outfile, "%s};\n", uchar2hexstring(outval));
      }
      else {                                                              70
          fprintf(outfile, "%s,\n", uchar2hexstring(outval));
      }
    }
  }
  fclose(outfile);
}


int fwrite_newblob(char *filename, tBITMAP checkmap, tNEW_BLOBTYPE blob,
                   int height, int width)                                 80
{
  int count, count2;

  tBITMAP retval = BitmapCreate();
```

```
  for(count=0;count<height;count++) {
    for(count2=blob[count].start_blob;count2<=blob[count].end_blob;count2++) {
      retval[count][count2] = checkmap[count][count2];
    }
  }

  fwrite_bitmap(filename, retval, height, width);

  BitmapDestroy(retval);
}



int fwrite_intmap(char *filename, tINTMAP intmap, int height, int width)
{
  int count1, count2, count3;
  FILE *outfile;
  unsigned char outval;
  unsigned char intmap_mask[] = {0x01, 0x02, 0x04, 0x08,
                                 0x10, 0x20, 0x40, 0x80};


  if( (outfile = fopen(filename, "w")) == NULL) {
    printf("fwrite_intmap:    Couldn't open file %s for writing\n", filename);
    exit(0);
  }

  fprintf(outfile, "#define %s_width %d\n", filename, width);
  fprintf(outfile, "#define %s_height %d\n", filename, height);
  fprintf(outfile, "static unsigned char %s_bits[] = {\n", filename);

  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<((int)ceil((double)(width/8)));count2++) {
      outval = 0;
      for(count3=0;count3<8;count3++) {
          if( (count2*8+count3) < 16) {
            if ( intmap[count1][count2*8+count3] ) {
              outval += intmap_mask[count3];
            }
          }
      }
      if(count1==(height-1) && count2==((int)ceil((double)(width/8))-1)) {
          fprintf(outfile, "%s};\n", uchar2hexstring(outval));
      }
      else {
          fprintf(outfile, "%s,\n", uchar2hexstring(outval));
      }
    }
  }
  fclose(outfile);
}


int get_next_bit(unsigned char *data)
```

```
{
  static int byte_offset=-1;
  static int bit_offset=-1;

  unsigned char bitmap_mask[] = {0x80, 0x40, 0x20, 0x10,
                                 0x08, 0x04, 0x02, 0x01};

  if( data == (void *)-1 ) {
    byte_offset=-1;
    bit_offset=-1;
    return 0;
  }
  bit_offset++;
  if(bit_offset==8)
    bit_offset=0;
  if(bit_offset==0)
    byte_offset++;

  return (data[byte_offset]&bitmap_mask[bit_offset]);
}

char *uchar2hexstring(unsigned char bits) {
  switch (bits)
    {
    case 0:
      return "0x00";
    case 1:
      return "0x01";
    case 2:
      return "0x02";
    case 3:
      return "0x03";
    case 4:
      return "0x04";
    case 5:
      return "0x05";
    case 6:
      return "0x06";
    case 7:
      return "0x07";
    case 8:
      return "0x08";
    case 9:
      return "0x09";
    case 10:
      return "0x0a";
    case 11:
      return "0x0b";
    case 12:
      return "0x0c";
    case 13:
      return "0x0d";
    case 14:
```

```
  return "0x0e";
case 15:
  return "0x0f";
case 16:
  return "0x10";
case 17:
  return "0x11";
case 18:
  return "0x12";
case 19:
  return "0x13";
case 20:
  return "0x14";
case 21:
  return "0x15";
case 22:
  return "0x16";
case 23:
  return "0x17";
case 24:
  return "0x18";
case 25:
  return "0x19";
case 26:
  return "0x1a";
case 27:
  return "0x1b";
case 28:
  return "0x1c";
case 29:
  return "0x1d";
case 30:
  return "0x1e";
case 31:
  return "0x1f";
case 32:
  return "0x20";
case 33:
  return "0x21";
case 34:
  return "0x22";
case 35:
  return "0x23";
case 36:
  return "0x24";
case 37:
  return "0x25";
case 38:
  return "0x26";
case 39:
  return "0x27";
case 40:
  return "0x28";
case 41:
```

```
  return "0x29";
case 42:
  return "0x2a";
case 43:
  return "0x2b";
case 44:
  return "0x2c";
case 45:
  return "0x2d";
case 46:
  return "0x2e";
case 47:
  return "0x2f";
case 48:
  return "0x30";
case 49:
  return "0x31";
case 50:
  return "0x32";
case 51:
  return "0x33";
case 52:
  return "0x34";
case 53:
  return "0x35";
case 54:
  return "0x36";
case 55:
  return "0x37";
case 56:
  return "0x38";
case 57:
  return "0x39";
case 58:
  return "0x3a";
case 59:
  return "0x3b";
case 60:
  return "0x3c";
case 61:
  return "0x3d";
case 62:
  return "0x3e";
case 63:
  return "0x3f";
case 64:
  return "0x40";
case 65:
  return "0x41";
case 66:
  return "0x42";
case 67:
  return "0x43";
case 68:
```

```
  return "0x44";
 case 69:
  return "0x45";
 case 70:
  return "0x46";
 case 71:
  return "0x47";
 case 72:
  return "0x48";
 case 73:
  return "0x49";
 case 74:
  return "0x4a";
 case 75:
  return "0x4b";
 case 76:
  return "0x4c";
 case 77:
  return "0x4d";
 case 78:
  return "0x4e";
 case 79:
  return "0x4f";
 case 80:
  return "0x50";
 case 81:
  return "0x51";
 case 82:
  return "0x52";
 case 83:
  return "0x53";
 case 84:
  return "0x54";
 case 85:
  return "0x55";
 case 86:
  return "0x56";
 case 87:
  return "0x57";
 case 88:
  return "0x58";
 case 89:
  return "0x59";
 case 90:
  return "0x5a";
 case 91:
  return "0x5b";
 case 92:
  return "0x5c";
 case 93:
  return "0x5d";
 case 94:
  return "0x5e";
 case 95:
```

```
  return "0x5f";
 case 96:
  return "0x60";
 case 97:
  return "0x61";
 case 98:
  return "0x62";
 case 99:
  return "0x63";
 case 100:
  return "0x64";
 case 101:
  return "0x65";
 case 102:
  return "0x66";
 case 103:
  return "0x67";
 case 104:
  return "0x68";
 case 105:
  return "0x69";
 case 106:
  return "0x6a";
 case 107:
  return "0x6b";
 case 108:
  return "0x6c";
 case 109:
  return "0x6d";
 case 110:
  return "0x6e";
 case 111:
  return "0x6f";
 case 112:
  return "0x70";
 case 113:
  return "0x71";
 case 114:
  return "0x72";
 case 115:
  return "0x73";
 case 116:
  return "0x74";
 case 117:
  return "0x75";
 case 118:
  return "0x76";
 case 119:
  return "0x77";
 case 120:
  return "0x78";
 case 121:
  return "0x79";
 case 122:
```

```
  return "0x7a";
 case 123:
  return "0x7b";
 case 124:
  return "0x7c";
 case 125:
  return "0x7d";
 case 126:
  return "0x7e";
 case 127:
  return "0x7f";
 case 128:
  return "0x80";
 case 129:
  return "0x81";
 case 130:
  return "0x82";
 case 131:
  return "0x83";
 case 132:
  return "0x84";
 case 133:
  return "0x85";
 case 134:
  return "0x86";
 case 135:
  return "0x87";
 case 136:
  return "0x88";
 case 137:
  return "0x89";
 case 138:
  return "0x8a";
 case 139:
  return "0x8b";
 case 140:
  return "0x8c";
 case 141:
  return "0x8d";
 case 142:
  return "0x8e";
 case 143:
  return "0x8f";
 case 144:
  return "0x90";
 case 145:
  return "0x91";
 case 146:
  return "0x92";
 case 147:
  return "0x93";
 case 148:
  return "0x94";
 case 149:
```

```
  return "0x95";
case 150:
  return "0x96";
case 151:
  return "0x97";
case 152:
  return "0x98";
case 153:
  return "0x99";
case 154:
  return "0x9a";
case 155:
  return "0x9b";
case 156:
  return "0x9c";
case 157:
  return "0x9d";
case 158:
  return "0x9e";
case 159:
  return "0x9f";
case 160:
  return "0xa0";
case 161:
  return "0xa1";
case 162:
  return "0xa2";
case 163:
  return "0xa3";
case 164:
  return "0xa4";
case 165:
  return "0xa5";
case 166:
  return "0xa6";
case 167:
  return "0xa7";
case 168:
  return "0xa8";
case 169:
  return "0xa9";
case 170:
  return "0xaa";
case 171:
  return "0xab";
case 172:
  return "0xac";
case 173:
  return "0xad";
case 174:
  return "0xae";
case 175:
  return "0xaf";
case 176:
```

```
    return "0xb0";
  case 177:
    return "0xb1";
  case 178:
    return "0xb2";
  case 179:
    return "0xb3";
  case 180:
    return "0xb4";
  case 181:
    return "0xb5";
  case 182:
    return "0xb6";
  case 183:
    return "0xb7";
  case 184:
    return "0xb8";
  case 185:
    return "0xb9";
  case 186:
    return "0xba";
  case 187:
    return "0xbb";
  case 188:
    return "0xbc";
  case 189:
    return "0xbd";
  case 190:
    return "0xbe";
  case 191:
    return "0xbf";
  case 192:
    return "0xc0";
  case 193:
    return "0xc1";
  case 194:
    return "0xc2";
  case 195:
    return "0xc3";
  case 196:
    return "0xc4";
  case 197:
    return "0xc5";
  case 198:
    return "0xc6";
  case 199:
    return "0xc7";
  case 200:
    return "0xc8";
  case 201:
    return "0xc9";
  case 202:
    return "0xca";
  case 203:
```

```
  return "0xcb";
case 204:
  return "0xcc";
case 205:
  return "0xcd";
case 206:
  return "0xce";
case 207:
  return "0xcf";
case 208:
  return "0xd0";
case 209:
  return "0xd1";
case 210:
  return "0xd2";
case 211:
  return "0xd3";
case 212:
  return "0xd4";
case 213:
  return "0xd5";
case 214:
  return "0xd6";
case 215:
  return "0xd7";
case 216:
  return "0xd8";
case 217:
  return "0xd9";
case 218:
  return "0xda";
case 219:
  return "0xdb";
case 220:
  return "0xdc";
case 221:
  return "0xdd";
case 222:
  return "0xde";
case 223:
  return "0xdf";
case 224:
  return "0xe0";
case 225:
  return "0xe1";
case 226:
  return "0xe2";
case 227:
  return "0xe3";
case 228:
  return "0xe4";
case 229:
  return "0xe5";
case 230:
```

```
      return "0xe6";
    case 231:
      return "0xe7";
    case 232:
      return "0xe8";
    case 233:
      return "0xe9";
    case 234:
      return "0xea";
    case 235:
      return "0xeb";
    case 236:
      return "0xec";
    case 237:
      return "0xed";
    case 238:
      return "0xee";
    case 239:
      return "0xef";
    case 240:
      return "0xf0";
    case 241:
      return "0xf1";
    case 242:
      return "0xf2";
    case 243:
      return "0xf3";
    case 244:
      return "0xf4";
    case 245:
      return "0xf5";
    case 246:
      return "0xf6";
    case 247:
      return "0xf7";
    case 248:
      return "0xf8";
    case 249:
      return "0xf9";
    case 250:
      return "0xfa";
    case 251:
      return "0xfb";
    case 252:
      return "0xfc";
    case 253:
      return "0xfd";
    case 254:
      return "0xfe";
    case 255:
      return "0xff";
    }
}
```

Appendix C

NIST Source Code Listings

```
#include "prototypes.h"


tBITMAP BitmapCreate(int height, int width)
{
  int count;
  tBITMAP temp;

  temp = (tBITMAP)calloc(height, sizeof(tBITMAP_ROW));
  for(count=0;count<height;count++) {                                    10
    temp[count] = (tBITMAP_ROW)calloc(width, sizeof(char));
  }
  return temp;
}

tNEW_BLOBTYPE NewBlobCreate(int height)
{
  return (tNEW_BLOBTYPE)calloc((size_t)height, sizeof(struct tBOUNDARY));
}
                                                                         20
tINTMAP IntmapCreate(int height, int width)
{
  int count, count2;
  tINTMAP temp;
  temp = (tINTMAP)calloc(height, sizeof(tINTMAP_ROW));
  for(count=0;count<height;count++) {
    temp[count] = (tINTMAP_ROW)calloc(width, sizeof(int));
  }
  return temp;
}                                                                        30

void IntmapDestroy(tINTMAP intmap, int height)
{
  int count;
  for (count=0; count<height; count++)
    free(intmap[count]);
  free(intmap);
}
```

```
void NewBlobDestroy(tNEW_BLOBTYPE blob)
{
 free(blob);
}

void BitmapDestroy(tBITMAP bitmap, int height)
{
 int count;
 for (count=0; count<height; count++)
  free(bitmap[count]);
 free(bitmap);
}
```

```
#include "prototypes.h"

int FindBox(tBITMAP bitmap, tNEW_BLOBTYPE *blobs, int valid_blobs,
                        int height)
{
 int count;
 int max_weight=0;
 int max_blob;
 int temp_weight;

 for (count=0; count<valid_blobs; count++) {
  temp_weight = FindBlobWeight(bitmap, blobs[count], height);
  if (temp_weight > max_weight) {
   max_weight = temp_weight;
   max_blob = count;
  }
 }
 return(max_blob);
}

int FindIntmapWeight(tINTMAP intmap, int height, int width)
{
 int count1, count2;
 int weight=0;

 for (count1=0; count1<height; count1++)
  for (count2=0; count2<width; count2++)
   if (intmap[count1][count2])
        weight++;
 return weight;
}

int FindBlobWeight(tBITMAP bitmap, tNEW_BLOBTYPE blob, int height)
{
 int temp_weight = 0;
 int count1, count2;

 for (count1=0; count1<height; count1++)
```

```
    for (count2=blob[count1].start_blob;count2<=blob[count1].end_blob;
        count2++)
     if (bitmap[count1][count2])
       temp_weight++;
  return temp_weight;
}

/* this is the same as FindExtremePoint, except that it takes in different */
/* parameters */
int ValidBlob(tBITMAP bitmap, tNEW_BLOBTYPE blob, int height, int width,
              ROW_COL row_col, START_END start_end)
{
  int count1, count2;
  int min, max;

  max = 0;
  min = width;
  for (count1=0; count1<height; count1++)
    for (count2=blob[count1].start_blob; count2<blob[count1].end_blob;
         count2++)
      if (bitmap[count1][count2]) {
        if (row_col == COL) {
          if (start_end == START) {
            if (min > count2) {
              min = count2;
            }
          }
          else
            if (max < count2) {
              max = count2;
            }
        }
        else
          if (start_end == START) {
            if (min > count1) {
              min = count1;
            }
          }
          else
            if (max < count1) {
              max = count1;
            }
      }
  if (start_end == START)
    return min;
  else return max;
}

int FindExtremePoint(tBITMAP bitmap, int height, int width, ROW_COL row_col,
                     START_END start_end)
{
  int count1, count2;
  int min, max;
```

```
  max = 0;
  min = width-1;
  for (count1=0; count1<height; count1++)
    for (count2=0; count2<width; count2++)
      if (bitmap[count1][count2]) {
        if (row_col == COL) {
          if (start_end == START) {
            if (min > count2) {
              min = count2;
            }
          }
          else
            if (max < count2) {
              max = count2;
            }
        }
        else
          if (start_end == START) {
            if (min > count1) {
              min = count1;
            }
          }
          else
            if (max < count1) {
              max = count1;
            }
      }
  if (start_end == START)
    return min;
  else return max;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define TOLERANCE 1
#define BLOB_THRESHOLD 15
#define OUTPUT_HEIGHT 16
#define OUTPUT_WIDTH 16
#define INT_HEIGHT 16
#define INT_WIDTH 16

typedef char *tBITMAP_ROW;
typedef tBITMAP_ROW *tBITMAP;

typedef int *tINTMAP_ROW;
typedef tINTMAP_ROW *tINTMAP;

struct tBOUNDARY {
  int start_blob, end_blob;
};
```

```
typedef struct tBOUNDARY *tNEW_BLOBTYPE;

typedef enum {START, END} START_END;
typedef enum {ROW, COL} ROW_COL;

/***** recognize *****/
#define nInputNodes     256
#define nHiddenNodes    40
#define nOutputNodes    10
#define nHiddenLayer    1                                                    30
#define INT_HEIGHT      16
#define INT_WIDTH       16
#define NUM_OF_OUTPUTS 10   /* pjliu: taken from datatype.h */
```

```
#include "prototypes.h"


int GetBlobs(tBITMAP input, tBITMAP **outblobs, int height, int width)
{
  int count1,count2;
  tBITMAP copy;
  tBITMAP temp;
  int num_blobs=0;
                                                                           10
  (*outblobs) = (tBITMAP *)calloc((size_t)100, sizeof(tBITMAP *));

  copy = BitmapCreate(height, width);

  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<width;count2++) {
      copy[count1][count2] = input[count1][count2];
    }
  }
                                                                           20
  while(1) {
    temp=NULL;
    GetBlob(copy, &temp, -1, -1, height, width);
    if( temp==NULL ) break;
    (*outblobs)[num_blobs] = temp;
    num_blobs++;
  }

  BitmapDestroy(copy, height);
                                                                           30
  return num_blobs;
}


tINTMAP Bitmap2Intmap(tBITMAP bitmap)
{
  int count1, count2;
  tINTMAP output;
  int startr, startc;
```

```
                                                                    40
  output = IntmapCreate(INT_HEIGHT, INT_WIDTH);
  startr = FindExtremePoint(bitmap, 128, 128, ROW, START);
  startc = FindExtremePoint(bitmap, 128, 128, COL, START);
  if (((startr+INT_HEIGHT) < 128) && ((startc+INT_WIDTH) < 128)) {
    for (count1=startr; count1<(startr+INT_HEIGHT); count1++)
      for (count2=startc; count2<(startc+INT_WIDTH); count2++)
          if (bitmap[count1][count2])
            output[count1-startr][count2-startc] = 1;
  }
  else {                                                            50
    if ((startr+INT_HEIGHT) < 128) {
      for (count1=startr; count1<(startr+INT_HEIGHT); count1++)
          for (count2=startc; count2<128; count2++)
            if (bitmap[count1][count2])
              output[count1-startr][count2-startc] = 1;
    }
    else {
      if ((startc+INT_WIDTH) < 128) {
          for (count1=startc; count1<128; count1++)
            for (count2=startr; count2<(startc+INT_WIDTH); count2++)   60
              if (bitmap[count1][count2])
                output[count1-startr][count2-startc] = 1;
      }
      else {
          for (count1=startc; count1<128; count1++)
            for (count2=startr; count2<128; count2++)
              if (bitmap[count1][count2])
                output[count1-startr][count2-startc] = 1;
      }
    }                                                               70
  }
  return (output);
}

tNEW_BLOBTYPE ConvertBlobs (tBITMAP blob, int height, int width)
{
  tNEW_BLOBTYPE temp_blob;
  int count1, count2;
  int min, max;
                                                                    80
  temp_blob = NewBlobCreate(height);
  for (count1=0; count1<height; count1++) {
    min = width;
    max = 0;
    for (count2=0; count2<width; count2++) {
      if (blob[count1][count2]) {
        if (count2<min)
          min=count2;
        if (count2>max)
          max=count2;                                               90
      }
    }
    temp_blob[count1].start_blob = min;
```

```
    temp_blob[count1].end_blob = max;
  }
  return temp_blob;
}


tBITMAP NewBlob2OldBlob(tBITMAP bitmap, tNEW_BLOBTYPE input, int height,          100
                        int width)
{
  tBITMAP temp_blob;
  int count1, count2;

  temp_blob = BitmapCreate(height, width);
  for (count1=0; count1<height; count1++)
    for (count2=input[count1].start_blob; count2<=input[count1].end_blob;
         count2++)
      if (bitmap[count1][count2])                                             110
        temp_blob[count1][count2] = 1;
  return temp_blob;
}


void GetBlob(tBITMAP input, tBITMAP *output, int start_row, int start_col,
             int height, int width)
{
  int row = start_row;
  int col = start_col;                                                        120
  int rcount;
  int ccount;

  if( (start_row==-1) && (start_col==-1) ) {
    for(col=0;col<width;col++) {
      for(row=0;row<height;row++) {
          if( input[row][col] ) break;
      }
      if( row!=height ) break;
    }                                                                         130
  }

  if( col==width && row==height ) {
    return;
  }

  if((*output) == NULL) (*output)=BitmapCreate(height, width);

  for(ccount=col-TOLERANCE;ccount<=col+TOLERANCE;ccount++) {
    for(rcount=row-TOLERANCE;rcount<=row+TOLERANCE;rcount++) {                140
      if(ccount>=0 && ccount<width && rcount>=0 && rcount<height) {
          if( input[rcount][ccount] ) {
            input[rcount][ccount] = (char)0;
            (*output)[rcount][ccount] = (char)1;
            GetBlob(input, output, rcount, ccount, height, width);
          }
      }
```

```
    }
  }
}
```

```
#include <stdio.h>
#include <ihead.h>

#define BUFSIZE         80

void main(int argc,char *argv[])
{
 int width,height,count1,count2,count3;
 unsigned char bits;
 unsigned char *data;
 IHEAD *ihead;
 FILE *fp;

 unsigned char bitmap_mask[] = {0x01, 0x02, 0x04, 0x08,
                                0x10, 0x20, 0x40, 0x80};


 if( argc != 3 ) {
  printf("Usage:  hsf2bmp <HSFfile> <bitmap>\n");
  exit(0);
 }

 ReadHSF(argv[1],&ihead,&data,&width,&height);

 if((fp = fopen(argv[2],"w")) == NULL){
  printf("Unable to open file %s for writing\n", argv[2]);
  exit(0);
 }

 fprintf(fp, "#define %s_width %d\n", argv[2], width);
 fprintf(fp, "#define %s_height %d\n", argv[2], height);
 fprintf(fp, "static unsigned char %s_bits[] = {\n", argv[2]);


 for(count1=0;count1<height;count1++) {
  for(count2=0;count2<width;count2++) {
   if( (!(count2%8) && count2!=0) || count2==(width-1)) {
        if( count1==(height-1) && count2==(width-1) ) {
         fprintf(fp, "%s};\n", uchar2hexstring(bits));
        }
        else {
         fprintf(fp, "%s,\n", uchar2hexstring(bits));
        }
        bits = (unsigned char)0;
   }
   if(get_next_bit(data)) bits += bitmap_mask[count2%8];
  }
 }
```

```
  fclose(fp);                                                          50
  free(ihead);
  free(data);
}

int get_next_bit(unsigned char *data)
{
  static int byte_offset=-1;
  static int bit_offset=-1;

  unsigned char bitmap_mask[] = {0x80, 0x40, 0x20, 0x10,               60
                                 0x08, 0x04, 0x02, 0x01};

  bit_offset++;
  if( bit_offset==8 )bit_offset=0;
  if( bit_offset==0 )byte_offset++;

  return (data[byte_offset]&bitmap_mask[bit_offset]);
}
```

```
/*************************************************************/
/*       File Name: HSF2SUN.C                                */
/*       Package:   NIST Ihead to Sun Rasterfile             */
/*       Author:    Michael D. Garris                        */
/*       Date:      3/08/90                                  */
/*************************************************************/
#include <stdio.h>
#include <rasterfile.h>
#include <ihead.h>
                                                               10
#define BUFSIZE        80
#define SUNEXT         ".ras"

/*************************************************************/
/* Converts an NIST ihead image to a sun rasterfile.         */
/*************************************************************/
main(argc,argv)
int argc;
char *argv[];
{                                                              20
  int width,height,filesize;
  struct rasterfile rasterhdr;
  char *filename, sunfile[BUFSIZE];
  unsigned char *data;
  IHEAD *ihead;
  FILE *fp;

  procargs(argc,argv,&filename);
  ReadHSF(filename,&ihead,&data,&width,&height);
  rasterhdr.ras_magic = RAS_MAGIC;                             30
  rasterhdr.ras_width = width;
  rasterhdr.ras_height = height;
  rasterhdr.ras_depth = 1;
```

```
  filesize = (width>>3) * height;
  rasterhdr.ras_length = filesize;
  rasterhdr.ras_type = RT_STANDARD;
  rasterhdr.ras_maplength = 0;
  rasterhdr.ras_maptype = RMT_NONE;

  strcpy(sunfile, filename);                                          40
  fileroot(sunfile);
  strcat(sunfile,SUNEXT);
  if((fp = fopen(sunfile,"w")) == NULL){
    fprintf(stderr,"Unable to open file %s\n",sunfile);
    exit(-1);
  }
  fwrite(&rasterhdr,sizeof(struct rasterfile),1,fp);
  fwrite(data,sizeof(char),rasterhdr.ras_length,fp);
  fclose(fp);
  free(ihead);                                                        50
  free(data);
}

/*********************************************************/
/*      Routine:   FileRoot()                            */
/*      Author:    Michael D. Garris                     */
/*      Date:      11/27/89                              */
/*********************************************************/
/*************************************************************/
/* Fileroot() is a destructive procedure which takes a file  */  60
/* name and strips off the rightmost extension, if it exists.*/
/*************************************************************/
fileroot(file)
char *file;
{
  char *cptr;

  cptr = file + strlen(file);
  while((cptr != file) && (*cptr != '.'))
    cptr--;                                                           70
  if(*cptr == '.')
    *cptr = NULL;
}

procargs(argc,argv,filename)
int argc;
char *argv[],**filename;
{
  if(argc < 2){
    fprintf(stderr,"Usage:  hsf2sun <HSF file>\n");                   80
    exit(-1);
  }
 *filename = argv[1];
}
```

```
#include <ihead.h>
```

```
#include "prototypes.h"

void main(int argc, char *argv[])
{
  IHEAD *ihead;
  unsigned char *data;
  int width, height;
  int valid_blobs;

  FILE *outfile;
  FILE *number_file;

  int count, count1, count2;
  int num_blobs, num_digit_blobs;
  tNEW_BLOBTYPE *digit_blobs;

  tBITMAP bitmap, blob;
  tBITMAP stage1_blob, stage2_blob, stage3_blob, stage4_blob;
  int num_digits;
  float temp_percent;
  char recognized;

  char filename[100];
  char out_filename[100];
  char number_filename[100];
  int filename_length;
  int num_files;
  char line_string[80];
  int digit;
  int right[10];
  int wrong[10];
  int uncertain[10];
  int intmap_weight;

  tINTMAP intmap;

  int startr, startc;
  char digit_string[10];

  char *recognizer_file = "/mit/imaging/pjliu/stn.wts";

  if (argc < 3) {
    printf("Usage:  remove_rectangles <MISfile> <output file>\n");
    exit(0);
  }

  sprintf(out_filename, argv[argc-1]);
  if( (outfile = fopen(out_filename, "w")) == NULL) {
    printf("remove_box:     Couldn't open file %s for writing\n", out_filename);
    exit(0);
  }

  ReadWeights(recognizer_file, 0);
```

```
for (count=0; count<10; count++) {
  right[count] = 0;
  wrong[count] = 0;
  uncertain[count] = 0;
}

for (num_files=1; num_files<(argc-1); num_files ++) {

  strcpy(number_filename, argv[num_files]);
  filename_length = strlen(number_filename);
  number_filename[filename_length-3] = 'c';
  number_filename[filename_length-2] = 'l';

  if ((number_file = fopen(number_filename, "r")) == NULL) {
    printf("remove_box:  Couldn't open %s for reading\n", number_file);
    exit(0);
  }

  fgets(line_string, 80, number_file);
  sscanf(line_string, "%d", &num_digits);

  ReadHSF(argv[num_files],&ihead, &data, &width, &height);

  bitmap = BitmapCreate(height, width);
  hsfbits_to_char_array(data, bitmap, height, width);

  blob = BitmapCreate(128, width);

  for (count=0; count<num_digits; count++) {

    for (count1=0; count1<128; count1++)
        for (count2=0; count2<width; count2++)
          blob[count1][count2] = bitmap[count1+(count*128)][count2];

    stage1_blob = Slant(blob, 128, width);
    stage2_blob = Thinning(stage1_blob, 128, width);
    stage3_blob = Thicken(stage2_blob, 128, width);
    stage4_blob = Normalize(stage3_blob, 128, width);

    fgets(line_string, 80, number_file);
    sscanf(line_string, "%d", &digit);
    digit -= 30;
    fprintf(outfile, "%d ", digit);

    intmap = Bitmap2Intmap(stage4_blob);


    recognized = Recognition(intmap, &temp_percent, digit);
    fprintf(outfile, "%c\n", recognized);

    sprintf(digit_string, "%d", digit);
    if (recognized == digit_string[0])
        right[digit]++;
    else {
```

```
        if (recognized == '*')
          uncertain[digit]++;
        else wrong[digit]++;
      }

      intmap_weight = FindIntmapWeight(intmap, INT_HEIGHT, INT_WIDTH);

      IntmapDestroy(intmap, INT_HEIGHT);

      BitmapDestroy(stage1_blob, 128);
      BitmapDestroy(stage2_blob, 128);
      BitmapDestroy(stage3_blob, 128);
      BitmapDestroy(stage4_blob, 128);
    }
    BitmapDestroy(bitmap, 128);
    BitmapDestroy(blob, 128);
  }
  fclose(outfile);

  for (count=0; count<10; count++)
    printf("%d\t%d\t%d\t%d\n", count, right[count], wrong[count], uncertain[count]);
}
```

```
#include "prototypes.h"

#define MIN(x,y) ((x) < (y) ? (x) : (y))
#define MAX(x,y) ((x) > (y) ? (x) : (y))


tBITMAP Normalize(tBITMAP bitmap, int height, int width)
{
  int startr, stopr, startc, stopc;
  tBITMAP output;
  float total;
  int count,count2,count3,count4;
  float rowscale, colscale;
  float rowfrac, colfrac;
  float block_startr, block_stopr, block_startc, block_stopc;
  int debug_total=0;

  output=BitmapCreate(height, width);

  startr = FindExtremePoint(bitmap, height, width, ROW, START);
  stopr = FindExtremePoint(bitmap, height, width, ROW, END);
  startc = FindExtremePoint(bitmap, height, width, COL, START);
  stopc = FindExtremePoint(bitmap, height, width, COL, END);

  rowscale = (float)(stopr-startr+1.0)/16.0;
  colscale = (float)(stopc-startc+1.0)/16.0;

  for(count=0;count<16;count++) {
    for(count2=0;count2<16;count2++) {
      total = 0.0;
```

```
    block_startr = (float)count*rowscale + (float)startr;
    block_stopr = block_startr+rowscale;
    if (block_stopr >= height)
        block_stopr--;

    block_startc = (float)count2*colscale + (float)startc;
    block_stopc = block_startc + colscale;
    if (block_stopc >= width)
        block_stopc--;                                                    40

    for(count3=(int)floor((double)block_startr);
          count3<=(int)floor((double)block_stopr); count3++) {
        for(count4=(int)floor((double)block_startc);
            count4<=(int)floor((double)block_stopc); count4++) {
          if( !((rowfrac=block_stopr-block_startr) < 1.0) ) {
            rowfrac = MIN( (float)count3+1.0-block_startr, 1.0);
            rowfrac = MIN( rowfrac, block_stopr-(float)count3);
          }
          if( !((colfrac=block_stopc-block_startc) < 1.0) ) {             50
            colfrac = MIN( (float)count4+1.0-block_startc, 1.0);
            colfrac = MIN( colfrac, block_stopc-(float)count4);
          }

          total += rowfrac * colfrac *
            (float)(bitmap[count3][count4]);
        }
    }
    if (total > colscale*rowscale/3.0) {
        output[count+(128/2)][count2+(width/2)] = 1;                      60
        debug_total++;
    }
   }
  }
  return output;
}
```

```
#include "prototypes.h"

int PostProcess(tINTMAP bit, float *act_val)
{
  int i ,j;
  int bits[INT_HEIGHT*INT_WIDTH];
  int ind_maxact1 = 0, ind_maxact2 = 0 ;
  float maxact1 = - 1.0 , maxact2 = -1.0 ;
  int index=0, result;
                                                                          10
  for(i=0;i < INT_HEIGHT;i++) {
    for(j=0;j < INT_WIDTH;j++) {
      bits[index]=bit[i][j];
      index++;
    }
  }
```

```
for(i = 0; i<10; i++) {
  if(maxact1 <  act_val[i]) {
    maxact1 = act_val[i] ;
    ind_maxact1 = i ;
  }
}
for(i=0; i<10; i++) {
  if(i != ind_maxact1) {
    if(maxact2 < act_val[i]) {
        maxact2 = act_val[i] ;
        ind_maxact2 = i ;
    }
  }
}

if((ind_maxact1 == 8 && ind_maxact2 == 9) ||
   (ind_maxact1 == 9 && ind_maxact2 == 8))
  return(check8and9(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 5) ||
          (ind_maxact1 == 5 && ind_maxact2 == 3))
  return(check3and5(bits)) ;
else if((ind_maxact1 == 5 && ind_maxact2 == 8) ||
          (ind_maxact1 == 8 && ind_maxact2 == 5))
  return(check5and8(bits)) ;
else if((ind_maxact1 == 4 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 4))
  return(check4and9(bits)) ;
else if((ind_maxact1 == 2 && ind_maxact2 == 6) ||
          (ind_maxact1 == 6 && ind_maxact2 == 2))
  return(check2and6(bits)) ;
else if((ind_maxact1 == 1 && ind_maxact2 == 6) ||
          (ind_maxact1 == 6 && ind_maxact2 == 1))
  return(check1and6(bits)) ;
else if((ind_maxact1 == 7 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 7))
  return(check7and9(bits)) ;
else if((ind_maxact1 == 5 && ind_maxact2 == 6) ||
          (ind_maxact1 == 6 && ind_maxact2 == 5))
  return(check5and6(bits)) ;
else if((ind_maxact1 == 5 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 5))
  return(check5and9(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 8) ||
          (ind_maxact1 == 8 && ind_maxact2 == 3))
  return(check3and8(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 3))
  return(check3and9(bits)) ;
else if((ind_maxact1 == 3 && ind_maxact2 == 6) ||
          (ind_maxact1 == 6 && ind_maxact2 == 3))
  return(check3and6(bits)) ;
else if((ind_maxact1 == 0 && ind_maxact2 == 1) ||
          (ind_maxact1 == 1 && ind_maxact2 == 0))
  return(check0and1(bits)) ;
```

```
  else if((ind_maxact1 == 0 && ind_maxact2 == 2) ||
          (ind_maxact1 == 2 && ind_maxact2 == 0))
    return(check0and2(bits)) ;
  else if((ind_maxact1 == 0 && ind_maxact2 == 3) ||
          (ind_maxact1 == 3 && ind_maxact2 == 0))
    return(check0and3(bits)) ;
  else if((ind_maxact1 == 0 && ind_maxact2 == 4) ||
          (ind_maxact1 == 4 && ind_maxact2 == 0))
    return(check0and4(bits)) ;
  else if((ind_maxact1 == 0 && ind_maxact2 == 5) ||                    80
          (ind_maxact1 == 5 && ind_maxact2 == 0))
    return(check0and5(bits)) ;
  else if((ind_maxact1 == 0 && ind_maxact2 == 6) ||
          (ind_maxact1 == 6 && ind_maxact2 == 0))
    return(check0and6(bits)) ;
  else if((ind_maxact1 == 0 && ind_maxact2 == 7) ||
          (ind_maxact1 == 7 && ind_maxact2 == 0))
    return(check0and7(bits)) ;
  else if((ind_maxact1 == 0 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 0))                      90
    return(check0and9(bits)) ;
  else if((ind_maxact1 == 1 && ind_maxact2 == 8) ||
          (ind_maxact1 == 8 && ind_maxact2 == 1))
    return(check1and8(bits)) ;
  else if((ind_maxact1 == 1 && ind_maxact2 == 9) ||
          (ind_maxact1 == 9 && ind_maxact2 == 1))
    return(check1and9(bits)) ;
  else return(ind_maxact1) ;
}
                                                                       100

int check1and2(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;                                                   110
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      if((j>=0 && j<5) || (j>10 && j<16))
          if(bits[total] == 1)
            mid_ones++ ;
      total++ ;
    }                                                                  120
  if(mid_ones > 2)
    return(2);
  else
    return(1);
```

```
}


int  check1and3(int * bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int total = 0 ;
 int changed;
 int *origout2, i, j;
 int trials = 0;
 int rejected = 0;
 int mid_ones = 0 ;
 int wrong_ones = 0;
 int orig, pres_bit;

 for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
   if((j>=0 && j<5) || (j>10 && j<16))
        if(bits[total] == 1)
          mid_ones++ ;
   total++ ;
  }
 if(mid_ones > 2)
  return(3) ;
 else
   return(1) ;
}


int  check1and4(int * bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int total = 0 ;
 int changed;
 int *origout2, i, j;
 int trials = 0;
 int rejected = 0;
 int mid_ones = 0 ;
 int wrong_ones = 0;
 int orig, pres_bit;

 for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
   if((j>=0 && j<5) || (j>10 && j<16))
        if(bits[total] == 1)
          mid_ones++ ;
   total++ ;
  }
 if(mid_ones > 2)
  return(4) ;
 else
  return(1) ;
}
```

```
int  check1and5(int * bits)                                          180
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;                                                190

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      if((j>=0 && j<5) || (j>10 && j<16))
           if(bits[total] == 1)
             mid_ones++ ;
      total++ ;
    }
  if(mid_ones > 2)
    return(5) ;                                                      200
  else
    return(1) ;
}


int  check1and7(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;                                                       210
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
      if((j>=0 && j<5) || (j>10 && j<16))                            220
           if(bits[total] == 1)
             mid_ones++ ;
      total++ ;
    }
  if(mid_ones > 2)
    return(7) ;
  else
    return(1) ;
}
                                                                     230

int check0and1(int * bits)
```

```
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
  for (i = 0; i < 16; i++) {                                          240
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      if (i > 9 && i < 16) {
          pres_bit = bits[total];
          if (pres_bit != orig) {
            changes++;
            if (changes > 2)
              second_changes++;                                       250
      total++ ;
          }
          orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(1) ;
  else
    return(0) ;                                                       260
}


int check0and2(int *bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;                                             270
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0, j < 16; j++) {
      if (i > 9 && i < 16) {
          pres_bit = bits[total];
          if (pres_bit != orig) {                                     280
            changes++;
            if (changes > 2)
              second_changes++;
      total++ ;
          }
          orig = pres_bit;
```

```
    }
   }
  }
  if (second_changes < 2)                                          290
   return(2) ;
  else return(0) ;
}


int  check0and3(int *bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;                                300
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
   orig = 0;
   pres_bit = 0;
   changes = 0;
   for (j = 0; j < 16; j++) {
    if (i > 9 && i < 16) {                                         310
        pres_bit = bits[total];
        if (pres_bit != orig) {
          changes++;
          if (changes > 2)
            second_changes++;
    total++ ;
        }
        orig = pres_bit;
    }
   }                                                               320
  }
  if (second_changes < 2)
   return(3) ;
  else return(0) ;
}


int  check0and4(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];        330
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int which_one = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
   orig = 0;
   pres_bit = 0;                                                   340
```

```
    changes = 0;
    for (j = 0; j < 16; j++) {
      if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
      total++ ;
            }
            orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(4) ;
  else return(0) ;
}


int  check0and5(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      if (i > 9 && i < 16) {
            pres_bit = bits[total];
            if (pres_bit != orig) {
              changes++;
              if (changes > 2)
                second_changes++;
      total++ ;
            }
            orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(5) ;
  else return(0) ;
}


int check0and6(int * bits)
```

```
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int change_matrix[20];
 int *origout2, i, j, total = 0  ;
 int trials = 0;
 int second_changes = 0;                                          400
 int pres_bit, orig, changes;

 for (i = 0; i < 16; i++) {
  orig = 0;
  pres_bit = 0;
  changes = 0;
  for (j = 0; j < 16; j++) {
   if (i >= 0 && i < 9) {
        pres_bit = bits[total];
        if (pres_bit != orig) {                                   410
          changes++;
          if (changes > 2)
            second_changes++;
    total++ ;
        }
        orig = pres_bit;
   }
  }
 }
 if (second_changes < 2)                                          420
  return(6) ;
 else return(0) ;
}


int  check0and7(int * bits)
{
 int actual, *pnumeral, *origout, *origin, matrix[16][16];
 int change_matrix[20];
 int *origout2, i, j, total = 0  ;                                430
 int trials = 0;
 int second_changes = 0;
 int pres_bit, orig, changes;

 for (i = 0; i < 16; i++) {
  orig = 0;
  pres_bit = 0;
  changes = 0;
  for (j = 0; j < 16; j++) {
   if (i > 9 && i < 16) {                                         440
        pres_bit = bits[total];
        if (pres_bit != orig) {
          changes++;
          if (changes > 2)
            second_changes++;
    total++ ;
        }
        orig = pres_bit;
```

```
    }
   }
  }
  if (second_changes < 2)
   return(7) ;
  else return(0) ;
}


int check0and8(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int mid_ones = 0 ;
  int wrong_ones = 0;
  int orig, pres_bit;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
     if(i>=6 && i<9 && j>=6 && j<9)
          if(bits[total])
            mid_ones++ ;
     total++ ;
    }
  if(mid_ones > 2)
    return(8) ;
  else return(0) ;
}


int  check0and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int which_one = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
     if (i > 9 && i < 16) {
          pres_bit = bits[total];
          if (pres_bit != orig) {
            changes++;
            if (changes > 2)
```

```
            second_changes++;
      total++ ;
          }
          orig = pres_bit; }
    }
  }
  if (second_changes < 2)
    return(9) ;
  else return(0) ;
}


int  check1and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j;
  int trials = 0;
  int which_one = 0;
  int total_number = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
  int index = 0 ;

  for (i = 0; i < 20; i++)
    change_matrix[i] = 0;

  changes = 0;
  total_number++;
  second_changes = 0;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;
    for (j = 0; j < 16; j++) {
      if (i > 9 && i < 16) {
          pres_bit = bits[index] ;
          if (pres_bit != orig) {
            changes++;
            if (changes > 2)
              second_changes++;
      index++ ;
          }
          orig = pres_bit;
      }
    }
  }
  if (second_changes < 2)
    return(1) ;
  else return(6) ;
}
```

```
int  check1and8(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];                                          560
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int which_one = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;

  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;
    changes = 0;                                                  570
    for (j = 0; j < 16; j++) {
      if (i > 9 && i < 16) {
          pres_bit = bits[total];
          if (pres_bit != orig) {
            changes++;
            if (changes > 2)
              second_changes++;
      total++ ;
          }
          orig = pres_bit;                                        580
      }
    }
  }
  if (second_changes < 2)
    return(1) ;
  else return(8) ;
}


int  check1and9(int * bits)                                       590
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
  int second_changes = 0;
  int pres_bit, orig, changes;
  for (i = 0; i < 16; i++) {
    orig = 0;
    pres_bit = 0;                                                 600
    changes = 0;
    for (j = 0; j < 16; j++) {
      if (i >= 0 && i < 9) {
          pres_bit = bits[total];
          if (pres_bit != orig) {
            changes++;
            if (changes > 2)
              second_changes++;
      total++ ;
          }                                                       610
```

```
            orig = pres_bit;
        }
      }
    }
  if (second_changes < 2)
    return(1) ;
  else return(9) ;
}

                                                                    620
int  check2and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int total = 0 ;
  int changed;
  int *origout2, i, j;
  int trials = 0;
  int rejected = 0;
  int zone1_ones, zone2_ones, zone1a_ones, zone2a_ones;
  int zone1b_ones, zone2b_ones, zone1c_ones, zone2c_ones;           630
  int wrong_ones = 0;
  int orig, pres_bit;

  zone1_ones = 0;
  zone2_ones = 0;
  zone1a_ones = 0;
  zone2a_ones = 0;
  zone1b_ones = 0;
  zone2b_ones = 0;
  zone1c_ones = 0;                                                  640
  zone2c_ones = 0;

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    for (j = 0; j < 16; j++) {
      if (i > 3 && i < 7) {
          pres_bit = bits[total] ;
          if (pres_bit != 0) {
            if (j < 6 && i != 4)
              zone1_ones++;                                         650
            if (j < 7 && i != 4)
              zone1a_ones++;
            if (j < 8 && i != 4)
              zone1b_ones++;
            if (j < 8)
              zone1c_ones++;
            if (j > 7)
              zone2c_ones++;
            if (j > 7 && i != 6)
              zone2b_ones++;                                        660
            if (j > 8 && i != 6)
              zone2a_ones++;
            if (j > 9 && i != 6)
              zone2_ones++;
```

```
        }
      }
      total++ ;
    }
  }
  if (zone1_ones == zone2_ones) {                                    670
    if (zone1a_ones == zone2a_ones) {
      if (zone1b_ones == zone2b_ones) {
          zone1_ones = zone1c_ones;
          zone2_ones = zone2c_ones;
      }
      else {
          zone1_ones = zone1b_ones;
          zone2_ones = zone2b_ones;
      }
    }                                                                680
    else {
      zone1_ones = zone1a_ones;
      zone2_ones = zone2a_ones;
    }
  }
  if (zone1_ones > zone2_ones)
    return(6) ;
  if (zone2_ones > zone1_ones)
    return(2) ;
  if (zone1_ones == zone2_ones)                                      690
    printf("cannot make a decision between 2 and 6\n") ;
}


int  check3and5(int * bits)
{
  int i, j , sum = 0, total = 0 ;
  double average ;
  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)                                         700
      {
          if (i >= 3 && i <= 7)
            if (bits[total] == 1)
              sum = sum+j ;
          total++ ;
      }
  average = sum/16.0 ;
  if(average >= 8.0)
    return(3) ,
  else                                                               710
    return(5) ;
}


int  check3and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
```

```
int *origout2, i, j;
int first_max, second_max;
int zone1_ones, zone2_ones;
int pres_bit, total[16], total2[16];
double avg_position[16], avg2_position[16];
int max, min;
int first_sum, last_sum, diff_sum;
int first_pos[16], last_pos[16];
double first_avg, last_avg, diff_avg;
int how_many = 0;
int wrong_ones = 0;
int rejected = 0;
int total_number = 0;
int index = 0, index2 = 0  ;

for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  last_pos[i] = 0;
  total[i] = 0;
  total2[i] = 0;
}

first_max = 0;
second_max = 0;

for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0;
}

how_many++;

for (i = 0; i < 16; i++) {
  pres_bit = 0;
  first_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
    pres_bit = bits[index] ;
    if (pres_bit != 0 && changed == 0) {
        firsts[i] = j;
        changed = 1;
    index++ ;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;
        changed2 = 1;
    }
  }
```

```
    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i];
  }

  for (j = 8; j < 11; j++) {
    if (firsts[j] > first_max) {
      second_max = first_max;
      first_max = firsts[j];                                          780
    }
    else
      if (firsts[j] > second_max)
          second_max = firsts[j];
    first_sum = first_sum + firsts[j];
  }

  first_avg = ((double) (first_max + second_max)) / 2.0;

  if (first_avg > 5.01)                                               790
    return(3) ;
  else return(6) ;
}


int  check3and8(int * bits)
{
  int index = 0 ;
  int index2  = 0 ;
  int actual, *pnumeral, *origout, *origin, matrix[16][16];           800
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16], first_hits[16];
  double first_avg, last_avg, diff_avg, first_hit_avg[16];
  int how_many = 0;                                                   810
  int wrong_ones = 0;
  int rejected = 0;
  int total_number = 0;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    first_hits[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;                                                    820
  }

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
  }
```

```
  how_many++;

  for (i = 0; i < 16; i++) {                                          830
    pres_bit = 0;
    first_sum = 0;
    changed = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
      pres_bit = bits[index] ;
      if (pres_bit != 0 && changed == 0) {
           firsts[i] = j;
           changed = 1;
      index++ ;                                                       840
      }
    }
    for (j = 15; j >= 0; j = j - 1) {
      index2 = i*16 + j ;
      pres_bit = bits[index2] ;
      if (pres_bit != 0 && changed2 == 0) {
           lasts[i] = j;
           changed2 = 1;
      }
    }                                                                 850
    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i];
  }

  for (i = 0, max = lasts[i], min = lasts[i]; i < 16; i++)
    if ((i > 2 && i < 5) || (i > 8 && i < 11))
      first_sum = first_sum + firsts[i];

  first_avg = ((double) first_sum) / 4.0;
                                                                      860
  if (first_avg > 5.49)
    return(3) ;
  else return(8) ;
}


int  check3and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;                       870
  int *origout2, i, j;
  int first_max, second_max;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;                                                   880
```

```
int wrong_ones = 0;
int rejected = 0;
int total_number = 0;
int index = 0 , index2 = 0 ;

for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  last_pos[i] = 0;
  total[i] = 0;
  total2[i] = 0;                                                    890
}

first_max = 0;
second_max = 0;

for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0;
}
                                                                    900
how_many++;

for (i = 0; i < 16; i++) {
  pres_bit = 0;
  first_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
    pres_bit =  bits[index] ;
    if (pres_bit != 0 && changed == 0) {                            910
        firsts[i] = j;
        changed = 1;
    index++ ;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;                                               920
        changed2 = 1;
    }
  }

  total[i] = total[i] + firsts[i];
  total2[i] = total2[i] + lasts[i];
}

for (j = 3; j < 5; j++)
  first_sum = first_sum + firsts[j];                                930

first_avg = ((double) first_sum) / 2.0;

if (first_avg > 4.99)
```

```
    return(3) ;
  else return(9) ;
}


int  check4and9(int * bits)                                              940
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int first_max, second_max;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;                                     950
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;
  int wrong_ones = 0;
  int rejected = 0;
  int total_number = 0;
  int index = 0 ;

  pnumeral = &matrix[0][0];
  /* origout = p1; */                                                   960
  /* origout2 = p2; */
  origin = pnumeral;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;
  }
                                                                        970
  first_max = 0;
  second_max = 0;

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
  }

  how_many++;
                                                                        980
  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    first_sum = 0;
    changed  = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
      index = j*16 + i ;
      pres_bit = bits[index] ;
```

```
      if (pres_bit != 0 && changed == 0) {
            firsts[i] = j;                                          990
            changed = 1;
      }
    }
    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i];
  }

  for (j = 6; j < 10; j++)
    first_sum = first_sum + firsts[j];
                                                                    1000
  total_number++;

  first_avg = ((double) first_sum) / 4.0;

  if (first_avg < 1.51)
    return(9) ;
  else return(4) ;
}

                                                                    1010
int  check5and6(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int total_number = 0;
  int rejected = 0;
  int wrong_ones = 0;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];                              1020
  double avg_position[16], avg2_position[16];
  int first_sum, last_sum, max;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg;
  int how_many = 0;
  int index = 0 , index2 = 0 ;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;                                                1030
    total[i] = 0;
    total2[i] = 0;
  }

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
  }

  how_many++;                                                       1040

  for (i = 0; i < 16; i++) {
```

```
    pres_bit = 0;
    first_sum = 0;
    changed = 0;
    changed2 = 0;
    for (j = 0; j < 16; j++) {
      index++ ;
      pres_bit = bits[index] ;
      if (pres_bit != 0 && changed == 0) {                    1050
          firsts[i] = j;
          changed = 1;
      }
    }
    for (j = 15; j >= 0; j = j - 1) {
      index2 = i*16 + j ;
      pres_bit = bits[index2] ;
      if (pres_bit != 0 && changed2 == 0) {
          lasts[i] = j;
          changed2 = 1;                                       1060
      }
    }

    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i];
  }

  for (i = 8, max = firsts[i]; i < 13; i++) {
    if (firsts[i] > max)
      max = firsts[i];                                        1070
    first_sum = first_sum + firsts[i];
  }

  first_avg = ((double) first_sum) / 5.0;

  first_pos[(int) first_avg]++;

  if (first_avg < 2.0)
    return(6) ;
  else if (first_avg > 4.0)                                   1080
    return(5) ;
  else if (max > 5)
    return(5) ;
  else if (max < 6)
    return(6) ;
  else
    rejected++;
}

                                                              1090
int  check5and8(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int zone1_ones, zone2_ones;
```

```
int pres_bit, total[16], total2[16];
double avg_position[16], avg2_position[16];
int first_sum, last_sum;
int first_pos[16], last_pos[16];                                    1100
double first_avg, last_avg;
int how_many = 0;
int total_number = 0;
int rejected = 0;
int wrong_ones = 0;
int index = 0, index2 = 0 ;

for (i = 0; i < 16; i++) {
  first_pos[i] = 0;
  last_pos[i] = 0;                                                  1110
  total[i] = 0;
  total2[i] = 0;
}

for (i = 0; i < 16; i++) {
  firsts[i] = 16;
  lasts[i] = 0;
}

how_many++;                                                         1120

for (i = 0; i < 16; i++) {
  pres_bit = 0;
  first_sum = 0;
  last_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
    pres_bit = bits[index];
    if (pres_bit != 0 && changed == 0) {                            1130
        firsts[i] = j;
        changed = 1;
    index++ ;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
        lasts[i] = j;                                               1140
        changed2 = 1;
    }
  }

  total[i] = total[i] + firsts[i];
  total2[i] = total2[i] + lasts[i]; }

total_number++;

for (i = 8; i < 12; i++)                                            1150
```

```
    first_sum = first_sum + firsts[i];
  first_avg = ((double) first_sum) / 4.0;
  first_pos[(int) first_avg]++;

  for (i = 3; i < 6; i++)
    last_sum = last_sum + lasts[i];

  last_avg = ((double) last_sum) / 3.0;
  last_pos[(int) last_avg]++;
                                                                    1160
  if (last_avg - first_avg < 2.01)
    return(5) ;
  else if (last_avg - first_avg > 5.99)
    return(8) ;
  else if (last_avg < 5.99)
    return(5) ;
  else if (last_avg > 9.01)
    return(8) ;
  else if (first_avg > 5.99)
    return(5) ;                                                     1170
  else if (first_avg < 5.01)
    return(8) ;
  else
    rejected++;
}


int  check5and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];         1180
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min, sevens, threes;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;                                                 1190
  int wrong_ones = 0;
  int rejected = 0;
  int total_number = 0;
  int index = 0,  index2 = 0 ;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;                                                  1200
  }

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
```

```
  lasts[i] = 0; }

how_many++;
sevens = 0;
threes = 0;
                                                                    1210
for (i = 0; i < 16; i++) {
  pres_bit = 0;
  last_sum = 0;
  diff_sum = 0;
  changed = 0;
  changed2 = 0;
  for (j = 0; j < 16; j++) {
    pres_bit = bits[index] ;;
    if (pres_bit != 0 && changed == 0) {
          firsts[i] = j;                                            1220
          changed = 1;
    index++ ;
    }
  }
  for (j = 15; j >= 0; j = j - 1) {
    index2 = i*16 + j ;
    pres_bit = bits[index2] ;
    if (pres_bit != 0 && changed2 == 0) {
          lasts[i] = j;
          changed2 = 1;                                             1230
    }
  }

  total[i] = total[i] + firsts[i];
  total2[i] = total2[i] + lasts[i]; }

for (i = 2, max = lasts[i], min = lasts[i]; i < 7; i++) {
  if (lasts[i] > max)
    max = lasts[i];
  if (lasts[i] < min)                                               1240
    min = lasts[i];
  if (lasts[i] < 8)
    sevens++;
  if (lasts[i] < 4)
    threes++;
  last_sum = last_sum + lasts[i];
  diff_sum = diff_sum + (lasts[i] - firsts[i]);
}

last_avg = ((double) last_sum) / 5.0;                               1250
diff_avg = ((double) diff_sum) / 5.0;

last_pos[(int) last_avg]++;

if (last_avg < 7.0)
  return(5) ;
else if (last_avg >= 10.0)
  return(9) ;
```

```
  else if (diff_avg < 5.01)
    return(5) ;
  else if (diff_avg > 6.21)
    return(9) ;
  else if (sevens >= 3)
    return(5) :
  else if (threes >= 2)
    return(5) ;
  else
    return(9) ;
}

int  check7and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int changed, firsts[16], lasts[16], changed2;
  int *origout2, i, j;
  int ones = 0;
  int first_max, second_max;
  int zone1_ones, zone2_ones;
  int pres_bit, total[16], total2[16];
  double avg_position[16], avg2_position[16];
  int max, min;
  int first_sum, last_sum, diff_sum;
  int first_pos[16], last_pos[16];
  double first_avg, last_avg, diff_avg;
  int how_many = 0;
  int wrong_ones = 0;
  int rejected = 0;
  int total_number = 0;
  int first_hit_sum;
  int index = 0 , index2 = 0, index3 = 0 ;

  pnumeral = &matrix[0][0];
  origin = pnumeral;

  for (i = 0; i < 16; i++) {
    first_pos[i] = 0;
    last_pos[i] = 0;
    total[i] = 0;
    total2[i] = 0;
  }

  first_max = 0;
  second_max = 0;

  for (i = 0; i < 16; i++) {
    firsts[i] = 16;
    lasts[i] = 0;
  }

  first_hit_sum = 0;
  how_many++;
```

```
  ones = 0;

  for (i = 0; i < 16; i++) {
    pres_bit = 0;
    first_sum = 0;
    changed = 0;
    changed2 = 0;                                                    1320
    for (j = 0; j < 16; j++) {
      pres_bit = bits[index] ;
      if (pres_bit != 0 && changed == 0) {
           firsts[i] = j;
           changed = 1;
      index++ ;
      }
    }
    for (j = 15; j >= 0; j = j - 1) {
      index2 = i*16 + j ;                                            1330
      pres_bit = bits[index2] ;
      if (pres_bit != 0 && changed2 == 0) {
           lasts[i] = j;
           changed2 = 1;
      }
    }
    if (i == 9 || i == 10 || i == 11) {
      changed = 0;
      for (j = 0; j < (i + 1)  ; j++) {
           index3 = (i-j) * 16 + j ;                                 1340
           pres_bit = bits[index3] ;
           if (pres_bit != 0 && changed == 0) {
             first_hit_sum = first_hit_sum + j;
             changed = 1;
           }
      }
    }

    total[i] = total[i] + firsts[i];
    total2[i] = total2[i] + lasts[i]; }                              1350

  if (first_hit_sum > 16)
    return(7) ;
  else if (first_hit_sum < 17)
    return(9) ;
  else rejected++;
  total_number++;
}


                                                                     1360
int  check8and9(int * bits)
{
  int actual, *pnumeral, *origout, *origin, matrix[16][16];
  int change_matrix[20];
  int *origout2, i, j, total = 0  ;
  int trials = 0;
```

```
 int which_one = 0;
 int second_changes = 0;
 int pres_bit, orig, changes;
                                                                    1370
 for (i = 0; i < 16; i++) {
  orig = 0;
  pres_bit = 0;
  changes = 0;
  for (j = 0; j < 16; j++) {
   if (i > 9 && i < 16) {
        pres_bit = bits[total];
        if (pres_bit != orig) {
          changes++;
          if (changes > 2)                                          1380
            second_changes++;
    total++ ;
        }
        orig = pres_bit;
    }
   }
 }
 if (second_changes < 2)
   return(9) ;
 else return(8) ;                                                   1390
}
```

```
#include "defines.h"

/***** getblob.c *****/
void GetBlob(tBITMAP, tBITMAP *, int, int, int, int);
tNEW_BLOBTYPE ConvertBlobs(tBITMAP, int height, int width);
tBITMAP NewBlob2OldBlob(tBITMAP, tNEW_BLOBTYPE, int height, int width);
tINTMAP Bitmap2Intmap(tBITMAP);
tBITMAP RemoveRectangle(tBITMAP, int, int);
int GetBlobs(tBITMAP, tBITMAP **, int, int);
                                                                    10
/***** allocation.c *****/
tBITMAP BitmapCreate(int height, int width);
tNEW_BLOBTYPE NewBlobCreate(int);
tINTMAP IntmapCreate(int height, int width);
void IntmapDestroy(tINTMAP, int height);
void NewBlobDestroy(tNEW_BLOBTYPE);
void BitmapDestroy(tBITMAP, int height);

/***** box_utils.c *****/
int FindBox(tBITMAP bitmap, tNEW_BLOBTYPE *, int valid_blobs, int height);   20
int FindIntmapWeight(tINTMAP intmap, int height, int width);
```

```
int FindBlobWeight(tBITMAP bitmap, tNEW_BLOBTYPE blob, int height);
int ValidBlob(tBITMAP bitmap, tNEW_BLOBTYPE, int height, int width,
              ROW_COL row_col, START_END start_end);
int FindExtremePoint(tBITMAP bitmap, int height, int width, ROW_COL row_col,
                     START_END start_end);

/***** utils.c *****/
int hsfbits_to_char_array(unsigned char *hsf_data, tBITMAP bitmap,
                          int height, int width);                    30
void TextWrite(FILE *outfile, tBITMAP);
int fwrite_bitmap(char *filename, tBITMAP bitmap, int height, int width);
int fwrite_intmap(char *filename, tINTMAP intmap, int height, int width);
int fwrite_newblob(char *filename, tBITMAP checkmap, tNEW_BLOBTYPE blob, int
                   height, int width);
int get_next_bit(unsigned char *data);
char *uchar2hexstring(unsigned char bits);

/***** normalize.c *****/
tBITMAP Normalize(tBITMAP bitmap, int height, int width);            40

/***** slant.c *****/
tBITMAP rotate(tBITMAP bitmap, int height, int width, double angle, int shift);
int get_rotated_width(tBITMAP bitmap, int height, int width, double angle,
                      int *total_shift);
tBITMAP Slant(tBITMAP bitmap, int height, int width);

/***** thicken.c *****/
tBITMAP Thicken (tBITMAP bitmap, int height, int width);
                                                                     50
/***** thinning.c *****/
tBITMAP Thinning (tBITMAP bitmap, int height, int width);

/***** recognize.c *****/

/* IO functions */
int ReadWeights(char *wInfileName, int x);

/* BackPropagation Core function prototypes */
                                                                     60
/* pjliu: (3-30-96) added last parameter to Recognition */
char Recognition(tINTMAP, float *, int);
int InitNet(void);
float Logistic(float);
float *Recognize_Hand(tINTMAP, int);
int FirstPropagate(int x);
int FreeLearn(void);

/***** postproc.c *****/
int PostProcess(tINTMAP, float*);                                    70
int check8and9(int *) ;
int check3and5(int *) ;
int check2and6(int *) ;
int check3and8(int *) ;
int check5and8(int *) ;
```

```
int check5and9(int *) ;
int check5and6(int *) ;
int check4and9(int *) ;
int check7and9(int *) ;
int check1and6(int *) ;
int check3and9(int *) ;
int check3and6(int *) ;
int check0and1(int *) ;
int check0and2(int *) ;
int check0and3(int *) ;
int check0and4(int *) ;
int check0and5(int *) ;
int check0and6(int *) ;
int check0and7(int *) ;
int check0and9(int *) ;
int check1and8(int *) ;
int check1and9(int *) ;
```

```
#include "defines.h"

/***** getblob.c *****/
void GetBlob(tBITMAP, tBITMAP *, int, int, int, int);
tNEW_BLOBTYPE ConvertBlobs(tBITMAP, int height, int width);
tBITMAP NewBlob2OldBlob(tBITMAP, tNEW_BLOBTYPE, int height, int width);
tBITMAP RemoveRectangle(tBITMAP, int, int);
int GetBlobs(tBITMAP, tBITMAP **, int, int);

/***** allocation.c *****/
tBITMAP BitmapCreate(int height, int width);
tNEW_BLOBTYPE NewBlobCreate(int);
void NewBlobDestroy(tNEW_BLOBTYPE);
void BitmapDestroy(tBITMAP, int height);

/***** box_utils.c *****/
int FindBox(tBITMAP bitmap, tNEW_BLOBTYPE *, int valid_blobs, int height);
int FindBlobWeight(tBITMAP bitmap, tNEW_BLOBTYPE blob, int height);
int ValidBlob(tBITMAP bitmap, tNEW_BLOBTYPE, int height, int width,
              ROW_COL row_col, START_END start_end);
int FindExtremePoint(tBITMAP bitmap, int height, int width, ROW_COL row_col,
                     START_END start_end);

/***** utils.c *****/
void bitmap_to_char_array(unsigned char *xbitmap, tBITMAP bitmap,
                          int height, int width);
int hsfbits_to_char_array(unsigned char *hsf_data, tBITMAP bitmap,
                          int height, int width);
void TextWrite(FILE *outfile, tBITMAP, int startr, int startc);
int fwrite_bitmap(char *filename, tBITMAP bitmap, int height, int width);
int fwrite_newblob(char *filename, tBITMAP checkmap, tNEW_BLOBTYPE blob, int
                   height, int width);
int get_next_bit(unsigned char *data);
char *uchar2hexstring(unsigned char bits);
```

```
/***** normalize.c *****/
tBITMAP Normalize(tBITMAP bitmap, int height, int width);

/***** slant.c *****/
tBITMAP rotate(tBITMAP bitmap, int height, int width, double angle, int shift);

int GetBlobWidth(tBITMAP bitmap, int height, int width, int *min, int *max,
                 int *total_shift, int startr, int startc);
tBITMAP Slant(tBITMAP bitmap, int height, int width);

/***** thicken.c *****/
tBITMAP Thicken (tBITMAP bitmap, int height, int width);

/***** thinning.c *****/
tBITMAP Thinning (tBITMAP bitmap, int height, int width);
```

```
#include "prototypes.h"

/* pjliu: types were float huge, instead of just float */
/* activation nodes */
float iNode[nInputNodes];
float hNode[nHiddenNodes];
float oNode[nOutputNodes];
float inputHNode[nHiddenNodes];
float inputONode[nOutputNodes];

/* synaptic weights and delta weights and bias */
float wt1[nHiddenNodes][nInputNodes];
float wt2[nOutputNodes][nHiddenNodes];
float bias1[nHiddenNodes];
float bias2[nOutputNodes];

float wt3[nHiddenNodes][nInputNodes];
float wt4[nOutputNodes][nHiddenNodes];
float bias3[nHiddenNodes];
float bias4[nOutputNodes];


char Recognition(tINTMAP intmap, float *percentage, int correct_number)
{
  int output;
  float *percents;
  int max_index, max_index2;
  int count;
  int temp;

  max_index = 0;
  max_index2 = 0;

  percents = (float *)calloc((size_t)10, sizeof(float));
```

```
  percents = Recognize_Hand(intmap, 0);

  for (count=0; count<10; count++)
    if (percents[count] > percents[max_index])
      max_index = count;
    else                                                              40
      if (percents[count] > percents[max_index2])
          max_index2 = count;

  if ((percents[max_index] < 0.7) || ((percents[max_index] > 0.7) &&
                                      (percents[max_index2] > 0.25))) {
    temp = PostProcess(intmap, percents);

/*    printf("%d: %d %d\n", correct_number, max_index, temp); */

    if (temp != max_index)                                            50
      return('*');
    else
      if (percents[max_index] < 0.4)
          return('*');
  }

  *percentage = percents[max_index];
  return((char)(max_index-48));
}
                                                                      60


float *Recognize_Hand(tINTMAP map, int NET)
{
  register int i, j;
  int cnt = 0;
  char szInfo[30];

  /* pjliu: added result variable */
  float *result;                                                      70

  result = (float *)malloc((size_t)10 * sizeof(float));

  /* fill the inputNodes */
  for (i = 0; i < 16; ++i) {
    for (j= 0; j < 16; ++j) {
      iNode[cnt++] = map[i][j];
    }
  }
                                                                      80
  if (NET == 0)
    FirstPropagate(0);
  else
    FirstPropagate(1);

  for(i=0;i<10;i++)
    result[i] = oNode[i];
```

```
  return result;
}                                                                    90


/* weight file format is outlined in backprop.doc */
int ReadWeights(char *wInfileName, int x)
{
  static char szAppName[] = "PROTO2";
  char szInfo[30];
  float temp;
  float wt[nOutputNodes][nHiddenNodes];
  float bias[nOutputNodes];                                          100
  float wts[nOutputNodes][nHiddenNodes];
  float biass[nOutputNodes];
  FILE *wInfile;
  short y, z;

  int count=0; /* pjliu (3-23-96): for debugging */
  int scan_return;

  if ((wInfile = fopen(wInfileName, "r")) == NULL)
    return 1;                                                        110
  {
    /* current implementation has only 1 Hidden layer */
    register int i, j;

    for (i = 0; i < nHiddenNodes; ++i)
      {

          count++;

          if ((scan_return = fscanf(wInfile, "%f", &temp)) != 1)     120
            printf("%d %d\n", count, scan_return);

          if (x)
            bias3[i] = temp;
          else
            bias1[i] = temp;
      }

    for (i = 0; i < nHiddenNodes; ++i)
      {                                                              130
          for (j = 0; j < nInputNodes; ++j)
            {

              count++;
              if ((scan_return = fscanf(wInfile, "%f", &temp)) != 1)
                printf("%d %d\n", count, scan_return);

              if (x)
                wt3[i][j] = temp;
              else                                                   140
                wt1[i][j] = temp;
            }
```

```
    }

  for (i = 0; i < nOutputNodes; ++i)
   {
        count++;

        if ((scan_return = fscanf(wInfile, "%f", &temp)) != 1)
          printf("%d %d\n", count, scan_return);

        if (x)
          bias4[i] = temp;
        else
          bias2[i] = temp;
    }

  for (i = 0; i < nOutputNodes; ++i)
   {
        for (j = 0; j < nHiddenNodes; ++j)
         {
          count++;

          if ((scan_return = fscanf(wInfile, "%f", &temp)) != 1)
            printf("%d %d\n", count, scan_return);

          if (x)
            wt4[i][j] = temp;
          else
            wt2[i][j] = temp;
         }
    }

 }

 fclose(wInfile);
 return 0;
}

/*-----------------------------------------*/
/* BackPropagation Routines are below here */

float Logistic(float num)
{
 if (num > 10.0)
   return(0.99999);
 else if (num < -10.0)
   return(0.00001);
 else
   return(1.0 / (1.0 + (float) exp( (double) ((-1.0) * num))));
}


int FirstPropagate(int x)
{
```

```
  register int i, j;
  float wt[nOutputNodes][nHiddenNodes];
  float bias[nOutputNodes];
  float wts[nOutputNodes][nHiddenNodes];                                    200
  float biass[nOutputNodes];
  short y, z;

  /* forward propagation */
  for (i = 0; i < nHiddenNodes; ++i)
    {
      if (x)
          inputHNode[i] = bias3[i];
      else
          inputHNode[i] = bias1[i];                                         210

      for (j = 0; j < nInputNodes; ++j)
          {
            if (x)
              inputHNode[i] += wt3[i][j] * iNode[j];
            else
              inputHNode[i] += wt1[i][j] * iNode[j];
          }

      /* inputHNode[i] /= nInputNodes + 1; */                               220

      hNode[i] = (float) Logistic(inputHNode[i]);
    }

  for (i = 0; i < nOutputNodes; ++i)
    {
      if (x)
          inputONode[i] = bias4[i];
      else
          inputONode[i] = bias2[i];                                         230

      for (j = 0; j < nHiddenNodes; ++j)
          {
            if (x)
              inputONode[i] += wt4[i][j] * hNode[j];
            else
              inputONode[i] += wt2[i][j] * hNode[j];
          }

      /* inputONode[i] /= nHiddenNodes + 1; */                              240

      oNode[i] = (float) Logistic(inputONode[i]);
    }

  return 0;
}

                                                                            250
```

```
#include <ihead.h>
#include "prototypes.h"

void main(int argc, char *argv[])
{
 IHEAD *ihead;
 unsigned char *data;
 int width, height;
 int valid_blobs;

 FILE *outfile;
 FILE *number_file;

 int count, count1, count2;
 int num_blobs, num_digit_blobs;
 tNEW_BLOBTYPE *digit_blobs;

 tBITMAP bitmap, blob;
 tBITMAP stage1_blob, stage2_blob, stage3_blob, stage4_blob;
 int num_digits;

 char filename[100];
 char out_filename[100];
 char number_filename[100];
 int filename_length;
 int num_files;
 char line_string[80];
 int digit;

 int startr, startc;

 if (argc < 3) {
  printf("Usage:  remove_rectangles <MISfile> <output file>\n");
  exit(0);
 }

 sprintf(out_filename, argv[argc-1]);
 if( (outfile = fopen(out_filename, "w")) == NULL) {
  printf("remove_box:     Couldn't open file %s for writing\n", out_filename);
  exit(0);
 }

 for (num_files=1; num_files<argc-1; num_files ++) {

  strcpy(number_filename, argv[num_files]);
  filename_length = strlen(number_filename);
  number_filename[filename_length-3] = 'c';
  number_filename[filename_length-2] = 'l';
```

```
    if ((number_file = fopen(number_filename, "r")) == NULL) {
      printf("remove_box:  Couldn't open %s for reading\n", number_file);
      exit(0);
    }

    fgets(line_string, 80, number_file);
    sscanf(line_string, "%d", &num_digits);

    ReadHSF(argv[num_files],&ihead, &data, &width, &height);

    bitmap = BitmapCreate(height, width);
    hsfbits_to_char_array(data, bitmap, height, width);

    blob = BitmapCreate(128, width);

    for (count=0; count<num_digits; count++) {

      for (count1=0; count1<128; count1++)
          for (count2=0; count2<width; count2++)
            blob[count1][count2] = bitmap[count1+(count*128)][count2];

      stage1_blob = Slant(blob, 128, width);
      stage2_blob = Thinning(stage1_blob, 128, width);
      stage3_blob = Thicken(stage2_blob, 128, width);
      stage4_blob = Normalize(stage3_blob, 128, width);

      startr = FindExtremePoint(stage4_blob, 128, 128, ROW, START);
      startc = FindExtremePoint(stage4_blob, 128, 128, COL, START);
      fgets(line_string, 80, number_file);
      sscanf(line_string, "%d", &digit);
      fprintf(outfile, "%d\n", (digit-30));
      TextWrite(outfile, stage4_blob, startr, startc);
      fprintf(outfile, "\n");

      BitmapDestroy(stage1_blob, 128);
      BitmapDestroy(stage2_blob, 128);
      BitmapDestroy(stage3_blob, 128);
      BitmapDestroy(stage4_blob, 128);
    }
    BitmapDestroy(bitmap, 128);
    BitmapDestroy(blob, 128);
  }
  fclose(outfile);
}
```

```
#include <values.h>
#include "prototypes.h"


tBITMAP rotate(tBITMAP bitmap, int height, int width, double angle, int shift)
{
  int x, y;
```

```
  tBITMAP output = BitmapCreate(height, width);

  for(y=0;y<height;y++) {                                          10
    int offset = (int)floor(y*tan(angle) + 0.5);

    for(x=0;x<width;x++) {
      int orig_x = x - offset + shift - width/2;
      if(orig_x > 0 && orig_x < width) {
          output[y][x] = bitmap[y][orig_x];
      }
    }
  }
  return output;                                                   20
}


int get_rotated_width(tBITMAP bitmap, int height, int width, double angle,
                      int *total_shift)
{
  int y;
  int min_x=MAXINT;
  int max_x=0;
  float tangent=tan(angle);                                        30
  int rot_x_min, rot_x_max;
  int min_shift = width;
  int max_shift = 0;
  int shift;

  *total_shift = 0;
  for (y=0;y<height;y++) {
    for (rot_x_min=0; rot_x_min<width; rot_x_min++)
      if (bitmap[y][rot_x_min])
          break;                                                   40
    for (rot_x_max=width-1; rot_x_max>=0; rot_x_max--)
      if (bitmap[y][rot_x_max])
          break;
    if(rot_x_max>rot_x_min) {
      shift = (int)floor((float)y * tangent + 0.5);

      rot_x_max += shift;
      rot_x_min += shift;
      min_x = rot_x_min < min_x ? rot_x_min : min_x;
      max_x = rot_x_max > max_x ? rot_x_max : max_x;               50
    }
  }
  /* must keep track of the actual image position and not how much shift */
  *total_shift = (max_x + min_x)/2;
  return max_x - min_x;
}


tBITMAP Slant(tBITMAP bitmap, int height, int width)
{                                                                  60
  int count1, count2;
```

```
  tBITMAP working_blob, temp_blob;
  double angle_count, best_angle;
  int min_width, temp_width;
  int copy_counter, start_copy;
  int total_shift;
  int best_shift;

  best_angle = 0.0;
  min_width = width;                                                    70

  for (angle_count=-0.8; angle_count<=0.9; angle_count+=0.1) {
    temp_width = get_rotated_width(bitmap, height, width, angle_count,
                                   &total_shift);
    if (temp_width < min_width) {
      min_width = temp_width;
      best_angle = angle_count;
      best_shift = total_shift;
    }
  }                                                                     80

  temp_blob = rotate(bitmap, height, width, best_angle, best_shift);
  return (temp_blob);
}
```

```
#include "prototypes.h"

struct array_place{
  int x;
  int y;
};

struct array_place cell[8] = {
  -1, -1,
  -1, 0,                                                                10
  0, -1,
  0, 0
  }; /* These are the relative displacements from the eight directions */


tBITMAP Thicken (tBITMAP bitmap, int height, int width)
{

  tBITMAP temp_data;
                                                                        20
  int count, count1, count2;

  temp_data = BitmapCreate(height, width);

  for (count1=1; count1<(height-1); count1++)
    for (count2=1; count2<(width-1); count2++)
      if (bitmap[count1][count2])
          for (count=0; count<4; ++count)
            temp_data[count1+cell[count].x][count2+cell[count].y] = 1;
```

```
  return (temp_data);
}
```

```
#include "prototypes.h"

int IsBorder (tBITMAP, int, int);
int RightConditions (tBITMAP, int, int);


struct thinning_array_place
{
  int x;
  int y;
};

struct thinning_array_place thinning_cell[8] = {
  -1,  0,
  -1, +1,
  0, +1,
  +1, +1,
  +1,  0,
  +1, -1,
  0, -1,
  -1, -1
  };


int IsBorder (tBITMAP bitmap, int i, int j)
{
  int count, temp_x, temp_y;

  count = 0;
  if (bitmap[i][j])
    for (count=0; count<8; count++) {
      temp_x = i + thinning_cell[count].x;
      temp_y = j + thinning_cell[count].y;
      if (!bitmap[temp_x][temp_y])
          return 1;
      else ++count;
    }
  return 0;
}


int RightConditions (tBITMAP bitmap, int i, int j)
{
  int count, temp_x, temp_y, cond1, cond2, cond3;
  int u, v, color, p, w_b, t1, t2;

  /* initialize */
  count = 0;
  cond1 = 0;
  cond2 = 0;
```

```
  cond3 = 0;

  for (u = 0; u <= 7; ++u) {
    temp_x = thinning_cell[u].x + i;
    temp_y = thinning_cell[u].y + j;
    if (bitmap[temp_x][temp_y])
      ++count;
  }
  if ((count > 1) && (count < 7))
    cond1 = 1;                                                          60

  w_b = 0;
  color = bitmap[i-1][j];
  for (u = 1; u <= 8; ++u) {
    if (u == 8)
      v = 0;
    else v = u;
    temp_x = thinning_cell[v].x + i;
    temp_y = thinning_cell[v].y + j;
    if (!bitmap[temp_x][temp_y]) {                                      70
      if (color) {
          ++ w_b;
          color = 0;
      }
    }
    else color = 1;
  }
  if (w_b == 1)
    cond2 = 1;
  t1 = bitmap[i-1][j]+bitmap[i-1][j+1]+bitmap[i][j+1]+bitmap[i+1][j-1];  80
  t2 = bitmap[i][j+1]+bitmap[i+1][j+1]+bitmap[i+1][j]+bitmap[i-1][j-1];
  if (((! t1) && bitmap[i+1][j]  && bitmap[i][j-1]) ||
      ((! t2) && bitmap[i][j-1] && bitmap[i-1][j]))
    cond3 = 1;
  if (cond1 && (cond2 || cond3))
    return 1;
  else return 0;
}


                                                                        90
tBITMAP Thinning (tBITMAP bitmap, int height, int width)
{
  tBITMAP temp_data;
  int count1, count2;
  int more, g_switch;

  temp_data = BitmapCreate(height, width);

  more = 1;
  g_switch = 1;                                                         100

  while (more) {
    g_switch = !g_switch;
    more = 0;
```

```
   for (count1=1; count1<(height-1); count1++)
    for (count2=1; count2<(width-1); count2++)
        if (IsBorder(bitmap, count1, count2)) {
          if (RightConditions(bitmap, count1, count2)) {
            if (g_switch) {
              if (!((bitmap[count1-1][count2] + bitmap[count1][count2-1])
                    * bitmap[count1][count2+1] * bitmap[count1+1][count2])) {
                  temp_data[count1][count2] = 0;
                  more = 1;
              }
              else temp_data[count1][count2] = 1;
            }
            else {
              if (!((bitmap[count1][count2+1] + bitmap[count1+1][count2])
                    * bitmap[count1-1][count2] * bitmap[count1][count2-1])) {
                  temp_data[count1][count2] = 0;
                  more = 1;
              }
              else temp_data[count1][count2] = 1;
            }
          }
          else temp_data[count1][count2] = bitmap[count1][count2];
        }
        else temp_data[count1][count2] = bitmap[count1][count2];
   for (count1=0; count1<height; count1++)
    for (count2=0; count2<width; count2++)
        bitmap[count1][count2] = temp_data[count1][count2];
  }
  return(temp_data);
}
```

```
#include "prototypes.h"

#define INITIALIZE (void *)-1

int hsfbits_to_char_array(unsigned char *hsf_data, tBITMAP bitmap,
                          int height, int width)
{
  int count1, count2;

  get_next_bit(INITIALIZE);
  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<width;count2++) {
      if(get_next_bit(hsf_data))
          bitmap[count1][count2] = (char)1;
      else bitmap[count1][count2] = (char)0;
    }
  }
}


void TextWrite(FILE *outfile, tBITMAP bitmap)
{
```

```
  int count1, count2;
  int startr, startc;

  /* don't need these because normalize will always bring the image up
     to the top left hand corner of the bitmap
  startr = FindExtremePoint(bitmap, height, width, ROW, START);
  startc = FindExtremePoint(bitmap, height, width, COL, START);
  */                                                                    30

  for (count1=0; count1<OUTPUT_HEIGHT; count1++) {
    for (count2=0; count2<=OUTPUT_HEIGHT; count2++) {
      if (bitmap[count1][count2])
          fprintf(outfile, "1");
      else fprintf(outfile, "0");
    }
    fprintf(outfile,"\n");
  }
}                                                                       40


int fwrite_intmap(char *filename, tINTMAP intmap, int height, int width)
{
  int count1, count2, count3;
  FILE *outfile;
  unsigned char outval;
  unsigned char intmap_mask[8];
  intmap_mask[0] = 0x01;
  intmap_mask[1] = 0x02;                                                50
  intmap_mask[2] = 0x04;
  intmap_mask[3] = 0x08;
  intmap_mask[4] = 0x10;
  intmap_mask[5] = 0x20;
  intmap_mask[6] = 0x40;
  intmap_mask[7] = 0x80;


  if( (outfile = fopen(filename, "w")) == NULL) {
    printf("fwrite_intmap:    Couldn't open file %s for writing\n", filename);   60
    exit(0);
  }

  fprintf(outfile, "#define %s_width %d\n", filename, width);
  fprintf(outfile, "#define %s_height %d\n", filename, height);
  fprintf(outfile, "static unsigned char %s_bits[] = {\n", filename);

  for(count1=0;count1<height;count1++) {
    for(count2=0;count2<((int)ceil((double)(width/8)));count2++) {
      outval = 0;                                                       70
      for(count3=0;count3<8;count3++) {
        if( (count2*8+count3) < 16) {
          if ( intmap[count1][count2*8+count3] ) {
            outval += intmap_mask[count3];
          }
        }
```

```
    }
    if(count1==(height-1) && count2==((int)ceil((double)(width/8))-1)) {
      fprintf(outfile, "%s};\n", uchar2hexstring(outval));
    }
    else {
      fprintf(outfile, "%s,\n", uchar2hexstring(outval));
    }
  }
 }
 fclose(outfile);
}


int fwrite_bitmap(char *filename, tBITMAP bitmap, int height, int width)
{

 int count1, count2;
 FILE *outfile;
 unsigned char outval=0;

 unsigned char bitmap_mask[8];
 bitmap_mask[0] = 0x01;
 bitmap_mask[1] = 0x02;
 bitmap_mask[2] = 0x04;
 bitmap_mask[3] = 0x08;
 bitmap_mask[4] = 0x10;
 bitmap_mask[5] = 0x20;
 bitmap_mask[6] = 0x40;
 bitmap_mask[7] = 0x80;


 if( (outfile = fopen(filename, "w")) == NULL) {
  printf("fwrite_bitmap:    Couldn't open file %s for writing\n", filename);
  exit(0);
 }

 fprintf(outfile, "#define %s_width %d\n", filename, width);
 fprintf(outfile, "#define %s_height %d\n", filename. height);
 fprintf(outfile, "static unsigned char %s_bits[] = {\n", filename);

 for(count1=0;count1<height;count1++) {
  for(count2=0;count2<width;count2++) {
   if( (!(count2%8) && count2!=0) || count2==(width-1)) {
       if( count1==(height-1) && count2==(width-1) ) {
         fprintf(outfile, "%s};\n", uchar2hexstring(outval));
       }
       else {
         fprintf(outfile, "%s,\n", uchar2hexstring(outval));
       }
       outval = (unsigned char)0;
   }
   if(bitmap[count1][count2])
       outval += bitmap_mask[count2%8];
  }
```

```
  }

  fclose(outfile);
}


int fwrite_newblob(char *filename, tBITMAP checkmap, tNEW_BLOBTYPE blob,
                   int height, int width)
{
  int count, count2;                                                    140

  tBITMAP retval = BitmapCreate(height, width);

  for(count=0;count<height;count++) {
    for(count2=blob[count].start_blob;count2<=blob[count].end_blob;count2++) {
      retval[count][count2] = checkmap[count][count2];
    }
  }

  fwrite_bitmap(filename, retval, height, width);                       150

  BitmapDestroy(retval, height);
}


int get_next_bit(unsigned char *data)
{
  static int byte_offset=-1;
  static int bit_offset=-1;
                                                                        160
  unsigned char bitmap_mask[8];
  bitmap_mask[0] = 0x80;
  bitmap_mask[1] = 0x40;
  bitmap_mask[2] = 0x20;
  bitmap_mask[3] = 0x10;
  bitmap_mask[4] = 0x08;
  bitmap_mask[5] = 0x04;
  bitmap_mask[6] = 0x02;
  bitmap_mask[7] = 0x01;
                                                                        170

  if( data == (void *)-1 ) {
    byte_offset=-1;
    bit_offset=-1;
    return 0:
  }
  bit_offset++;
  if(bit_offset==8)
    bit_offset=0;
  if(bit_offset==0)                                                     180
    byte_offset++;

  return (data[byte_offset]&bitmap_mask[bit_offset]);
}
```

```
char *uchar2hexstring(unsigned char bits) {
 switch (bits)
  {
  case 0:                                                    190
   return "0x00";
  case 1:
   return "0x01";
  case 2:
   return "0x02";
  case 3:
   return "0x03";
  case 4:
   return "0x04";
  case 5:                                                    200
   return "0x05";
  case 6:
   return "0x06";
  case 7:
   return "0x07";
  case 8:
   return "0x08";
  case 9:
   return "0x09";
  case 10:                                                   210
   return "0x0a";
  case 11:
   return "0x0b";
  case 12:
   return "0x0c";
  case 13:
   return "0x0d";
  case 14:
   return "0x0e";
  case 15:                                                   220
   return "0x0f";
  case 16:
   return "0x10";
  case 17:
   return "0x11";
  case 18:
   return "0x12";
  case 19:
   return "0x13";
  case 20:                                                   230
   return "0x14";
  case 21:
   return "0x15";
  case 22:
   return "0x16";
  case 23:
   return "0x17";
  case 24:
```

```
  return "0x18";
case 25:
  return "0x19";
case 26:
  return "0x1a";
case 27:
  return "0x1b";
case 28:
  return "0x1c";
case 29:
  return "0x1d";
case 30:
  return "0x1e";
case 31:
  return "0x1f";
case 32:
  return "0x20";
case 33:
  return "0x21";
case 34:
  return "0x22";
case 35:
  return "0x23";
case 36:
  return "0x24";
case 37:
  return "0x25";
case 38:
  return "0x26";
case 39:
  return "0x27";
case 40:
  return "0x28";
case 41:
  return "0x29";
case 42:
  return "0x2a";
case 43:
  return "0x2b";
case 44:
  return "0x2c";
case 45:
  return "0x2d";
case 46:
  return "0x2e";
case 47:
  return "0x2f";
case 48:
  return "0x30";
case 49:
  return "0x31";
case 50:
  return "0x32";
case 51:
```

```
  return "0x33";
case 52:
  return "0x34";
case 53:
  return "0x35";
case 54:
  return "0x36";
case 55:
  return "0x37";
case 56:
  return "0x38";
case 57:
  return "0x39";
case 58:
  return "0x3a";
case 59:
  return "0x3b";
case 60:
  return "0x3c";
case 61:
  return "0x3d";
case 62:
  return "0x3e";
case 63:
  return "0x3f";
case 64:
  return "0x40";
case 65:
  return "0x41";
case 66:
  return "0x42";
case 67:
  return "0x43";
case 68:
  return "0x44";
case 69:
  return "0x45";
case 70:
  return "0x46";
case 71:
  return "0x47";
case 72:
  return "0x48";
case 73:
  return "0x49";
case 74:
  return "0x4a";
case 75:
  return "0x4b";
case 76:
  return "0x4c";
case 77:
  return "0x4d";
case 78:
```

```
  return "0x4e";
case 79:
  return "0x4f";
case 80:
  return "0x50";
case 81:
  return "0x51";
case 82:
  return "0x52";
case 83:
  return "0x53";
case 84:
  return "0x54";
case 85:
  return "0x55";
case 86:
  return "0x56";
case 87:
  return "0x57";
case 88:
  return "0x58";
case 89:
  return "0x59";
case 90:
  return "0x5a";
case 91:
  return "0x5b";
case 92:
  return "0x5c";
case 93:
  return "0x5d";
case 94:
  return "0x5e";
case 95:
  return "0x5f";
case 96:
  return "0x60";
case 97:
  return "0x61";
case 98:
  return "0x62";
case 99:
  return "0x63";
case 100:
  return "0x64";
case 101:
  return "0x65";
case 102:
  return "0x66";
case 103:
  return "0x67";
case 104:
  return "0x68";
case 105:
```

```
  return "0x69";
case 106:
  return "0x6a";
case 107:
  return "0x6b";
case 108:
  return "0x6c";
case 109:
  return "0x6d";
case 110:
  return "0x6e";
case 111:
  return "0x6f";
case 112:
  return "0x70";
case 113:
  return "0x71";
case 114:
  return "0x72";
case 115:
  return "0x73";
case 116:
  return "0x74";
case 117:
  return "0x75";
case 118:
  return "0x76";
case 119:
  return "0x77";
case 120:
  return "0x78";
case 121:
  return "0x79";
case 122:
  return "0x7a";
case 123:
  return "0x7b";
case 124:
  return "0x7c";
case 125:
  return "0x7d";
case 126:
  return "0x7e";
case 127:
  return "0x7f";
case 128:
  return "0x80";
case 129:
  return "0x81";
case 130:
  return "0x82";
case 131:
  return "0x83";
case 132:
```

```
  return "0x84";
case 133:
  return "0x85";
case 134:
  return "0x86";
case 135:
  return "0x87";
case 136:
  return "0x88";
case 137:
  return "0x89";
case 138:
  return "0x8a";
case 139:
  return "0x8b";
case 140:
  return "0x8c";
case 141:
  return "0x8d";
case 142:
  return "0x8e";
case 143:
  return "0x8f";
case 144:
  return "0x90";
case 145:
  return "0x91";
case 146:
  return "0x92";
case 147:
  return "0x93";
case 148:
  return "0x94";
case 149:
  return "0x95";
case 150:
  return "0x96";
case 151:
  return "0x97";
case 152:
  return "0x98";
case 153:
  return "0x99";
case 154:
  return "0x9a";
case 155:
  return "0x9b";
case 156:
  return "0x9c";
case 157:
  return "0x9d";
case 158:
  return "0x9e";
case 159:
```

```
  return "0x9f";
case 160:
  return "0xa0";
case 161:
  return "0xa1";
case 162:
  return "0xa2";
case 163:
  return "0xa3";
case 164:
  return "0xa4";
case 165:
  return "0xa5";
case 166:
  return "0xa6";
case 167:
  return "0xa7";
case 168:
  return "0xa8";
case 169:
  return "0xa9";
case 170:
  return "0xaa";
case 171:
  return "0xab";
case 172:
  return "0xac";
case 173:
  return "0xad";
case 174:
  return "0xae";
case 175:
  return "0xaf";
case 176:
  return "0xb0";
case 177:
  return "0xb1";
case 178:
  return "0xb2";
case 179:
  return "0xb3";
case 180:
  return "0xb4";
case 181:
  return "0xb5";
case 182:
  return "0xb6";
case 183:
  return "0xb7";
case 184:
  return "0xb8";
case 185:
  return "0xb9";
case 186:
```

```
  return "0xba";
case 187:
  return "0xbb";
case 188:
  return "0xbc";
case 189:
  return "0xbd";
case 190:
  return "0xbe";
case 191:
  return "0xbf";
case 192:
  return "0xc0";
case 193:
  return "0xc1";
case 194:
  return "0xc2";
case 195:
  return "0xc3";
case 196:
  return "0xc4";
case 197:
  return "0xc5";
case 198:
  return "0xc6";
case 199:
  return "0xc7";
case 200:
  return "0xc8";
case 201:
  return "0xc9";
case 202:
  return "0xca";
case 203:
  return "0xcb";
case 204:
  return "0xcc";
case 205:
  return "0xcd";
case 206:
  return "0xce";
case 207:
  return "0xcf";
case 208:
  return "0xd0";
case 209:
  return "0xd1";
case 210:
  return "0xd2";
case 211:
  return "0xd3";
case 212:
  return "0xd4";
case 213:
```

```
  return "0xd5";
case 214:
  return "0xd6";
case 215:
  return "0xd7";
case 216:
  return "0xd8";
case 217:
  return "0xd9";
case 218:
  return "0xda";
case 219:
  return "0xdb";
case 220:
  return "0xdc";
case 221:
  return "0xdd";
case 222:
  return "0xde";
case 223:
  return "0xdf";
case 224:
  return "0xe0";
case 225:
  return "0xe1";
case 226:
  return "0xe2";
case 227:
  return "0xe3";
case 228:
  return "0xe4";
case 229:
  return "0xe5";
case 230:
  return "0xe6";
case 231:
  return "0xe7";
case 232:
  return "0xe8";
case 233:
  return "0xe9";
case 234:
  return "0xea";
case 235:
  return "0xeb";
case 236:
  return "0xec";
case 237:
  return "0xed";
case 238:
  return "0xee";
case 239:
  return "0xef";
case 240:
```

```
      return "0xf0";
    case 241:
      return "0xf1";
    case 242:
      return "0xf2";
    case 243:
      return "0xf3";
    case 244:
      return "0xf4";
    case 245:                                                    680
      return "0xf5";
    case 246:
      return "0xf6";
    case 247:
      return "0xf7";
    case 248:
      return "0xf8";
    case 249:
      return "0xf9";
    case 250:                                                    690
      return "0xfa";
    case 251:
      return "0xfb";
    case 252:
      return "0xfc";
    case 253:
      return "0xfd";
    case 254:
      return "0xfe";
    case 255:                                                    700
      return "0xff";
    }
}
```