NISTIR 89-4177

DECODING BAR CODES FROM IMAGE DATA

Michael D. Garris
Charles L. Wilson

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
National Computer Science Laboratory
Advanced Systems Division
Image Recognition Group
Gaithersburg, MD 20899

Sponsored by:
Bureau of the Census
Suitland, MD 20735

September 1989

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Acting Director

NISTIR 89-4177

DECODING BAR CODES FROM IMAGE DATA

Michael D. Garris
Charles L. Wilson

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Science Laboratory
Advanced Systems Division
Image Recognition Group
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Acting Director

NIST

Decoding Bar Codes from Image Data

National Institute of Standards and Technology
Advanced Systems Division
July 21, 1989

Michael D. Garris
Charles L. Wilson

1. Introduction

Direct image storage technology promises to provide massive improvements in the transmission and storage of forms. Many existing forms use bar codes to provide machine readable indexing information. While it is possible to provide separate bar code readers to index these forms, direct decoding of the bar code in a scanned image can simplify paper handling and systems design. This software was designed to provide portable 'C' programs which demonstrate the feasibility to directly decode bar codes from raster images.

2. Factors Affecting Decoding of Bar Codes from Images

The two principal difficulties in direct image decoding of bar codes are image quality and decoding speed. The bar codes used in the development of these algorithms contain 14 digits of data which would be decoded and transmitted as serial ASCII by a typical attached reader at 1200 baud in approximately 0.093 seconds with transmission being the time limiting factor. Software decoding must occur in a comparable time.

Image quality issues arise from the basic specifications of the bar code image. The nominal width of narrow lines or spaces in bar codes can be as small as 0.19 mm wide.[1,2] This yields 1 to 2 pixel/line with 12 pixel/mm scanning. If the scanning density is decreased to save on scan time, transmission bandwidth, or image storage space, then false breaks will result from undersampling the narrow features. By the same argument, aliasing can introduce false lines.

This aspect of image degradation is particularly evident in the sample set of bar codes used in this project. These bar codes were generated by a dot matrix printer so that each wide bar is comprised of 3 concatenated narrow lines. The separation of the 3 component lines is noticeable by humans, and in some samples, the separation is large enough to be replicated in the scanned images.

3. Basic Algorithms

Three distinct decoding algorithms were developed and tested. Initially a single scan method was developed which decodes a single scan line without any prior image processing or enhancements. This method minimizes the processing time necessary to decode a bar code, but it is susceptible to noise within the selected scan line. To ensure reliability, a global averaging method was developed. This approach takes into account information from the entire image maximizing reliability at the expense of processing time. Finally, a hybrid method was designed which samples the original bar code image into a collection of representative scan lines. These scan lines are then averaged together providing noise reduction while maintaining efficiency.

3.1 Scan Method:

The first algorithm developed at NIST simulates in software the physical action of passing a hand–held reader across a bar code. A single scan line of data from a bar code raster image has the same information as raw data captured by a conventional reader. A method was designed to accept a scan line from an image, decode it, and return the ASCII characters represented by the bar code.

To illustrate this method, a bar code in Figure 1 is shown which uses Interleaved Two of Five to encode 14 numeric digits plus stop and start characters.[3] A selected scan line is first quantized into an array of integers. One integer quantity is accumulated for each consecutive duration of on or off pixel values. The quantized integers are thresholded into binary values corresponding to wide and narrow elements in the original bar code. These element representations are then organized into character groupings according to the format specifications of the bar code and interpreted using a look–up table which maps element groupings to ASCII characters.

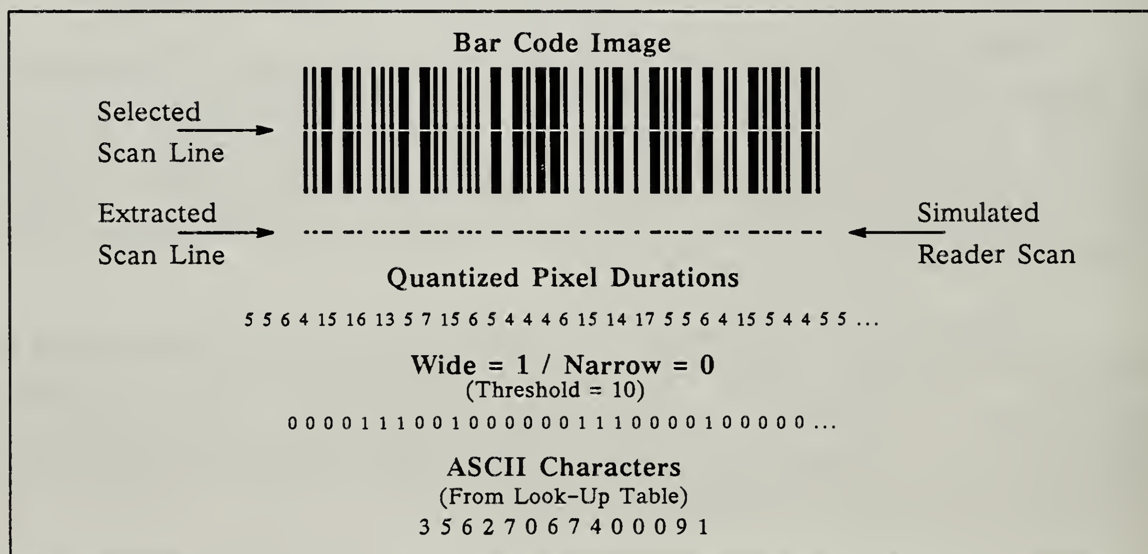

Figure 1: Illustration of Scan Method

This method implemented in software to decode images has several advantages over physical readers. The scan algorithm is resolution and size independent, whereas physical devices are typically calibrated explicitly for bar codes of uniform dimension and fixed element size. Programmed parameters can be modified in software to accurately decode images of varied resolution and size as long as the image is free of image degradation factors discussed in Section 2. Although extremely time efficient, this method is critically dependent on image quality for ensured reliability.

3.2 Averaging Method:

A second method was designed to average the image data prior to decoding in order to eliminate image degradations and thereby increase reliability. The fundamental steps described in Section 3.1 are also used in this algorithm. The difference with this method is, rather than using a raw scan line of image data for input, a histogram is first computed on the bar code image. This histogram globally averages all the scan lines from an entire image which in the experiments reported was on average 295 scan lines. By globally accumulating scan lines, missing pixels rise to the top of the histogram and extra pixels to fall to the bottom. These accumulators are thresholded producing a binary array of integers representing a clean raster scan line. Figure 2 illustrates the averaging process. Once a clean scan line representation has been produced, it can be decoded just as in Section 3.1.

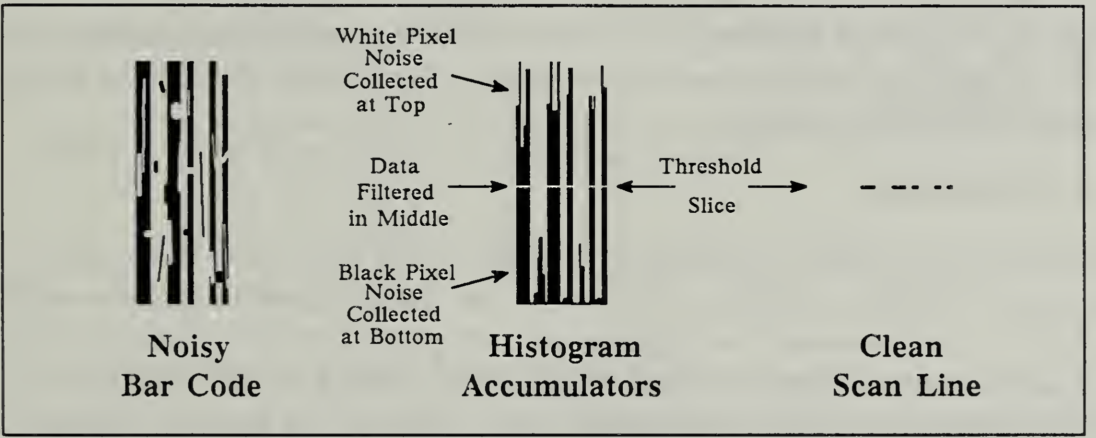

Figure 2: Illustration of Histogram Averaging

3.3 Hybrid Method:

This method combines the speed efficiency of the Scan Method with the reliability of the Averaging Method. In this algorithm, a randomly selected sample of scan lines is chosen to represent the entire bar code image. A histogram computation like the one in Section 3.2 is performed on only the sampled scan lines, and the pixel accumulators are thresholded. This process is illustrated in Figure 3. This produces an averaged scan line representation which eliminates most of the image noise while minimizing the amount of histogram computation. This method then applies the steps used in Section 3.1 to decode the averaged scan line.

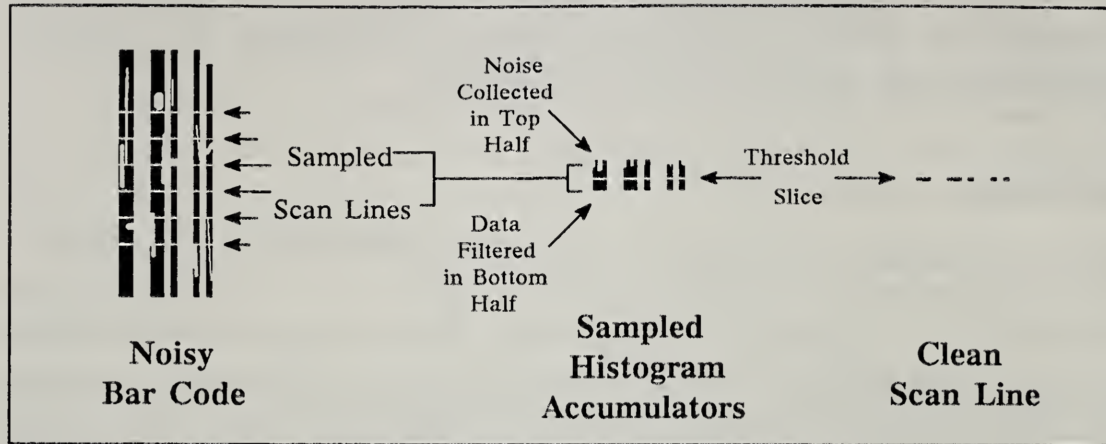

Figure 3: Illustration of Hybrid Sampling and Averaging

4. Results

Two important aspects of decoding bar codes in software were studied and used in the development of the methods in Section 3. One aspect is the measurement of computation time versus the reliability of decoding, and the second is what effects do different factors of image quality have on each method.

4.1 Speed *vs* Reliability

Experiments were implemented and executed at NIST on a Sun 3/180 Workstation to compile speed and reliability statistics for each of the decoding methods. A sample database of 108 bar code images was digitized at 300 pixels/inch on a Calera binary scanner and stored for testing. Each of these images is on average 29,578 bytes long with an average width of 794 pixels and average height of 295 pixels. On average, memory to hold the image can be allocated and the image read and loaded into the memory in 0.08 seconds.

4.1.1 Scan Method

The Scan Method decodes bar codes from raw image data without any image preprocessing or enhancements. This allows the method to capitalize on speed at the expense of reliability. An experiment was conducted whereby attempts to select a new scan line upon decode failure was measured against resulting recognition percentages. A parameter was programmed to regulate the number of retries permitted before the bar code is rejected as non-decodable.

The following results were tabulated by running the retry algorithm across the set of 108 bar codes. Timings and decode percentages were accumulated for retry parameter settings varying from 0 to 10. For each unique parameter value, 100 random number seeds where used and the Scan Method was applied to all 108 images per individual seed. The times and recognition percentages that follow are averages of all the runs executed for each of the retry parameter values. The results of this test are shown in Figure 4. The times reported are theoretically the maximum time required to successfully decode a bar code for the given number of retries permitted. These times include the system overhead to call the Scan Method function and return the decoded value, the random selection of all permitted scan lines, and the attempt to decode each of the scan lines. Image load time is not included.

# Retries	Seconds	Decodes %
0	0.0177593	94.111
1	0.0288148	98.389
2	0.0398703	99.370
3	0.0509258	99.657
4	0.0619813	99.769
5	0.0730368	99.852
6	0.0840923	99.880
7	0.0951478	99.926
8	0.1062033	99.944
9	0.1172588	99.944
10	0.1283143	99.972

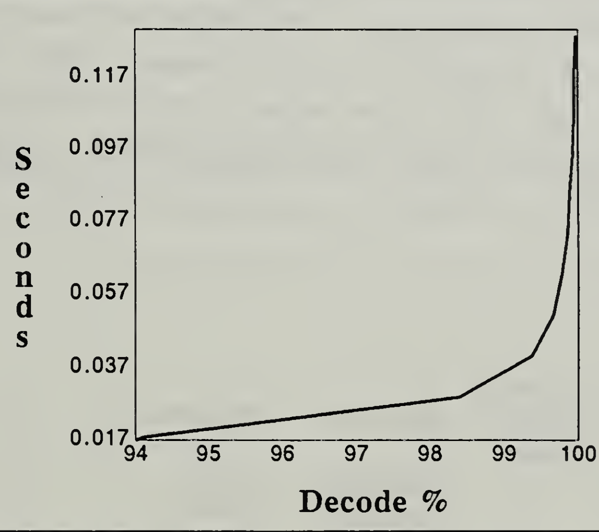

Figure 4: Scan Method Time vs Recognition Results

4.1.2 Averaging Method

The Averaging Method globally averages the entire image before decoding is attempted. This enhances the quality of the input data maximizing the percentage of successful decodings at the expense of processing time. This algorithm was tested across the sample set of 108 bar code images. Average time to process and decode an image along with the percentage of correct decodes where computed and the results are shown in Figure 5.

Average + Decode Seconds	Decode %
1.8081487	100.00

Figure 5: Averaging Method Time & Recognition

4.1.3 Hybrid Method

The Hybrid Method tries to capitalize on the strengths of the previous two methods while minimizing their weaknesses. By selecting a limited number of scan lines from the original image, a histogram average can be computed on just the sampled data. This enables preprocessing of image data before decoding while minimizing the amount of computation time needed. This method is intended to give fast and yet highly reliable bar code decodings.

Experiments which varied the number of sample scan lines to be used in the averaging process were conducted to measure both speed and reliability across the test set of 108 bar code images. Ten unique runs were executed for each scan line sample size across the complete set of 108 images. Time statistics were averaged across all runs. Twenty runs each with a unique random number seed were executed across the same set of images to compile decode percentages. The results from these tests are shown in Figure 6.

Number of Sample Lines	Seconds	Decodes %
6	0.0576228	99.815
7	0.0637267	99.907
8	0.0698306	99.815

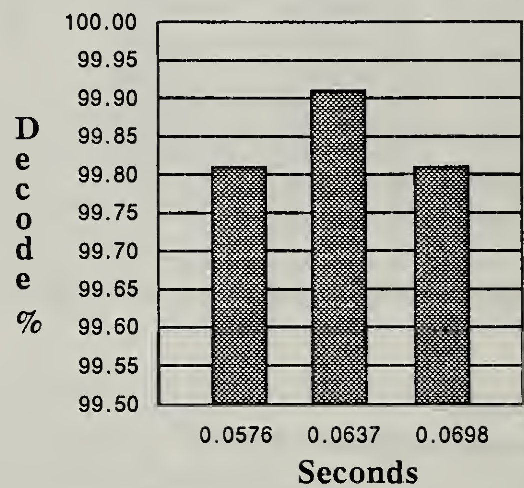

Figure 6: Hybrid Method Time vs Reliability

4.2 Image Quality

One of the significant advantages of software bar code processing is that the sensitivity of the software to image quality can be selected for the application. If the images are of uniform high quality, then the Scan Method algorithm can be used to maximize speed. If the image is of very poor quality, then the Averaging Method algorithm can be used to provide maximum tolerance to image noise. The Hybrid Method algorithm provides a continuously adjustable set of speed and noise immunity parameters.

It should be kept in mind that the test dot matrix images used in this study are, based on inspection, substantially lower in quality than images found in commercially printed bar codes.[4] It would be of interest to compare the reliability of the software methods presented here with hardware bar code readers.

5. Conclusions

A reliable method of software bar code reading has been developed at NIST. Using the Hybrid Method discussed in Section 3.3, a wide variety of speed *vs* reliability trade-offs are possible. The method is fully portable and can be incorporated in a wide range of imaging applications.

6. Acknowledgment

We wish to acknowledge B. Hammond, H. Chansky, A. E. Levin, and C. Linett of the Bureau of the Census for providing NIST the opportunity to research the problem of processing bar codes in software.

References

[1] Craig D. Harmon and Russ Adams, *Reading Between the Lines, an Introduction to Bar Code Technology*, (North American Technology, Inc., Peterborough, New Hampshire, 1984), p 19.

[2] ANSI Committee MH10.8M, "American National Standard for Material Handling - Bar Code Symbols on Unit Loads and Transport Packages," (American National Standards Institute, Inc., New York, New York, 1983), p 11.

[3] Automatic Identification Manufacturers, "Uniform Symbol Description-1 Interleaved Two of Five," (The Material Handling Institute, Inc., Pittsburgh, Pennsylvania, 1981).

[4] ANSI Committees MH10.8 and X3A1.3, "Guideline For Bar Code Print Quality," (proposed working standard, draft December, 1988), pp 16–17, 29–30.

Appendix A: Source Code for Scan Method

```
/*************************************************************/
/*        File Name: 2of5.c                                  */
/*        Package:   NIST Bar Codes                          */
/*                   Scan Method                             */
/*        Author:    Michael D. Garris                       */
/*************************************************************/
#include <stdio.h>

/* General Definitions */
#define TRUE                1
#define FALSE               0
#define BYTE_SIZE           8
#define ELEMENT_LEN         5
#define THRESH              10
#define TABLE_LEN           32
#define START_MASK          0x0000
#define START_LEN           4
#define STOP_MASK           0x0004
#define STOP_LEN            3
#define SPACE               0
#define BAR                 1
#define NARROW              0
#define WIDE                1
#define FOUND               1
#define NOT_FOUND           0
#define BADCHAR             0
#define BADSCAN             NULL

/* Function Type Declarations */
int *binarize();
int binary2char();
void collapse();
char *build_decode_string();
char *decode_2of5();
int startchar();
int stopchar();
void split();

/* ASCII Character Look-Up Table */
int table_2of5[TABLE_LEN] =
{
   '\0','\0','\0', '7','\0', '4', '0','\0',
   '\0', '2', '9','\0', '6','\0','\0','\0',
   '\0', '1', '8','\0', '5','\0','\0','\0',
    '3','\0','\0','\0','\0','\0','\0','\0'
};
```

```
/**************************************************************/
/* Decode_2of5 takes a scan line centered vertically within */
/* the input bar code image, decodes the scan line, and if  */
/* successful, returns an ASCII string.                     */
/*   expnum - fixed number of characters encoded.           */
/*   image  - stream of binary raster data.                 */
/*   width  - width in bytes of the image.                  */
/*   height - height in pixels of the image.                */
/**************************************************************/
char *decode_2of5(expnum,image,width,height)
int expnum;
unsigned char *image;
int width,height;
{
   unsigned char *line;
   int i,j;
   int *quants,nquants,*bins,nbins,*bptr;
   int *bars,nbars,*spaces,nspaces;
   char *barchars,*spacechars;
   int nbchars,nschars;
   char *code;

   /* assign pointer to center scanline in image */
   line = (unsigned char *)(image + ((height/2) * width));
   /* quantize consecutive bits of 1's or 0's */
   collapse(line,width,&quants,&nquants);
   /* convert quantized values to binary using a threshold */
   bins = binarize(quants,nquants,THRESH);
   /* store address of original memory allocation */
   bptr = bins;
   /* set number of binarized values */
   nbins = nquants;
   /* synchronize binbuf in order to search for the start character */
   /* if the first element in the line is a space ... */
   if((((line[0] & 0x0080) != 0) ? BAR : SPACE) == SPACE){
      /* strip off the space and point to the first  bar */
      bins += 1;
      --nbins;
   }
   /* search for the start character and, if found, strip it off */
   if(!startchar(&bins,&nbins,START_MASK,START_LEN)){
      fprintf(stderr,"Start character not found in scan\n");
      free(bptr);
      return(BADSCAN);
   }
```

```
   /* search for the stop character */
   if(!stopchar(bins,&nbins,ELEMENT_LEN,expnum,STOP_MASK,STOP_LEN)){
      fprintf(stderr,"Stop character not found in scan\n");
      free(bptr);
      return(BADSCAN);
   }
   /* separate bar bits from interleaved space bits */
   split(bins,nbins,&bars,&nbars,&spaces,&nspaces);
   /* deallocate binarized buffer */
   free(bptr);
   /* using LUT convert bars to characters */
   if(binary2char(bars,nbars,ELEMENT_LEN,&barchars,&nbchars) == BADCHAR){
      fprintf(stderr,"Bad character found in scan\n");
      free(bars);
      free(spaces);
      free(barchars);
      return(BADSCAN);
   }
   /* using LUT convert spaces to characters */
   if(binary2char(spaces,nspaces,ELEMENT_LEN,&spacechars,&nschars)==BADCHAR){
      fprintf(stderr,"Bad character found in scan\n");
      free(bars);
      free(spaces);
      free(barchars);
      free(spacechars);
      return(BADSCAN);
   }
   /* deallocate bar buffer */
   free(bars);
   /* deallocate space buffer */
   free(spaces);
   /* if number of bar chars differ from number of space chars, ERROR */
   if(nbchars != nschars){
      fprintf(stderr,"Character Synchronization Error in scan\n");
      free(barchars);
      free(spacechars);
      return(BADSCAN);
   }
   /* construct output decode string */
   code = build_decode_string(barchars,nbchars,spacechars,nschars);
   /* deallocate bar characters buffer */
   free(barchars)
   /* deallocate space characters buffer */
   free(spacechars);
   /* return the decoded ASCII character string */
   return(code);
}
```

```
/***************************************************************/
/* Collapes takes a raster scan line and returns a list of  */
/* quantized accumulations for consecutive durations of on  */
/* or off pixels in the scan line.                           */
/*    data - binary scan line.                               */
/*    len  - width in bytes of the scan line.                */
/*    obuf - pointer to return the quantized list.           */
/*    olen - pointer to the length of the quantized list.    */
/***************************************************************/
void collapse(data,len,obuf,olen)
unsigned char *data;
int **obuf;
int len,*olen;
{
   int i,count,bit,n,j = 0,mask,curbit;

   (*obuf) = (int *)malloc(len * sizeof(int));
   count = 0;
   /* get the first bit in the buffer */
   curbit = (((data[0] & 0x0080) != 0) ? 1 : 0);
   /* for each byte in the buffer */
   for(i = 0; i < len; i++){
      /* set a mask to get bits left to right */
      mask = 0x0080;
      /* for each bit in the current byte */
      for(n = 0; n < BYTE_SIZE; n++){
         /* get the bit */
         bit = (((data[i] & mask) != 0) ? 1 : 0);
          /* if the same  bit ... */
         if(bit == curbit)
            /* bump number of same bits */
            count++;
         else {
            /* store the duration into output buffer */
            (*obuf)[j++] = count;
            /* have seen the first bit of new bits */
            count = 1;
            curbit = bit;
         }
         /* shift mask for next bit */
         mask = mask >>  1;
      }
   }
   (*obuf)[j++] = count;
   *olen = j;
}
```

```
/************************************************************/
/* Binarize takes an input list and a threshold and returns */
/* the binarized list having applied the threshold.         */
/*   qbuf   - input list.                                   */
/*   qlen   - length of the input list.                     */
/*   thresh - threshold value.                              */
/************************************************************/
int *binarize(qbuf,qlen,thresh)
int *qbuf,qlen,thresh;
{
   int i,*bins;

  bins = malloc(qlen * sizeof(int));
   for(i = 0; i < qlen; i++){
      if(qbuf[i] < thresh)
         bins[i] = 0;
      else
         bins[i] = 1;
   }
   return(bins);
}

/************************************************************/
/* Startchar searches a list of elements for a start char,  */
/* and if successful, strips the start char from the list   */
/* of elements.                                             */
/*   ibuf  - list of elements.                              */
/*   ilen  - number of elements in the list.                */
/*   sc    - start character bit pattern.                   */
/*   sclen - number of elements in the start char.          */
/************************************************************/
int startchar(ibuf,ilen,sc,sclen)
int **ibuf, *ilen;
int sc,sclen;
{
   int i = 0, j, last, t;

   /* last possible position */
   last = (*ilen) - sclen;
   /* while not at end of list ... */
   while(i < last){
      t = 0;
      /* foreach element in start char ... */
      for(j = 0; j < sclen; j++){
         /* build a potential start char */
         t += (*ibuf)[i+j];
         if(j < (sclen-1))
            t = t << 1;
      }
      /* if start char found ... */
```

```
      if(t == sc){
         /* strip start char from list */
         (*ibuf) += (i+j);
         (*ilen) -= (i+j);
         return(FOUND);
      }
      ++i;
   }
   return(NOT_FOUND);
}

/*************************************************************/
/* Stopchar searches a list of elements for a stop char,    */
/* and if successful, strips the stop char from the list    */
/* of elements.                                              */
/*   ibuf  - list of elements.                               */
/*   ilen  - number of elements in the list.                 */
/*   incr  - number of elements per character.               */
/*   expt  - number of expected characters in bar code.      */
/*   sc    - start character bit pattern.                    */
/*   sclen - number of elements in the start char.           */
/*************************************************************/
int stopchar(ibuf,ilen,incr,expt,sc,sclen)
int *ibuf, *ilen, incr, expt, sc, sclen;
{
   int i = 0, j, t, last;

   /* last position of possible characters */
   last = expt * incr;
   /* if input buffer too short ... */
   if(*ilen < (last + sclen))
      return(NOT_FOUND);
   t = 0;
   /* foreach element in the stop char ...*/
   for(j = 0; j < sclen; j++){
      /* build a potential stop char */
      t += ibuf[last+j];
      if(j < (sclen-1))
         t = t << 1;
   }
   /* if stop character found ... */
   if(t == sc){
      /* strip off the stop char */
      (*ilen) = last;
      return(FOUND);
   }

   else
      return(NOT_FOUND);
}
```

```
/****************************************************************/
/* Split divides an input list into two even length output  */
/* lists by alternatingly assigning elements.               */
/*   inbuf - input list of elements.                        */
/*   inlen - number of input elements.                      */
/*   obuf1 - output list 1.                                 */
/*   olen1 - number of elements in obuf1.                   */
/*   obuf2 - output list 2.                                 */
/*   olen2 - number of elements in obuf2.                   */
/****************************************************************/
void split(inbuf,inlen,obuf1,olen1,obuf2,olen2)
int *inbuf,inlen;
int **obuf1,*olen1;
int **obuf2,*olen2;
{
   int i, j = 0, k = 0, s = 0;

   /* allocate output lists */
   *obuf1 = (int *)malloc(((inlen/2)+1) * sizeof(int));
   *obuf2 = (int *)malloc(((inlen/2)+1) * sizeof(int));
   /* foreach element in the input list ... */
   for(i = 0; i < inlen; i++){
      /* alternate list assignments */
      if(s == 0){
         (*obuf1)[j++] = inbuf[i];
         s = 1;
      }
      else{
         (*obuf2)[k++] = inbuf[i];
         s = 0;
      }
   }
   *olen1 = j;
   *olen2 = k;
}

/****************************************************************/
/* Binary2char converts an input element list into a decoded*/
/* output list of ASCII characters.                         */
/*   buf    - list of input elements.                       */
/*   len    - number of input elements.                     */
/*   window - number of elements per character.             */
/*   cbuf   - output character buffer.                      */
/*   clen   - number of output characters.                  */
/****************************************************************/
int binary2char(buf,len,window,cbuf,clen)
int *buf,len;
int window,*clen;
char **cbuf;
{
```

```
   int i,j,num;
   int t,b = 0;

   /* how many possible characters do we have */
   num = len/window;
   /* allocate the output character buffer */
   *cbuf = (char *)malloc(num*sizeof(char));
   /* foreach expected character */
   for(i = 0; i < num; i++){
      t = 0;
      /* foreach  element in a character */
      for(j = 0; j < window; j++){
         t +=  buf[b++];
         if(j < (window - 1))
            t = t << 1;
      }
      /* address the LUT with the char's elements */
      (*cbuf)[i] = table_2of5[t];
      /* if a bad character the ERROR */
      if((*cbuf)[i] == NULL)
         return(BADCHAR);
   }
   (*clen) = i;
   return(TRUE);
}

/************************************************************/
/* Build_decode_string interleaves an input buffer of bar   */
/* characters with a buffer of space characters returning   */
/* an ASCII string of characters represented by the bar     */
/* code.                                                    */
/*   barchars   - buffer of bar characters.                 */
/*   nbchars    - number of bar characters.                 */
/*   spacechars - buffer of space characters.               */
/*   nschars    - number of space characters.               */
/************************************************************/
char *build_decode_string(barchars,nbchars,spacechars,nschars)
char *barchars,*spacechars;
int nbchars,nschars;
{
   int i,j;
   char *code;

   /* allocate output decode buffer */
   code = (char *)malloc((nbchars + nschars + 1) * sizeof(char));
   /* for each bar and space character pair ...*/
   for(i = 0,j = 0; j < nbchars; i+=2){
       /* store bar character */
       code[i] = barchars[j];
       /* store space character */
```

```
        code[i+1] = spacechars[j++];
    }
    /* Null terminates to create a string for printing */
    code[i] = NULL;
    return(code);
}
```

Appendix B: Averaging Method Source Code

```
/***************************************************************/
/*        File Name: H2of5.c                                   */
/*        Package:   NIST Bar Codes                            */
/*                   Averaging Method                          */
/*        Author:    Michael D. Garris                         */
/***************************************************************/

#define PCT_THRESH          0.50

/* Function Type Declarations */
void compute_y_hist();
void quantize();
int find_max_bin();
void rasterize();


/***************************************************************/
/* Decode_h2of5 computes a global histogram on the input       */
/* bar code image, thresholds the histogram bins, decodes      */
/* the thresholded bins, and if successful, returns an         */
/* ASCII string.                                               */
/*   expnum - fixed number of characters encoded.              */
/*   image  - stream of binary raster data.                    */
/*   width  - width in bytes of the image.                     */
/*   height - height in pixels of the image.                   */
/***************************************************************/
char *decode_h2of5(expnum,image,width,height)
int expnum;
unsigned char *image;
int width,height;
{
   int i,j;
   int *quants,nquants,*bins,nbins,*bptr;
   int *bars,nbars,*spaces,nspaces;
   char *barchars,*spacechars;
   int nbchars,nschars;
   char *code;
   int *hbins,len,max;

   /* compute y-oriented histogram on entire image */
   compute_y_hist(image,width,height,&hbins,&len);
   /* find the maximum histogram bin value */
   max = find_max_bin(hbins,len);
   /* threshold histogram bin values to 1's and 0's */
   rasterize(hbins,len,(int)(max * PCT_THRESH));
   /* quantize consecutive integers of 1's or 0's */
   quantize(hbins,len,&quants,&nquants);
```

```
/* convert quantized values to binary using a defined threshold */
bins = binarize(quants,nquants,THRESH);
/* store address of original memory allocation */
bptr = bins;
/* set number of binarized values */
nbins = nquants;
/* deallocate quants buffer */
free(quants);
/* synchronize binarized buffer in order to search for */
/* the start character */
if(hbins[0] == SPACE){
   /* strip off the space and point to the first  bar */
   bins += 1;
   --nbins;
}
/* deallocate  istogram buffer */
free(hbins);
/* search for the start character and, if found, strip it off */
if(!startchar(&bins,&nbins,ELEMENT_LEN,expnum,START_MASK,START_LEN)){
   fprintf(stderr,"Start character not found in scan\n");
   free(bptr);
   return(BADSCAN);
}
/* search for the stop character */
if(!stopchar(bins,&nbins,ELEMENT_LEN,expnum,STOP_MASK,STOP_LEN)){
   fprintf(stderr,"Stop character not found in scan\n");
   free(bptr);
   return(BADSCAN);
}
/* separate bar elements from interleaved space elements */
split(bins,nbins,&bars,&nbars,&spaces,&nspaces);
/* deallocate binarized buffer */
free(bptr);
/* use LUT to convert bar elements to characters */
if(binary2char(bars,nbars,ELEMENT_LEN,&barchars,&nbchars) == BADCHAR){
   fprintf(stderr,"Bad character found in scan\n");
   free(bars);
   free(spaces);
   free(barchars);
   return(BADSCAN);
}
/* use LUT to convert space elements to characters */
if(binary2char(spaces,nspaces,ELEMENT_LEN,&spacechars,&nschars)==BADCHAR){
   fprintf(stderr,"Bad character found in scan\n");
   free(bars);
   free(spaces);
   free(barchars);
   free(spacechars);
   return(BADSCAN);
}
```

```
   /* deallocate bars buffer */
   free(bars);
   /* deallocate spaces buffer */
   free(spaces);
   /* if number of bar chars differs from number of space chars, ERROR */
   if(nbchars != nschars){
      fprintf(stderr,"Character Synchronization Error in scan\n");
      free(barchars);
      free(spacechars);
      return(BADSCAN);
   }
   /* construct output decode string */
   code = build_decode_string(barchars,nbchars,spacechars,nschars);
   free(barchars);
   free(spacechars);
   return(code);
}

/****************************************************************/
/* Compute_y_hist computes a Y-oriented histogram on the       */
/* input image with the parameters passed.                      */
/*    data   - raster image data.                               */
/*    width  - width of image in bytes.                         */
/*    height - height of image in pixels.                       */
/*    bins   - histogram accumulation bins.                     */
/*    len    - number of accumulators.                          */
/****************************************************************/
void compute_y_hist(data,width,height,bins,len)
unsigned char *data;
int width,height,**bins,*len;
{
   int num,y,x_byte,i,bnum,mask;
   unsigned char s;

   /* compute pixel width of image */
   num = width * BYTE_SIZE;
   /* allocate the list of accumulators */
   (*bins) = (int *)malloc(num * sizeof(int));
   /* initialize the accumulators to 0 */
   for(i = 0; i < num; i++)
      (*bins)[i] = 0;
   /* foreach scan line in the image ... */
   for(y = 0; y < height; y++){
      bnum = 0;
      /* foreach byte in the current scan line */
      for(x_byte = 0; x_byte < width; x_byte++){
         mask = 0x0080;
         s = *(data + ((y * width) + x_byte));
         /* foreach bit in the current byte ... */
         for(i = 0; i < BYTE_SIZE; i++){
```

```
            /* accumulate the bit */
            if ((s & mask) != 0)
               ((*bins)[bnum])++;
            mask = mask >> 1;
            bnum++;
         }
      }
   }
   *len = num;
}

/***************************************************************/
/* Find_max_bin returns the maximum value from the list of    */
/* integers passed.                                           */
/*    bins - list of integer accumulators.                    */
/*    len  - number of accumulators.                          */
/***************************************************************/
int find_max_bin(bins,len)
int *bins,len;
{
    int i,max;

    max = bins[0];
    for (i = 1; i < len; i++) {
       if (bins[i] > max) {
          max = bins[i];
       }
    }
    return(max);
}

/**************************************************************/
/* Rasterize takes an input list of integers and given a     */
/* threshold and returns the list binarized.                 */
/*   bins   - input integer list.                            */
/*   len    - length of the input list.                      */
/*   thresh - threshold value.                               */
/**************************************************************/
void rasterize(bins,len,thresh)
int *bins,len,thresh;
{
    int i;

    for (i = 0; i < len; i++) {
       if (bins[i] < thresh)
          bins[i] = 0;
       else
          bins[i] = 1;
    }
}
```

```
/**************************************************************/
/* Quantize takes a binarized list of integers and returns  */
/* a list of quantized accumulations for consecutive        */
/* durations of on or off integers in the input list.       */
/*    data - binary scan line.                              */
/*    len  - width in bytes of the scan line.               */
/*    obuf - pointer to return the quantized list.          */
/*    olen - pointer to the length of the quantized list.   */
/**************************************************************/
void quantize(data,len,obuf,olen)
int *data;
int **obuf;
int len,*olen;
{
   int i,count,curint,j = 0;

   /* allocate the output list */
   (*obuf) = (int *)malloc(len * sizeof(int));
   count = 0;
   /* get the first int in the buffer */
   curint = data[0];
   /* for each int in the buffer */
   for(i = 0; i < len; i++){
      /* if the same int ... */
      if(data[i] == curint)
         /* bump number of same ints */
         count++;
      else {
         /* store the duration into output buffer */
         (*obuf)[j++] = count;
         /* have seen the first int of new bits */
         count = 1;
         curint = data[i];
      }
   }
   (*obuf)[j++] = count;
   *olen = j;
}
```

Appendix C: Hybrid Method Source Code

```
/************************************************************/
/*         File Name: R2of5.c                               */
/*         Package:   NIST Bar Codes                        */
/*                    Hybrid Method                         */
/*         Author:    Michael D. Garris                     */
/************************************************************/

#define NUM_LINES            6
#define MAXRAND              2147483647
#define RANGE                150

void accum_y_hist();
int select_scanline();


/************************************************************/
/* Decode_r2of5 randomly selects a predefined number of     */
/* scan-lines to represent the bar code image and then      */
/* computes a vertical histogram on the sample represen-    */
/* tation.  The routine thresholds the accumulated pixel    */
/* bins from the histogram, averaging out any noise in      */
/* the original image data.  An array of quantized dur -    */
/* ations of on and off pixel values is then interpreted    */
/* as the bar code and an ASCII string of characters        */
/* represented by the original bar code is returned.        */
/*   expnum - fixed number of characters encoded.           */
/*   image  - stream of binary raster data.                 */
/*   width  - width in bytes of the image.                  */
/*   height - height in pixels of the image.                */
/************************************************************/
char *decode_r2of5(expnum,image,width,height)
int expnum;
unsigned char *image;
int width,height;
{
   unsigned char *line;
   int i,j;
   int *quants,nquants,*bins,nbins,*bptr;
   int *bars,nbars,*spaces,nspaces;
   char *barchars,*spacechars;
   int nbchars,nschars;
   char *code;
   int *hbins,len,max,rnd,nline;

   /* compute length of histogram bin list */
   len = width * BYTE_SIZE;
   /* allocate histogram bin list */
```

```
hbins = (int *)malloc(len * sizeof(int));
/* initialize histogram bins to zero */
for (i = 0; i < len; i++)
    hbins[i] = 0;
/* foreach scanline to be used in the histogram ... */
for (i = 0; i < NUM_LINES; i++) {
   /* randomly select a scanline */
   nline = select_scanline(height,RANGE);
   /* assign pointer to center scanline in image */
   line = (unsigned char *)(image + (nline * width));
   /* accumulate into histogram bins the current scanline */
   accum_y_hist(line,width,1,hbins,len);
}
/* find the maximum histogram bin value */
max = find_max_bin(hbins,len);
/* threshold histogram bin values to 1's and 0's */
rasterize(hbins,len,(int)(max * PCT_THRESH));
/* quantizes consecutive integers of 1's or 0's */
quantize(hbins,len,&quants,&nquants);
/* convert quantized values to binary using a defined threshold */
bins = binarize(quants,nquants,THRESH);
/* store address of original memory allocation */
bptr = bins;
/* set number of binarized values */
nbins = nquants;
/* deallocate quants buffer */
free(quants);
/* synchronize binarized buffer in order to search for */
/* the start character */
if(hbins[0] == SPACE){
   /* strip off the space and point to the first  bar */
   bins += 1;
   --nbins;
}
/* deallocate histogram buffer */
free(hbins);
/* search for the start character and, if found, strip it off */
if(!startchar(&bins,&nbins,ELEMENT_LEN,expnum,START_MASK,START_LEN)){
   fprintf(stderr,"Start character not found in scan\n");
   free(bptr);
   return(BADSCAN);
}
/* search for the stop character */
if(!stopchar(bins,&nbins,ELEMENT_LEN,expnum,STOP_MASK,STOP_LEN)){
   fprintf(stderr,"Stop character not found in scan\n");
   free(bptr);
   return(BADSCAN);
}
/* separate bar elements from interleaved space elements */
split(bins,nbins,&bars,&nbars,&spaces,&nspaces);
```

```
   /* deallocate binarized buffer */
   free(bptr);
   /* use LUT to convert bar elements to characters */
   if(binary2char(bars,nbars,ELEMENT_LEN,&barchars,&nbchars) == BADCHAR){
      fprintf(stderr,"Bad character found in scan\n");
      free(bars);
      free(spaces);
      free(barchars);
      return(BADSCAN);
   }
   /* use LUT to convert space elements to characters */
   if(binary2char(spaces,nspaces,ELEMENT_LEN,&spacechars,&nschars)==BADCHAR){
      fprintf(stderr,"Bad character found in scan\n");
      free(bars);
      free(spaces);
      free(barchars);
      free(spacechars);
      return(BADSCAN);
   }
   /* deallocate bars buffer */
   free(bars);
   /* deallocate spaces buffer */
   free(spaces);
   /* if number of bar chars differs from number of space chars, ERROR */
   if(nbchars != nschars){
      fprintf(stderr,"Character Synchronization Error in scan\n");
      free(barchars);
      free(spacechars);
      return(BADSCAN);
   }
   /* construct output decode string */
   code = build_decode_string(barchars,nbchars,spacechars,nschars);
   free(barchars);
   free(spacechars);
   return(code);
}

/************************************************************/
/* Select_scanline randomly returns a scanline index within */
/* a given range.                                           */
/*    height - of the raster image in pixels.               */
/*    range - range from which to choose scanlines.         */
/************************************************************/
int select_scanline(height,range)
int height, range;
{
   int rnd, nline;

   rnd = (rand() % range);
   if (rnd < (range>>1))
```

```
      nline = (height>>1) - rnd;
   else
      nline = (height>>1) + (rnd - (range>>1));
   return(nline);
}

/*************************************************************/
/* Accum_y_hist incrementally accumulates a histogram upon   */
/* successive calls given image data defined by the para-    */
/* meters passed.                                            */
/*    data   - raster image data.                            */
/*    width  - byte width of image.                          */
/*    height - pixel height of image.                        */
/*    bins   - output accumulators.                          */
/*    len    - number of output accumulators.                */
/*************************************************************/
void accum_y_hist(data,width,height,bins,len)
unsigned char *data;
int width,height,*bins,len;
{
   int num,y,x_byte,i,bnum,mask;
   unsigned char s;

   /* foreach scanline ... */
   for(y = 0; y < height; y++){
      bnum = 0;
      /* foreach byte in current scanline ... */
      for(x_byte = 0; x_byte < width; x_byte++){
         mask = 0x0080;
         s = *(data + ((y * width) + x_byte));
         /* foreach bit in the current byte ... */
         for(i = 0; i < BYTE_SIZE; i++){
            /* accumulate the bit */
            if ((s & mask) != 0)
               (bins[bnum])++;
            mask = mask >> 1;
            bnum++;
         }
      }
   }
}
```

NIST-114A (REV. 3-89)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 89-4177

2. PERFORMING ORGANIZATION REPORT NUMBE
SEPTEMBER 1989

3. PUBLICATION DATE

4. TITLE AND SUBTITLE

Decoding Bar Codes from Image Data

5. AUTHOR(S)

Michael D. Garris and Charles L. Wilson

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

Bureau of the Census
Suitland, MD 20735

10. SUPPLEMENTARY NOTES

[ ] DOCUMENT DESCRIBES A COMPUTER PROGRAM; SF-185, FIPS SOFTWARE SUMMARY, IS ATTACHED.

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

Image storage technology using direct decoding of bar codes in scanned images can simplify paper handling and improve the transmission and storage of forms. Software was developed to provide portable 'C' programs which demonstrate the feasibility to directly decode bar codes from raster images. Three distinct decoding algorithms were developed and tested:

1. A scan method was developed which decodes a single scan line without any prior image processing or enhancements. This method minimizes the processing time necessary to decode a bar code, but is susceptible to noise within the scan line.
2. A global histogram averaging method was developed to ensure reliability. This method takes into account information from the entire image maximizing reliability at the expense of processing time.
3. A hybrid method was designed which samples the original bar code into a collection of representative scan lines. These scan lines are then averaged together providing noise reduction while maintaining efficiency.

These methods are fully portable and can be incorporated in a wide range of image applications.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

averaging method; bar code; histogram; hybrid method; raster image; scan method

13. AVAILABILITY

[X] UNLIMITED
[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
28

15. PRICE
A03

ELECTRONIC FORM