NIST Form-Based Handprint Recognition System

NISTIR 5469

Michael D. Garris, James L. Blue, Gerald T. Candela, Darrin L. Dimmick, Jon Geist, Patrick J. Grother, Stanley A. Janet, and Charles L. Wilson

National Institute of Standards and Technology,
Building 225, Room A216
Gaithersburg, Maryland 20899

ACKNOWLEDGEMENTS

We would like to acknowledge the Internal Revenue Service and the Bureau of the Census who provided funding and resources in conjunction with NIST to support the development of this standard reference optical character recognition system.

TABLE OF CONTENTS

ABSTRACT **1**

1. INTRODUCTION **1**

2. INSTALLATION INSTRUCTIONS **4**

2.1 INTALLATING FROM CD-ROM 4

2.2 HIERARCHICAL DIRECTORY STRUCTURE 5

2.2.1 SOURCE CODE SUBDIRECTORY 6

2.3 AUTOMATED COMPILATION UTILITY 7

2.4 SYSTEM INVOCATION 10

3. SOFTWARE DOCUMENTATION **12**

3.1 DO HSF FORM; src/lib/hsf/form.c; do_hsf_form() 12

3.1.1 INITIALIZE FOR HSF FORM; src/lib/hsf/form.c; init_form() 12

3.1.1.1 READ FORM IMAGE; src/lib/image/readrast.c; ReadBinaryRaster() 12

3.1.1.2 READ FIELD TEMPLATE; src/lib/hsf/hsftmplt.c; read_hsftmplt() 18

3.1.2 PROCESS HSF FORM; src/lib/hsf/form.c; process_form2() 18

3.1.2.1 REGISTER FORM IMAGE; src/lib/hsf/reghsf.c; register_hsf2() 18

3.1.2.1.1 Read Reference Points; src/lib/mdg/readmfs.c;p readmfsint2() 19

3.1.2.1.2 Locate Hypothesized Points; src/lib/hsf/hsfpoint.c; hsfpoints() 20

3.1.2.1.3 Compute Distortion Parameters;
src/lib/stats/lsq3.c; chknfindparam3() 22

3.1.2.1.4 Transform Form Image;
src/lib/image/fitimage.c; f_fit_param3_image2() 24

3.1.2.2 REMOVE FORM; src/lib/hsf/rmform.c; remove_form() 25

3.1.2.2.1 Read Form Mask Image;
src/lib/image/readrast.c; ReadBinaryRaster() 25

3.1.2.2.2 Subtract Form Pixels; lib/image/binlogop.c; nandbinimage() 25

3.2 DO DIGIT FIELDS; src/lib/hsf/field.c; do_digit_fields() 25

3.2.1 INITIALIZE FOR FIELDS; src/lib/hsf/field.c; init_field() 26

3.2.1.1 READ BASIS FUNCTIONS; src/lib/nn/basis_io.c; read_basis() 26

3.2.1.2 READ PROTOTYPES & TREE; src/lib/nn/pat_io.c; readpatstreefile() 26

3.2.1.3 READ MEDIAN VECTORS; src/lib/nn/median_io.c; readmedianfile() 28

3.2.2 PROCESS DIGIT FIELD; src/lib/hsf/field.c; process_digit_field() 29

3.2.2.1 ISOLATE 1-LINE FIELD; src/lib/hsf/isolate.c; iso_1line_field() 29

3.2.2.2 SEGMENT DIGIT FIELD; src/lib/hsf/segblob.c; segbinblobdigits() 31

3.2.2.2.1 Extract Blobs; src/lib/hsf/segblob.c; segbinblob() 34

3.2.2.2.2 Paste Digit Blobs; src/lib/hsf/segblob.c; paste_digit_blobs() 35

3.2.2.3 NORMALIZE CHARACTER IMAGES; src/lib/hsf/normaliz.c; norm_2nd_gen() 35

3.2.2.4 SHEAR CHARACTER IMAGES; src/lib/hsf/shear.c; shear_mis() 36

3.2.2.5 EXTRACT FEATURES; src/lib/nn/kl_mis.c; kl_transform_mis() 37

3.2.2.6 CLASSIFY FEATURE VECTORS; src/lib/nn/pnn.c; treepnnhypsconsC() 38

3.2.3 STORE FIELD RESULTS; src/lib/fet/updatefet.c; updatefet() 41

3.2.4 DEALLOCATE FOR FIELDS; src/lib/hsf/field.c; free_field() 43

3.3 DO LOWER CASE FIELD; src/lib/hsf/field.c; do_alpha_field() 43

3.3.1 PROCESS ALPHABETIC FIELD; src/lib/hsf/field.c; process_alpha_field() 43

3.3.1.1 SEGMENT ALPHABETIC FIELD; src/lib/hsf/segblob.c; segbinblob() 43

3.4 DO UPPER CASE FIELD; src/lib/hsf/field.c; do_alpha_field() 47

3.5 DO CONSTITUTION FIELD; src/ib/hsf/field.c; do_const_field() 47

3.5.1 PROCESS CONSTITUTION FIELD; src/lib/hsf/field.c; process_const_field() 47

3.5.1.1 ISOLATE MULTIPLE LINE FIELD; src/lib/hsf/isolate.c; iso_nline_field() 47

3.5.1.2 BUILD PHRASE LISTS; src/lib/phrase/bld_pis.c; build_pi_lists() 47

3.5.1.2.1 Find Phrases; src/lib/phrase/find_pis.c; find_pi_lists() 48

3.5.1.2.2 Merge Phrases; src/lib/phrase/merg_pis.c; merge_pi_lists() 51

3.5.1.2.3 Sort Phrases Top To Bottom; src/lib/phrase/sort_pis.c; sort_pi_lists_on_y() 52

3.5.1.3 CORRECT AND IDENTITY WORDS; src/lib/phrase/spellphr.c; spell_phrases2() 52

3.5.1.3.1 Spell-Correct Line Of Text; src/lib/dict/line.c; spell_line2() 52

4. PERFORMANCE AND TIMING STATISTICS **55**

5. FINAL COMMENTS **57**

6. REFERENCES **58**

APPENDIX A. UTILITY *mis2evt* **60**

APPENDIX B. UTILITY *mis2pat* **61**

APPENDIX C. 2nd Census OCR Systems Conference **63**

NIST Form-Based Handprint Recognition System

Michael D. Garris (mdg@magi.ncsl.nist.gov)

James L. Blue, Gerald T. Candela, Darrin L. Dimmick, Jon Geist, Patrick J. Grother,
Stanley A. Janet, and Charles L. Wilson

National Institute of Standards and Technology,
Building 225, Room A216
Gaithersburg, Maryland 20899

ABSTRACT

The National Institute of Standards and Technology (NIST) has developed a standard reference form-based handprint recognition system for evaluating optical character recognition. NIST is making this recognition system freely available to the general public on CD-ROM. This is a source code distribution written primarily in C with two additional utilities having FORTRAN components. Library utilities are provided with the recognition system for conducting form registration, form removal, field isolation, field segmentation, character normalization, feature extraction, character classification, and dictionary-based postprocessing. A host of data structures and low-level utilities are also provided. These utilities include the application of spatial histograms, Least-Squares fitting, spatial zooming, connected components, Karhunen Loeve feature extraction, Probabilistic Neural Network classification, multiple-key sorting, dynamic string alignment, and dictionary matching. The recognition system has been successfully compiled and tested on a host of UNIX workstations including computers manufactured by Digital Equipment Corporation, Hewlett Packard, IBM, Silicon Graphics Incorporated, and Sun Microsystems. A CD-ROM can be obtained free of charge by sending a letter of request to NIST. This report documents the system in terms of its installation, organization, and functionality.

1. INTRODUCTION

The National Institute of Standards and Technology (NIST), has developed a standard reference form-based handprint recognition system for evaluating optical character recognition (OCR). NIST is making this recognition system freely available to the general public on CD-ROM. This report documents the system in terms of its installation, organization, and functionality. The standard reference recognition system is designed to run on UNIX workstations and has been successfully compiled and tested on a Digital Equipment Corporation (DEC) Alpha, Hewlett Packard (HP) Model 712/80, IBM RS6000, Silicon Graphics Incorporated (SGI) Indigo 2, SGI Onix, SGI Challenge, Sun Microsystems (Sun) IPC, Sun SPARCstation 2, Sun 4/470, and a Sun SPARCstation 10.[*] The standard reference recognition system runs on computers with as little as 8 Megabytes of memory, but this is not recommended as the system is computer resource intense.

The source code for the main system, *hsfsys*, is written in C (traditional K&R not ANSI) and is organized into 11 libraries. In all, there are approximately 19,000 lines of code supporting more than 550 subroutines. Source code is provided for form registration, form removal, field isolation, field segmentation, character normalization, feature extraction, character classification, and dictionary-based postprocessing. A host of data structures and low-level utilities are also provided. These utilities include the application of CCITT Group 4 decompression[1,2], IHead file manipulation[3,4], spatial histograms, Least-Squares fitting[5], spatial zooming, connected components, Karhunen Loeve feature extraction[6], Probabilistic Neural Network classification[7], multiple-key sorting, Levenstein distance dynamic string alignment[8], and dictionary matching[9].

Two other programs are provided that generate data files used by the recognition system. The first program, *mis2evt*, computes a covariance matrix and generates eigenvectors from a sample of segmented character images. The second program, *mis2pat*, produces prototype feature vectors for neural network training and classification.

[*] Specific hardware and software products identified in this paper were used in order to adequately support the development of the technology described in this document. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment identified is necessarily the best available for the purpose.

These feature vectors are computed using segmented character images and eigenvectors. Unlike the recognition system which is written entirely in C, these two programs contain FORTRAN 77 components. To support these programs, a training set of 168,365 segmented and labeled character images is provided in the distribution. About 1000 writers contributed to this training set.

A CD-ROM distribution of this standard reference system can be obtained free of charge by sending a letter of request to Michael D. Garris at the address above. Requests made by electronic mail will not be accepted. The letter, preferably on company letterhead, should identify the requesting organization or individuals. This system or any portion of this system may be used without restrictions because it was created with U.S. government funding. Redistribution of this standard reference system is strongly discouraged as any subsequent corrections or updates will be sent to registered recipients only. This software was produced by NIST, an agency of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibilities associated with its operation, modification, and maintenance.

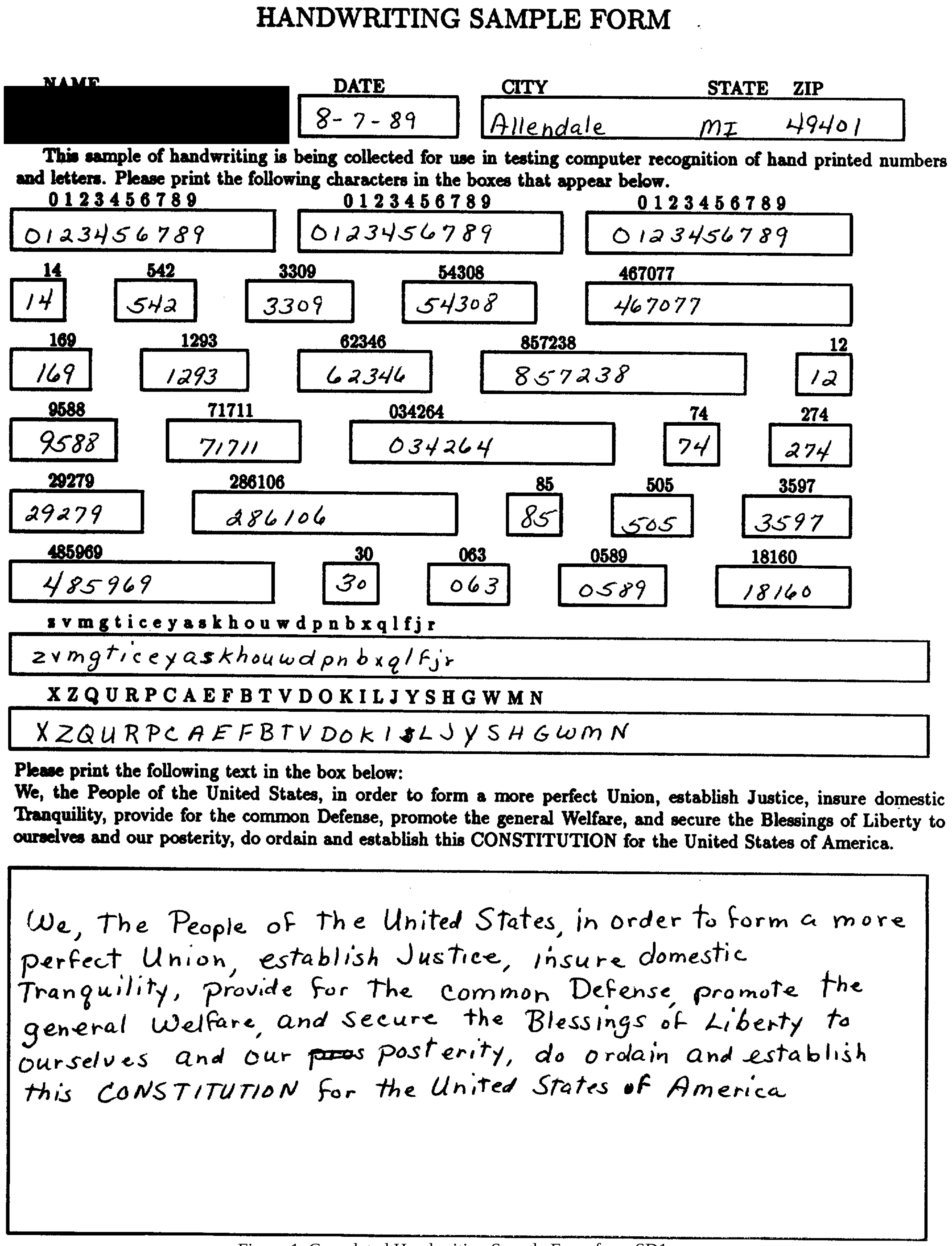
Hsfsys processes the Handwriting Sample Forms distributed with *NIST Special Database 1* (SD1)[10] and *NIST Special Database 3* (SD3)[11]. *NIST Special Database 1* contains 2,100 full page images of handwriting samples printed by 2,100 different writers geographically distributed across the United States with a sampling roughly proportional to population density. The writers used in this collection were permanent Census field representatives experienced in filling out forms. *NIST Special Database 3,* a CD-ROM containing 313,389 segmented and labeled character images, was extracted from SD1 and contains the original HSF forms as well. The forms were scanned at 12 pixels per millimeter (300 dots per inch - dpi) binary and contain entry fields demarcated by boxes, one box for the entire field value.

Each of the 2,100 forms in SD1 is an image of a structured form filled in by a unique writer. A single field template specifying the number of entry fields, their size and location, was used. An image of a completed form is shown in Figure 1. The form is comprised of 3 identification boxes, 28 digit boxes of varying length, a randomly ordered lower case alphabet, a randomly ordered upper case alphabet, and a handprinted text paragraph containing the Preamble to the U.S. Constitution. Notice that the first field, the name field, has been covered with black pixels making the writer of each form anonymous. *Hsfsys* has been designed to read all but the first 3 identification fields.

There are 10 HSF forms provided with this distribution. In addition, there is one blank form provided both in Latex and PostScript formats that can be printed, filled in, scanned, and then recognized by *hsfsys*. For additional HSF forms, SD1 and SD3 may be purchased by contacting:

Standard Reference Data
NIST
221/A323
Gaithersburg, MD 20899
voice: (301) 975-2208
FAX: (301) 926-0416
email: srdata@enh.nist.gov

Section 2 gives installation instructions and discusses the organization, compilation, and invocation of the *hsfsys* system. Section 3 documents the functionality of the provided software. Section 4 presents some performance and timing results, and Section 5 contains a few final comments. This report also has three Appendices. Appendix A documents the invocation and functionality of *mis2evt*, while Appendix B documents the invocation and functionality of *mis2pat*. Appendix C compares the NIST standard reference recognition system to the results reported from the Second Census Optical Character Recognition System Conference.[12]



HANDWRITING SAMPLE FORM

NAME [redacted]

DATE 8-7-89

CITY Allendale STATE MI ZIP 49401

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below.

0123456789	0123456789
0123456789	0123456789
0123456789	0123456789
14	14
542	542
3309	3309
54308	54308
467077	467077
169	169
1293	1293
62346	62346
857238	857238
12	12
9588	9588
71711	71711
034264	034264
74	74
274	274
29279	29279
286106	286106
85	85
505	505
3597	3597
485969	485969
30	30
063	063
0589	0589
18160	18160

z v m g t i c e y a s k h o u w d p n b x q l f j r

zvmgticeyaskhouwdpnbxqlfjr

X Z Q U R P C A E F B T V D O K I L J Y S H G W M N

XZQURPCAEFBTVDOkILJYSHGWmN

Please print the following text in the box below:
We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

We, the People of the United States, in order to form a more
perfect Union, establish Justice, insure domestic
Tranquility, provide for the common Defense, promote the
general Welfare, and secure the Blessings of Liberty to
ourselves and our ~~pros~~ posterity, do ordain and establish
this CONSTITUTION for the United States of America

Figure 1. Completed Handwriting Sample Form from SD1.

2. INSTALLATION INSTRUCTIONS

2.1 Installing from CD-ROM

Hsfsys is distributed on CD-ROM in the ISO-9660 data format.[13] This format is widely supported on UNIX workstations, DOS personal computers, and VMS computers. Therefore, the distribution can be read and downloaded onto these various platforms. Keep in mind that the source code has been developed to run on UNIX workstations. It is the responsibility of the recipient to modify the distribution source code so that it will execute on their particular computer architectures and operating systems.

Upon receiving the CD-ROM, load the disc onto your computer using a CD-ROM drive equipped with a device driver that supports the ISO-9660 data format. You may need to be assisted by your system administrator as mounting a file system usually requires root permission. Then recursively copy the disc contents into a read-writable file system. The entire distribution requires approximately 150 Megabytes upon compilation. The top-level distribution directory *doc* contains just under 72 Megabytes of PostScript reference documents. These files are not necessary to compile and run the standard reference recognition system. Therefore, they do not have to be copied off of the CD-ROM if disk space is limited on your computer, in which case, the entire distribution requires approximately 80 Megabytes upon compilation. For example, the CD-ROM can be mounted and the entire distribution copied with the following UNIX commands on a Sun SPARCstation:

```
# mount -v -t hsfs -o ro /dev/sr0 /cdrom
# mkdir /usr/local/hsfsys
# cp -r /cdrom /usr/local/hsfsys
# umount -v /cdrom
```

where */dev/sr0* is the device file associated with the CD-ROM drive, */cdrom* represents the directory to which the CD-ROM is mounted, and */usr/local/hsfsys* is the directory into which the distribution is copied. If the distribution is installed as the root user, it may be desirable to change ownership of the installation directory using the *chown* command. CD-ROM is a read-only media, so copied directories and files are likely to retain read-only permissions. The file permissions should be changed using the *chmod* command so that directories and scripts within the copied distribution are read, write, and executable. All catalog files should be changed to be read-writable. In general, source code files can remain read-only. Section 2.2 identifies the location of these various file types within the distribution. Specifically, the file *bin/catalog.csh* must be assigned executable permission, and files with the name *catalog.txt* under the top-level *src* directory must be assigned read-writable permission.

By default, the distribution assumes the installation directory to be */usr/local/hsfsys*. If this directory is used, the software can be compiled directly without any path name modifications. To minimize installation complexity, the directory */usr/local/hsfsys* should be used if at all possible. If insufficient space exists in your */usr/local* file system, the installation can be copied elsewhere and referenced through a symbolic link from */usr/local/hsfsys*.

If you decide to install this distribution in some other directory, then editing a number of source code files will be necessary prior to compiling the programs. Edit the line “PROJDIR = /usr/local/hsfsys” in the file *makefile.mak* in the top-level installation directory, replacing */usr/local/hsfsys* with the full path name of the installation directory you have chosen. Likewise replace all references to */usr/local/hsfsys* in the files *histgram.h*, *hsfsys.h*, and *invbytes.h* found in the top-level directory *include*. Remember, to make these file modifications, the permission of these files will have to be changed first. Once these edits are made, follow the instructions in Section 2.3 for compilation.

2.2 Hierarchical Directory Structure

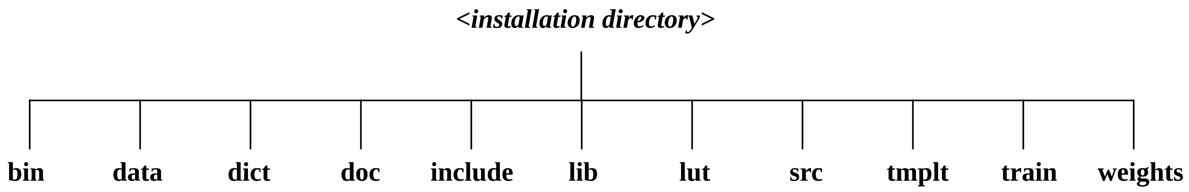


Figure 2. The top-level directory structure in the software distribution.

The top-level directories in this distribution are shown in Figure 2. The first directory *bin* holds all distributed shell scripts and locally compiled programs that support the recognition system. The full path name to this directory should be added to your environment's search path prior to compilation. Upon successful compilation, the programs *hsfsys*, *mis2evt*, and *mis2pat* are installed in the top-level *bin* directory. The invocation of *hsfsys* is discussed in Section 2.4, while the invocation of *mis2evt* and *mis2pat* are discussed in the appendices. The directory *bin* also contains the file *catalog.csh* that must be assigned executable permission. This file is a C-shell script that is used to automatically catalog programs and library routines, which is discussed in Section 2.3.

The directory *data* contains 10 subdirectories *f0000_14* through *f0009_06* containing completed forms from SD1. Each subdirectory holds the form image in an IHead format[3,4] file with extension *pct*, a reference file with extension *ref* listing the values the writer was instructed to enter in each field, and two system output files generated by *hsfsys* running at NIST.[28] The first output file, a hypothesis file, has an extension of *nhy* and lists the system's recognized values for each field. The second output file, a confidence file, has an extension of *nco* and lists the corresponding confidence values for each character classification reported in the hypothesis file. The format of these files is presented in Section 3.2.3.

The directory *dict* contains the dictionary file *const.mfs* listing in alphabetical order all the words present in the Preamble to the U.S. Constitution. The directory *include* holds all the header files that contain constants and data structure definitions required by the system source code. The directory *lib* holds all locally compiled object code libraries used in compiling the distribution programs. The directory *lut* contains two lookup tables, *bitcount.lut* and *inv_byte.lut*. The file *bitcount.lut* contains a lookup table used to determine the number of black pixels in given byte of binary image data. The file *inv_byte.lut* contains a lookup table used to reverse the bit pattern within a given byte of data. The directory *src* contains all the source code files (excluding the header files in top-level directory *include*) provided with the recognition system distribution. The organization of *src* subdirectories is discussed Section 2.2.1.

This software distribution provides a number of PostScript reference documents contained in the top-level directory *doc*. The PostScript file for this specific document is *hsfsys.ps*. All the other files in this directory are papers and reports published by NIST that are referenced within this document. These files have been assigned names according to their reference numbers listed on pages 58 and 59. All but three files in *doc* are PostScript documents ending with the extension *ps*. The files *ref_05.tar* and *ref_27.tar* were created with the UNIX *tar* command, and they contain multiple PostScript files. For example, the PostScript files contained in the file *ref_05.tar* can be extracted into the current working directory using the following command:

```
# tar xvf ref_05.tar
```

The files *ref_12_1.ps* and *ref_12_2.z* contain the Second Census Optical Character Recognition Systems Conference report. The first part is a PostScript file, whereas the second part is a UNIX compressed tar file. To extract the PostScript files archived in *ref_12_2.z*, use the following the command. Warning, extracting these files requires a large amount of disk space.

```
# zcat < ref_12_2.z | tar xvf -
```

The directory *tmplt* contains files pertaining to HSF forms. A blank HSF form is provided in both Latex and PostScript formats. The Latex file *hsf_0.tex* or the PostScript file *hsf_0.ps* can be printed, filled in, scanned at 12 pixels per millimeter (300 dpi), and then recognized by *hsfsys*. The points used to register an HSF form are stored in the file *hsfreg.pts*, and the points defining the location of each HSF entry field are stored in the file *hsftmplt.pts*. A registered blank HSF form image from which these points have been extracted is stored in the file *hsftmplt.pct*, and a dilated version of this form used in form removal is stored in the file *hsftmplt.d4*.

A large sample of training data is provided in the top-level directory *train*. As mentioned earlier, there are 168,365 segmented and labeled handprint characters contained in this directory. In all there are 119,740 images of handprint digits, 24,205 lower case letters, and 24,420 upper case letters. The handprint from about 1000 different writers are represented in this set of character images, which is divided among two subdirectories *td1* and *td3*. These two subdirectories are further subdivided into groups of 25 writers. The images of segmented characters are stored in the Multiple Image Set (MIS) file format, which was used to distribute character images in SD3.[11] Each MIS file ends with the extension *mis*. Those files beginning with *d* contain data related to handprint digits, files beginning with *l* correspond to lower case letters, and files beginning with *u* correspond to upper case letters. The four digit number embedded in each file name is an index identifying the writer. For each MIS file in the training set, there is an associated classification file containing the identity of each character contained in the MIS file. These classification files end with the extension *cls*. The first line in a classification file contains the number of character images contained in the corresponding MIS file. All subsequent lines store the identity (in hexadecimal ASCII representation) of each successive character image. MIS files containing images of lower case letters have a second classification file associated with them that ends with the extension *cus*. These files store the identity of each lower case letter as their corresponding upper case equivalent. For example, an image of a the lower case character k is stored in a *cls* file as 6b, whereas it is stored in a *cus* file as 4b (the hexadecimal ASCII representation for the upper case character K). The labelling of lower case letters as upper case is used when classifying characters in the Constitution box.

The last top-level directory *weights* holds the files associated with feature extraction and character classification. The files with the extension *evt* contain eigenvector basis functions used to compute Karhunen Loeve coefficients. The pattern (or prototype) files with the extension *pat* contain training sets of Karhunen Loeve prototype vectors and a search tree used by the Probabilistic Neural Network. Another type of file in this directory contains class-based median vectors computed from the prototypes stored in the corresponding *pat* file. Median vector files end with the extension *med*.

The *evt* files were computed using *mis2evt* discussed in Appendix A, and both the *pat* and *med* files were computed using *mis2pat* discussed in Appendix B. The files *td13_l.evt*, *td13_l.pat*, and *td13_l.med* were computed from 24,205 lower case images in both *train/td1* and *train/td3* and are used to compute features and classify lower case characters. The files *td13_u.evt*, *td13_u.pat*, and *td13_u.med* were computed from 24,420 upper case images in both *train/td1* and *train/td3* and are used to compute features and classify upper case characters. The files *td13_ul.evt*, *td13_ul.pat*, and *td13_ul.med* were computed from 48,625 images of both lower and upper case in *train/td1* and *train/td3* and are used to compute features and classify characters for lower and upper case combined. The files *td3_d.evt*, *td3_d.pat,* and *td3_d.med* were computed from 61,094 images of digits in *train/td3* and are used to compute features and classify segmented images of digits. Two additional pairs of *evt*, *pat*, and *med* files are provided so that computers with limited memory of at least 8 Megabytes are able to execute all options of the recognition system. The files *td3_ul_s.ev*t, *td3_ul_s.pat*, and *td3_ul_s.med* were computed from 24,684 images of both lower and upper case only in *train/td3*, whereas *td3_d_s.evt*, *td3_d_s.pat*, and *td3_d_s.med* were computed from 21,293 images of digits in *train/td3*. In general, recognition accuracy decreases as the number of prototypes is decreased. Therefore, the larger pattern files should be used when possible.

2.2.1 Source Code Subdirectory

The organization of subdirectories under the top-level directory *src* is shown in Figure 3. The subdirectory *src/bin* contains all program main routines. Included in this directory is a *catalog.txt* file providing a short description of each program provided in this distribution. In this distribution there are three programs and therefore three directories *src/bin/hsfsys*, *src/bin/mis2evt*, and *src/bin/mis2pat*. The first directory contains the recognition system's main routine in the file *hsfsys.c* and a number of different architecture-dependent compilation scripts used by the UNIX *make* utility. The use of the *make* utility is discussed in Section 2.3. The second directory contains the main routine and compilation scripts for the program *mis2evt*. The third directory contains the main routine and compilation scripts for the program *mis2pat*. The programs *mis2evt* and *mis2pat* have FORTRAN components, therefore their corresponding source directories also contain FORTRAN source code files ending with the extension *f*. These two directories, *src/bin/mis2evt* and *src/bin/mis2pat*, contain the only FORTRAN files in the entire distribution. If your computer does not have a FORTRAN compiler, you won't be able to compile these two supporting programs. However, all you need is a C compiler to be able to compile all the libraries in *src/lib* and run the recognition system *hsfsys*. Upon successful compilation, the directories under *src/bin* will contain compiled object files and a development copy of each program's executable file. Production copies of these programs are automatically installed in the top-level directory *bin*.

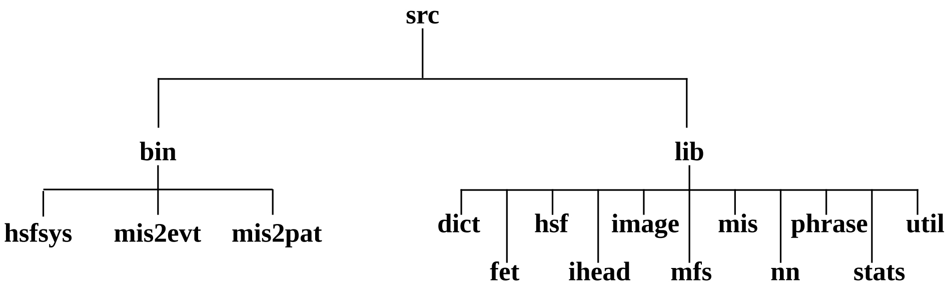


Figure 3. Directory hierarchy under the top-level directory *src*.

The subdirectory *src/lib* contains the source code for all the recognition system's supporting libraries. This distribution has 11 libraries each represented as a subdirectory under *src/lib*. Each library contains a suite of C source code files designated with the extension *c* and a set of different architecture-dependent compilation scripts designated with the root file name *makefile*. Also included in each library subdirectory is a *catalog.txt* file providing a short description of each routine contained in that specific library. Upon successful compilation, each library subdirectory under *src/lib* will contain compiled object files (with file extension *o*) and a development copy of each library's archive file (with file extension *a*). Production copies of the library archive files are automatically installed in the top-level directory *lib*.The *dict* library holds routines responsible for dictionary manipulation and matching. The *fet* library is responsible for manipulating Feature (FET) structures and files. The *hsf* library is responsible for form processing with respect to HSF forms. The *ihead* library is responsible for manipulating IHead structures and files. The *image* library contains general image manipulation and processing routines. The *mfs* library is responsible for manipulating Multiple Feature Set (MFS) structures and files. The *mis* library is responsible for manipulating Multiple Image Set (MIS) structures and files. The *nn* library contains general feature extraction and neural network routines. The *phrase* library holds routines responsible for processing the segmented text from a multiple-line field like the Constitution box on HSF forms. The *stats* library contains general statistics routines. Lastly, the *util* library contains a collection of miscellaneous routines. These various structure definitions and file formats are defined in Section 3.

2.3 Automated Compilation Utility

Before compiling the recognition system distribution, the full path name to the top-level directory *bin* in the installation directory must be added to your shell's executable search path. For example, if the distribution is installed in */usr/local/hsfsys*, your search path should be augmented to include */usr/local/hsfsys/bin*. It may also be necessary to edit the path names contained in a number of files as discussed in Section 2.1.

Source code compilation of the recognition system distribution is controlled through a system of hierarchical compilation scripts used by the UNIX *make* utility. Each one of these scripts is contained in a file with the root name *makefile*. This automated compilation system is responsible for installing all architecture-dependent source code files and compilation scripts, clearing all compiled object files and development copies of libraries and programs, automatically generating source code dependency lists, and installing production versions of libraries and programs. One *makefile.mak* file exists in the top-level installation directory, and one *makefile.mak* file exists in each of the *src, src/bin*, and *src/lib* subdirectories. These compilation scripts are architecture independent and contain Bourne shell commands.

Man.	Model	O.S.	# Proc.*	RAM	*arch*
DEC	Alpha	OSF/1 V1.3	1	32 Mb	osf
HP	Model 712/80	HP-UX 9.03	1	64 Mb	hp
IBM	RS6000	AIX 3.2.5	1	128 Mb	aix
SGI	Challenge (IP19)	IRIX 5.2	8	512 Mb	sgi
SGI	Indigo 2 (IP22)	IRIX 4.0.5H	1	128 Mb	sgi
SGI	Onyx (IP19)	IRIX 5.1.1.3	4	512 Mb	sgi
Sun	SPARCserver 4/470	SunOS 4.1.1	1	32 Mb	sun
Sun	SPARCstation IPC	SunOS 4.1.2	1	8 Mb	sun
Sun	SPARCstation 2 (Weitek 80MHz CPU)	SunOS 4.1.3	1	64 Mb	sun
Sun	SPARCstation 10	SunOS 4.1.3	1	32 Mb	sun
Sun	SPARCstation 10	SunOS 5.2 (Solaris)	2	128 Mb	sol

Figure 4. Table of different computers on which the standard reference recognition system has been successfully ported and tested, and for which architecture-dependent files are provided in the distribution.
*All computers, including those with multiple processors, were compiled and tested serially.

There are a number of architecture-dependent compilation scripts found within each program directory under *src/bin* and each library directory under *src/lib*. This standard reference recognition system has been successfully ported and tested on computers running various versions and releases of the UNIX operating system. These machines are listed in the table shown in Figure 4. The table from left to right lists each computer's manufacturer, model, operating system, number of processors, amount of main memory, and an architecture identifier. There are numerous differences between these different computers and their operating systems. Common discrepancies include differences in the syntax of compilation scripts and built in macro definitions; some operating systems require manually building the symbol table in archived library files, while other systems update these symbol tables automatically; every one of these operating systems has an *install* command, but each seems to require its own special set of arguments; finally, each manufacturer's compilers have different options and switches for controlling language syntax and optimization. To account for these variations, there are architecture-dependent compilation scripts provided for each program and library in the distribution. These compilation scripts have the root file name *makefile* and end with an extension identifying their corresponding architecture. The right column in Figure 4 lists the set of extensions used to identity architecture groups for the computers and operating systems tested.

There are also a number of architecture dependent source code files provided in the distribution. These files share the same root file name and end with an architecture-identifying extension consistent with those used for compilation scripts. Architecture-dependent source code files exist in *mis2evt* and *mis2pat* to support the calling of FORTRAN subroutines from C. Some compilers require the C-side caller to include an underscore after the FORTRAN subroutine name, whereas other compilers require no underscore be present. There are other architecture-dependent source code files provided to support DEC-like machines that use a different byte order to represent unformatted binary data. All unformatted binary data files provided in this distribution were created on machines using the Motorola-based byte order. When these files are read by a machine using an Intel-based byte order, the bytes must be swapped before the data can be used. The overhead of swapping the bytes in these data files can be avoided by regenerating them with locally compiled versions of *mis2evt* and *mis2pat* on your computer according to the instructions provided in the appendices.

It was stated earlier that the automated compilation system is responsible for installing all architecture-dependent source code files and compilation scripts, clearing all compiled object files and development copies of libraries and programs, automatically generating source code dependency lists, and installing production versions of libraries and programs. These tasks are initiated by invoking the *make* command at the top-level installation directory. All subsequent lower-level *makefile.mak* scripts are invoked automatically in a prescribed order, and the 19,000 lines of source code are automatically maintained and object files kept up to date. The *make* command can be invoked from the location of any lower-level *makefile.mak* file and thereby isolate specific portions of the source code for recompilation. However, the details of doing this are slightly involved and left to the installer to pursue on his own.

The standard reference recognition system in *src/bin/hsfsys*, and its supporting libraries under *src/lib*, are all coded in C. Two other utilities located in *src/bin/mis2evt* and *src/bin/mis2pat* have FORTRAN components that require a FORTRAN 77 compiler. The software distribution has been organized so that you can compile the recognition system even though your computer may not have an installed FORTRAN compiler. To remove *mis2evt* and *mis2pat* from the hierarchical compilation, edit the file *src/bin/makefile.mak*, removing *mis2evt* and *mis2pat* from the assignment to the variable “SUBS”.

Assuming the installation directory is */usr/local/hsfsys*, the following steps are required to compile the distribution for the first time on your computer:

```
# cd /usr/local/hsfsys
# make -f makefile.mak instarch INSTARCH=<arch>
# make -f makefile.mak bare
# make -f makefile.mak depend
# make -f makefile.mak install
```

The first *make* invocation uses the *instarch* option to install architecture-dependent files required to support the compilation and execution of the distribution’s programs and libraries. The actual architecture is defined by replacing the argument <*arch*> with one of the extensions listed in Figure 4. For example, “INSTARCH=sun” must be used to compile the distribution on computers running SunOS 4.1.?. If you are installing this software on a machine not listed in Figure 4, you first need to determine which set of architecture-dependent files is most similar to those required by your particular computer. Invoke *make* using the *instarch* option with INSTARCH set to the closest known architecture. Then, edit the resulting *makefile.mak* files in the subdirectories under *src/bin* and *src/lib* according to the requirements of your machine. One other hint, if you are compiling on a Solaris (SunOS 5.?) machine using the parallel *make* utility, you may have to add a “-R” option prior to the “-f” option for each of the *make* invocations.

The *bare* option causes the compilation scripts to remove all temporary, backup, core, and object files from the program directories in *src/bin* and the library directories in *src/lib*. The *depend* option causes the compilation scripts to automatically generate source code dependency lists and modify the *makefile.mak* files within the program and library directories. Your C compiler may not have this capability, in which case you may want to generate the dependency lists by hand. The *install* option builds source code dependency lists as needed, compiles all program and library source code files, and installs compiled libraries and programs into their corresponding production directories. Compiled libraries are installed in the installation top-level directory *lib*. Compiled programs are installed in the installation top-level directory *bin*.

One other capability, the automatic generation of catalog files, has been incorporated into the hierarchical compilation scripts. A formatted comment header is included at the top of every program and library source code file in the recognition system distribution. When the *install* option is used, the low-level *makefile.mak* files invoke the C-shell script *bin/catalog.csh*. The script *catalog.csh* extracts all source code headers associated with all the programs or a specific library in the distribution and compiles a *catalog.txt* file. A *catalog.txt* file exists in the subdirectory *src/bin*, and one *catalog.txt* file exists in each of the library directories in *src/lib*. This provides a convenient and quick reference to the source code provided in the distribution.

2.4 System Invocation

This section describes how the recognition system program *hsfsys* is invoked and controlled from the command line. Once you have successfully compiled the software distribution on your computer, the recognition system can be tested on the HSF forms provided in the top-level installation directory *data*.

The recognition system is run in batch mode with image file inputs and ASCII text file outputs, and the system contains no Graphical User Interface. The command line usage of *hsfsys* is as follows:

```
# hsfsys
Usage:
    hsfsys [options] <hsf file> <output root>
        -d                  process digit fields
        -l                  process lower case fields
        -u                  process upper case fields
        -c nodict           process Constitution field without dictionary
        -c dict             process Constitution field using dictionary
        -m                  small memory mode
        -s                  silent mode
        -v                  verbose mode
        -t                  compute and report timings
```

The command line arguments for *hsfsys* are organized into option specifications, followed by an input file name specification, and an output file name specification. The options can be subgrouped into three general types (field type options, memory control options, and message control options).

Field type options:

-d designates the processing of the digit fields on an HSF form.

-l designates the processing of the lower case field on an HSF form.

-u designates the processing of the upper case field on an HSF form.

-c designates the processing of the Constitution field on an HSF form. This option requires an argument. If the argument *nodict* is specified, then no dictionary-based postprocessing is performed and the raw character classifications and associated confidence values are reported. If the argument *dict* is specified, then dictionary-based postprocessing is performed and matched words from the dictionary are reported without any confidence values.

The options **-dluc** can be used in any combination. For example, use only the **-l** option to process the lower case field, or use only the **-d** option to process all of the digit fields. If processing both lower case and upper case fields, then specify both options **-l** and **-u** (or an equivalent syntax **-lu**). The system processes all of the fields on the form if no field type options are specified, and dictionary-based postprocessing is performed on the Constitution field by default.

Memory control options:

-m specifies the use of alternative prototype files for classification that have fewer training patterns, so that machines with limited main memory may be able to completely process all the fields on an HSF form. In general, decreasing the number of training prototypes reduces the accuracy of the recognition system's classifier. It is recommended that this option be used only when necessary.

Message control options:

- **-s** specifies that the silent mode is to be used and all messages sent to standard output and standard error are suppressed except upon the detection of a fatal internal error. Silent mode facilitates silent batch processing and overrides the verbose mode option. By default, the system posts its recognition results to standard output as each field is processed.
- **-v** specifies that the verbose mode is to be used so that messages providing a functional trace through the system are printed to standard error.
- **-t** specifies that timing data is to be collected on system functions and reported to a timing file upon system completion.

File name specifications:

- **<hsf file>** specifies the binary HSF image in IHead format that is to be read by the system.
- **<output root>** the root file name that is to be appended to the front of each output file generated by the system. Upon completion, the system will create a hypothesis file with the extension *hyp* and a confidence file with the extension *con*. If the **-t** option is specified, a timing file with the extension *tim* will also be created.

For example, to run the system in verbose mode on all the HSF fields on the form in *data/f0000_14* and store the system results in the same location with the same root name as the form, the following commands are equivalent (assuming the installation directory is */usr/local/hsfsys*). In each case, the files created by the system will be */usr/local/hsfsys/data/f0000_14/f0000_14.hyp* and */usr/local/hsfsys/data/f0000_14/f0000_14.con*.

```
# hsfsys -v /usr/local/hsfsys/data/f0000_14/f0000_14.pct /usr/local/hsfsys/data/f0000_14/f0000_14
# hsfsys -v /usr/local/hsfsys/data/f0000_14/f0000_14.{pct,}
# (cd /usr/local/hsf/data/f0000_14; hsfsys -v f0000_14.pct ./f0000_14)
```

To run the system in silent mode on only the digit and upper case fields on the same form with results including timing data all stored in */tmp* with the root name *foo*, the following command can be used. In this example, the files created by the system will be */tmp/foo.hyp*, */tmp/foo.con*, and */tmp/foo.tim*.

```
# hsfsys -stdu /usr/local/hsfsys/data/f0000_14/f0000_14.pct /tmp/foo
```

3. SOFTWARE DOCUMENTATION

This section documents the overall functionality of the *hsfsys* program. Each subsection describes one of the many steps conducted by the standard reference recognition system. Included with each subsection heading is the file and subroutine within the software distribution responsible for carrying out the steps described therein. Figures such as Figure 6 have been organized in order to provide a top-down functional road map through the source code which in turn is cross-referenced to the documentation in this section.

The main routine is located in the distribution file *src/bin/hsfsys/hsfsys.c*. Figure 5 depicts the main routine divided into five functional groups. The first group “DO HSF FORM” is responsible for processing an HSF form image, dividing the image into separate fields. To accomplish this, the HSF form has to be registered so that any distortion due to reproduction and scanning is removed. Once the image is registered, the pixel information comprising the HSF form is removed. This involves erasing the black pixels in the image that comprise the form’s boxes and instructions. The other four groups listed in Figure 5 represent field-level processing. These functions are respectively responsible for reading the values handprinted in the digit fields, lower case field, upper case field, and the Constitution box on an HSF form. The final step of writing the system results to output files is not included in the figure.

3.1 DO HSF FORM; src/lib/hsf/form.c; do_hsf_form()

Two processing steps are conducted on the input HSF form image. The HSF form is first registered and then the form itself is removed from the image. Upon completion, the handprinted characters within each field on the form are ready to be processed. Figure 6 lists the steps used to process the HSF form. The figure is divided into two parallel lists. The left list contains functional titles assigned to each step, whereas the right list provides source code references that cite the file and subroutine names within the software distribution. Both lists contain the section numbers corresponding to where each topic is discussed in this document. For example, the topic “Transform Form Image” is discussed in Section 3.1.2.1.4 and is performed by the subroutine *f_fit_param3_image2()* found in the file *src/lib/image/fitimage.c*. The overall processing of the HSF form image is divided into an initialization step and a processing step.

3.1.1 INITIALIZE FOR HSF FORM; src/lib/hsf/form.c; init_form()

Initializing the system to process an HSF form image involves reading two files, a file of an HSF form image that has been filled in and a file containing a spatial field template. The HSF form image file is specified on the command line when *hsfsys* is invoked. The field template file is defined internal to the source code and is provided in the file *tmplt/hsftmplt.pts*. The field template defines the location of each entry field on the form. The formats of these two files are discussed below.

3.1.1.1 READ FORM IMAGE; src/lib/image/readrast.c; ReadBinaryRaster()

Hsfsys expects input images to be in the IHead file format. Image file formats and effective data compression are critical to the usefulness of these types of image recognition systems. HSF form image files must be digitized in binary at 12 pixels per millimeter (300 dpi), must be 2560 pixels wide and 3300 pixels high, and can be 2-dimensionally compressed using CCITT Group 4. These are the same file format conventions used with the distribution of SD1 and SD3.

In this application, a raster image is a digital encoding of light reflected from discrete points on a scanned form. The 2-dimensional area of the form is divided into discrete locations according to the resolution of a specified grid. Each cell of this grid is represented by a single bit value 0 or 1 called a pixel; 0 represents a cell predominately white, 1 represents a cell predominately black. Pixels are scanned from the 2-dimensional sampling grid, and they are then stored as a 1-dimensional vector of bit values in raster order; left to right, top to bottom (row major). Upon scanning, certain attributes such as image width and height are required to accurately interpret the 1-dimensional vector of pixels as a 2-dimensional image.

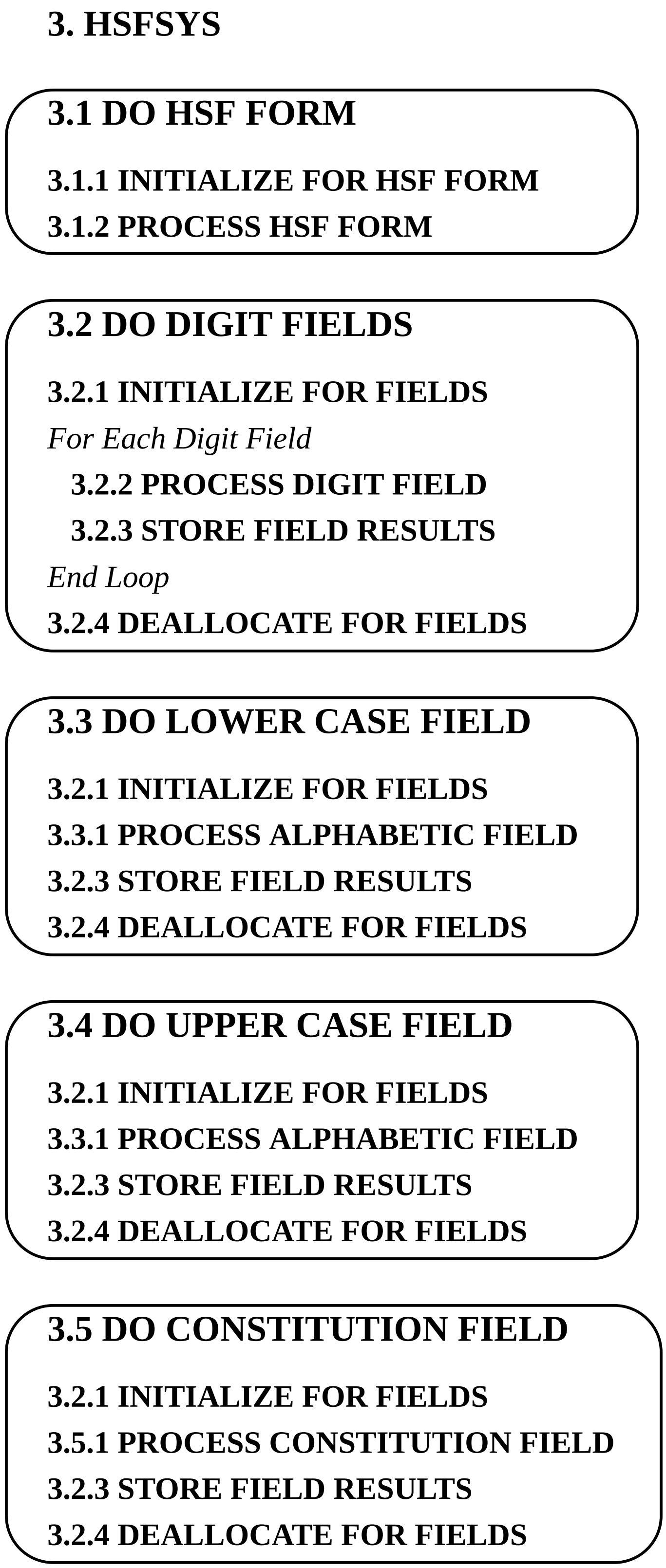

3. HSFSYS
3.1 DO HSF FORM
3.1.1 INITIALIZE FOR HSF FORM
3.1.2 PROCESS HSF FORM
3.2 DO DIGIT FIELDS
3.2.1 INITIALIZE FOR FIELDS
For Each Digit Field
3.2.2 PROCESS DIGIT FIELD
3.2.3 STORE FIELD RESULTS
End Loop
3.2.4 DEALLOCATE FOR FIELDS
3.3 DO LOWER CASE FIELD
3.2.1 INITIALIZE FOR FIELDS
3.3.1 PROCESS ALPHABETIC FIELD
3.2.3 STORE FIELD RESULTS
3.2.4 DEALLOCATE FOR FIELDS
3.4 DO UPPER CASE FIELD
3.2.1 INITIALIZE FOR FIELDS
3.3.1 PROCESS ALPHABETIC FIELD
3.2.3 STORE FIELD RESULTS
3.2.4 DEALLOCATE FOR FIELDS
3.5 DO CONSTITUTION FIELD
3.2.1 INITIALIZE FOR FIELDS
3.5.1 PROCESS CONSTITUTION FIELD
3.2.3 STORE FIELD RESULTS
3.2.4 DEALLOCATE FOR FIELDS

Figure 5. Functionality of the system’s main routine *src/bin/hsfsys/hsfsys.c*.

3.1 DO HSF FORM	**3.1 src/lib/hsf/form.c; do_hsf_form()**
3.1.1 INITIALIZE FOR HSF FORM	**3.1.1 src/lib/hsf/form.c; init_form()**
3.1.1.1 READ FORM IMAGE	3.1.1.1 src/lib/image/readrast.c; ReadBinaryRaster()
3.1.1.2 READ FIELD TEMPLATE	3.1.1.2 src/lib/hsf/hsftmplt.c; read_hsftmplt()
3.1.2 PROCESS HSF FORM	**3.1.2 src/lib/hsf/form.c; process_form2()**
3.1.2.1 REGISTER FORM IMAGE	3.1.2.1 src/lib/hsf/reghsf.c; register_hsf2()
3.1.2.1.1 Read Reference Points	3.1.2.1.1 src/lib/mfs/readmfs.c; readmfsint2()
3.1.2.1.2 Locate Hypothesized Points	3.1.2.1.2 src/lib/hsf/hsfpoint.c; hsfpoints()
3.1.2.1.3 Compute Distortion Parameters	3.1.2.1.3 src/lib/stats/lsq3.c; chknfindparam3()
3.1.2.1.4 Transform Form Image	3.1.2.1.4 src/lib/image/fitimage.c; f_fit_param3_image2()
3.1.2.2 REMOVE FORM	3.1.2.2 src/lib/hsf/rmform.c; remove_form()
3.1.2.2.1 Read Form Mask Image	3.1.2.2.1 src/lib/image/readrast.c; ReadBinaryRaster()
3.1.2.2.2 Subtract Form Pixels	3.1.2.2.2 src/lib/image/binlogop.c; nandbinimage()

Figure 6. Steps to process the HSF form.

NIST has designed a header structure called IHead to hold these attributes and has developed a file interchange format based on this header. Numerous image formats exist; some are widely supported on small personal computers, others supported on larger workstations; most are proprietary formats; few are public domain. IHead is an attempt to design an open image format which can be universally implemented across heterogeneous computer architectures and environments. IHead has been successfully ported and tested on several systems including: UNIX workstations and servers, DOS personal computers, and VMS mainframes. IHead has been designed with an extensive set of attributes in order to: adequately represent both binary and gray level images; represent images captured from different scanners and cameras; and satisfy the image requirements of diverse applications, including but not limited to, image archival/retrieval, character recognition, and fingerprint classification.

```
/************************************************************
        File Name: IHead.h
        Package:   NIST Internal Image Header
        Author:    Michael D. Garris
        Date:      2/08/90
************************************************************/
/* Defines used by the ihead structure */
#define IHDR_SIZE       288     /* len of hdr record (always even bytes) */
#define SHORT_CHARS     8       /* # of ASCII chars to represent a short */
#define BUFSIZE         80      /* default buffer size */
#define DATELEN         26      /* character length of date string */

typedef struct ihead{
  char id[BUFSIZE];                  /* identification/comment field */
  char created[DATELEN];             /* date created */
  char width[SHORT_CHARS];           /* pixel width of image */
  char height[SHORT_CHARS];          /* pixel height of image */
  char depth[SHORT_CHARS];           /* bits per pixel */
  char density[SHORT_CHARS];         /* pixels per inch */
  char compress[SHORT_CHARS];        /* compression code */
  char complen[SHORT_CHARS];         /* compressed data length */
  char align[SHORT_CHARS];           /* scanline multiple: 8|16|32 */
  char unitsize[SHORT_CHARS];        /* bit size of image memory units */
  char sigbit;                       /* 0->sigbit first | 1->sigbit last */
  char byte_order;                   /* 0->highlow | 1->lowhigh*/
  char pix_offset[SHORT_CHARS];      /* pixel column offset */
  char whitepix[SHORT_CHARS];        /* intensity of white pixel */
  char issigned;                     /* 0->unsigned data | 1->signed data */
  char rm_cm;                        /* 0->row maj | 1->column maj */
  char tb_bt;                        /* 0->top2bottom | 1->bottom2top */
  char lr_rl;                        /* 0->left2right | 1->right2left */
  char parent[BUFSIZE];              /* parent image file */
  char par_x[SHORT_CHARS];           /* from x pixel in parent */
  char par_y[SHORT_CHARS];           /* from y pixel in parent */
}IHEAD;
```

Figure 7. C structure definition for the IHead header.

The IHead structure definition written in C and stored in *include/ihead.h* is listed in Figure 7, while Figure 8 lists the header values from an IHead file corresponding to these structure members. This header information belongs to the isolated box image displayed in Figure 9 (scaled up 2X). Referencing the structure members listed in Figure 7, the first attribute field of IHead is the identification field, **id**. This field uniquely identifies the image file, typically by a file name. The attribute field, **created**, is the date on which the image was captured or digitized. The next three fields hold the image's pixel **width**, **height**, and **depth**. A binary image has a pixel depth of 1 whereas a gray scale image containing 256 possible shades of gray has a pixel depth of 8. The attribute field, **density**, contains the scan resolution of the image; in this case, 12 pixels per millimeter (300 dpi). The next two fields deal with compression.

In the recognition system distribution, input IHead images can be uncompressed or compressed using CCITT Group 4. Whether the image is compressed or not, the IHead header is always uncompressed. This enables header interpretation and manipulation without the overhead of decompression. The **compress** field is an integer flag which signifies which compression technique, if any, has been applied to the raster image data which follows the header. If the compression code is zero, then the image data is not compressed, and the data dimensions: width, height, and depth, are sufficient to load the image into main memory. However, if the compression code is nonzero, then the **complen** field must be used in addition to the image's pixel dimensions. For example, the image described in Figure 8 has a compression code of 2. By convention, this signifies that CCITT Group 4 compression has been applied to the image data prior to file creation. In order to load the compressed image data into main memory, the value in **complen** is used to load the compressed block of data into main memory. Once the compressed image data has been loaded into memory, CCITT Group 4 decompression can be used to produce an image which has the pixel dimensions consistent with those stored in its header. A compression ratio of 20 to 1 is typically achieved using CCITT Group 4 compression on the HSF form images provided in this distribution.

```
IMAGE FILE HEADER
~~~~~~~~~~~~~~~~~
Identity          : box_03.pct
Header Size       : 288 (bytes)
Date Created      : Thu Jan 4 17:34:21 1990
Width             : 656 (pixels)
Height            : 135 (pixels)
Bits per Pixel    : 1
Resolution        : 300 (ppi)
Compression       : 2 (code)
Compress Length   : 874 (bytes)
Scan Alignment    : 16 (bits)
Image Data Unit   : 16 (bits)
Byte Order        : High-Low
MSBit             : First
Column Offset     : 0 (pixels)
White Pixel       : 0
Data Units        : Unsigned
Scan Order        : Row Major,
                     Top to Bottom,
                     Left to Right
Parent            : data/f0000_14/f0000_14.pct
X Origin          : 192 (pixels)
Y Origin          : 732 (pixels)
```

Figure 8. Contents of an IHead header listed in a formatted report.

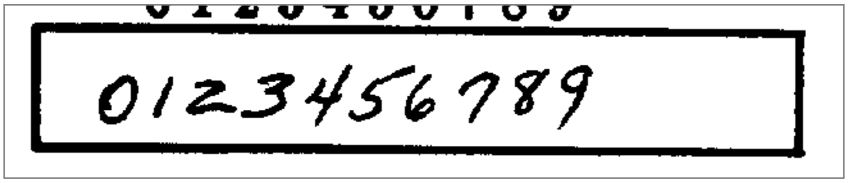


Figure 9. Image belonging to the header listed in Figure 8.

The attribute field, **align**, stores the alignment boundary to which scan lines of pixels are padded. Pixel values of binary images are stored 8 pixels (or bits) to a byte. In general, images are not an even multiple of 8 pixels in width. In order to minimize the overhead of ending a previous scan line and beginning the next scan line within the same byte, a number of padded pixels are provided in order to extend the previous scan line to an even byte boundary. Some digitizers extend this padding of pixels out to an even multiple of 8 pixels, other digitizers extend this padding of pixels out to an even multiple of 16 pixels. This field stores the image's pixel alignment value used in padding out the ends of raster scan lines.

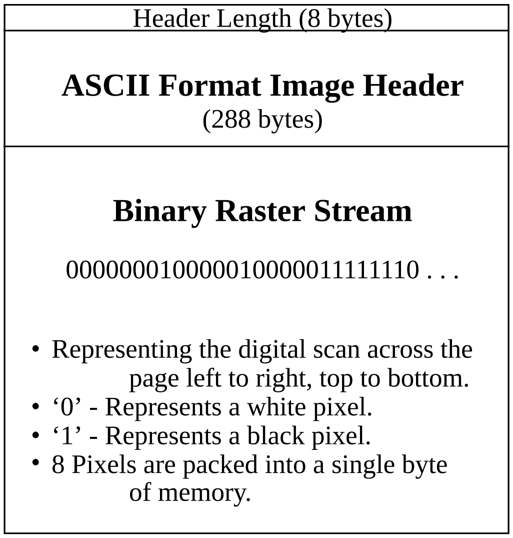
The next three attribute fields identify binary interchanging issues among heterogeneous computer architectures and displays. The **unitsize** field specifies how many contiguous pixel values are bundled into a single unit by the digitizer. The **sigbit** field specifies the order in which bits of significance are stored within each unit; most significant bit first or least significant bit first. The last of these three fields is the **byte_order** field. If **unitsize** is a multiple of bytes, then this field specifies the order in which bytes occur within the unit. Given these three attributes, binary incompatibilities across computer hardware and binary format assumptions within application software can be identified and effectively dealt with.

The **pix_offset** attribute defines a pixel displacement from the left edge of the raster image data to where a particular image's significant image information begins. The **whitepix** attribute defines the value assigned to the color white. For example, the binary image described in Figure 8 is black text on a white background and the value of the white pixels is 0. This field is particularly useful to image display routines. The **issigned** field is required to specify whether the units of an image are signed or unsigned. This attribute determines whether an image with a pixel depth of 8, should have pixels values interpreted in the range of -128 to +127, or 0 to 255. The orientation of the raster scan may also vary among different digitizers. The attribute field, **rm_cm**, specifies whether the digitizer captured the image in row-major order or column-major order. Whether the scan lines of an image were accumulated from top to bottom, or bottom to top, is specified by the field, **tb_bt**, and whether left to right, or right to left, is specified by the field, **rl_lr**.

The final attributes in IHead provide a single historical link from the current image to its parent image; the one from which the current image was derived or extracted. In Figure 8, the **parent** field contains the full path name to the image from which the image displayed in Figure 9 was extracted. The **par_x** and **par_y** fields contain the origin, upper left hand corner pixel coordinate, from where the extraction took place from the parent image. These fields provide a historical thread through successive generations of images and subimages.

We believe that the IHead image format contains the minimal amount of ancillary information required to successfully manage binary and gray scale images. The IHead format is extremely diverse in its ability to represent a wide variety of images. However, *hsfsys* requires a predetermined set of attributes to be used in the IHead structure. All HSF form images must be 2560 by 3300 pixels in dimension. The images must be binary, one bit per pixel with 0 representing white and 1 representing black. The images can be either uncompressed or compressed using CCITT Group 4. The binary raster data is assumed to be in a high-low byte order with the most significant bit first in a byte of pixel data. The pix_offset attribute is not used, so all pixel data in the image is processed by *hsfsys*. Finally, the data units are assumed to be unsigned and the scan order is left-to-right and top-to-bottom.

The file format is illustrated in Figure 10. Each IHead image file is divided into an IHead header followed by the image's raster data. Preceding the header is an 8-byte record containing the length of the IHead header. Both the value of the length record and the header values themselves are represented in ASCII. The raster data following the header is in the binary format described by the attribute values in the header and may be compressed. In this way, the header portion of the IHead image always remains uncompressed and can be interpreted by heterogeneous computer architectures. Applications that intend to manipulate the raster data of an IHead image are able to first read the ASCII header containing the image's attributes and determine the proper interpretation of the data that follows it.



Header Length (8 bytes)

ASCII Format Image Header
(288 bytes)

Binary Raster Stream

00000001000001000001111110 . . .

- Representing the digital scan across the page left to right, top to bottom.
- ‘0’ - Represents a white pixel.
- ‘1’ - Represents a black pixel.
- 8 Pixels are packed into a single byte of memory.

Figure 10. Illustrated IHead file format for an uncompressed image.

3.1.1.2 READ FIELD TEMPLATE; src/lib/hsf/hsftmplt.c; read_hsftmplt()

The spatial field template defines the location of each entry field on a registered HSF form. A registered form is a form that has no distortion and the location of the pixels comprising the form is known. The field template file is an ASCII file in which the first line contains the number of fields represented in the file. Each subsequent line in the template file represents an independent field on the form. In all, there are 34 entry fields on an HSF form. All but the first 3 fields are processed by *hsfsys*. All ASCII files used by *hsfsys* were generated and designed to run on UNIX computers so the end of each line is represented by the single line feed character with decimal representation 10 and hexadecimal representation 0A (0x0A in C). This is different from ASCII files on DOS computers that represent the end of each line with two characters, a line feed character 0x0A followed by a carriage return character 0x0D. Each field is represented as a rectangular region in the template file by a line comprised of 8 numbers. These 8 numbers represent four (x, y) vertex pairs. The first pair represents the upper-left corner, the second pair represents the upper-right corner, the third pair represents the lower-left corner, and the fourth pair represents the lower-right corner. Each number on a line is separated by a space character 0x20 or a tab character 0x09.

The field template provided with this distribution is in *tmplt/hsftmplt.pts*. The field regions stored within this file were measured from the blank registered form image *tmplt/hsftmplt.pct*.

3.1.2 PROCESS HSF FORM; src/lib/hsf/form.c; process_form2()

To process an HSF form, the image is first registered to remove any distortion introduced from reproduction or scanning, then the pixels comprising the form’s boxes and instructions are removed, leaving only the handprinted data entered inside of each field in the image.

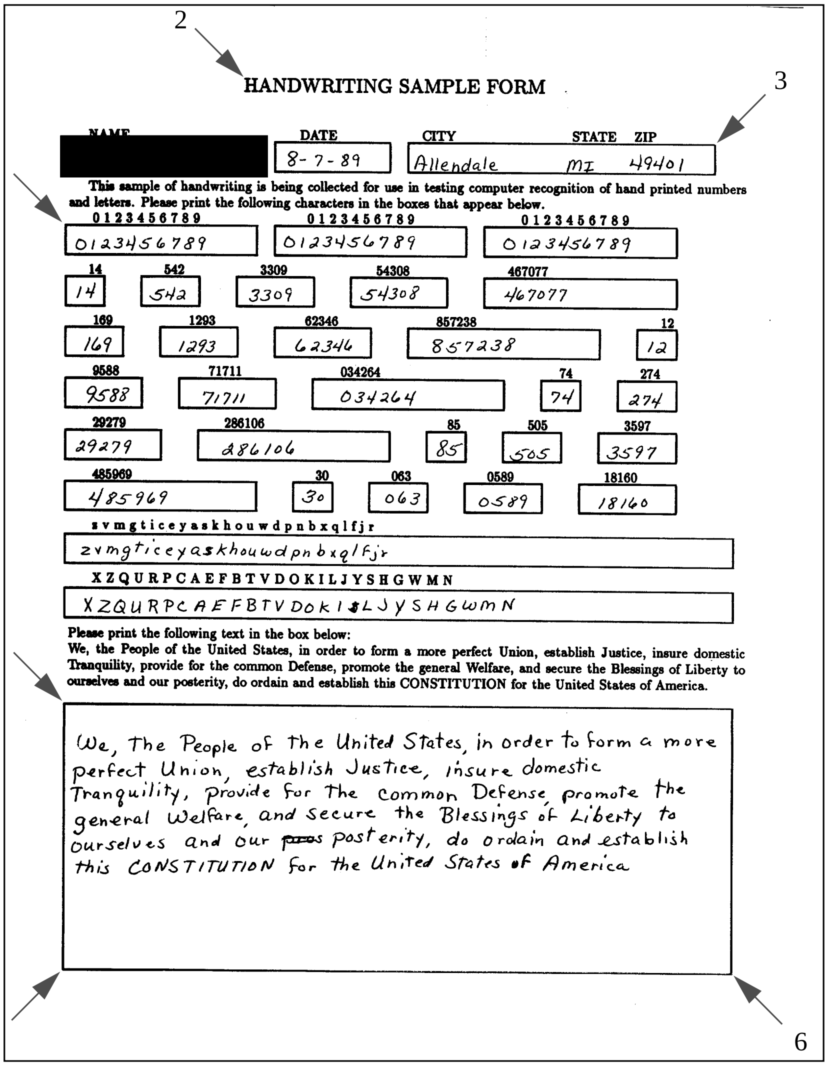
3.1.2.1 REGISTER FORM IMAGE; src/lib/hsf/reghsf.c; register_hsf2()

The HSF forms distributed with SD1 and SD3 were type-set on a computer and originally produced on paper using a laser printer. Multiple copies of each form were then reproduced on a large photocopier. The copies were bifolded into legal-size envelopes, mailed out, filled in by Census representatives, mailed back to NIST in business return envelopes, and finally digitized through an automated document feeder on a scanner. This process produces several sources of distortion in the final image. The distortion includes rotation, translation, scale, and fold distortions that must be accounted for in order for the recognition system to reliably locate the data entered in each field on a form. These types of distortions are detected and removed through a process known as form registration.

Hsfsys uses a registration technique based on Linear Least Squares[14] where a set of predefined registration marks in an input HSF form image are matched to marks on an ideal undistorted template. Global estimates of rotation, translation, and scale are automatically computed and applied so that the input image is transformed to line up as well as possible in a least squares sense with the ideal template.

3.1.2.1.1 Read Reference Points; src/lib/mdg/readmfs.c;p readmfsint2()

A set of registration marks is needed in order to estimate the amount of distortion in an input image. These registration marks correspond to structures easily detectable within an image of a form. These may be actual fiducial marks or they may be structures embedded within the form itself. Six points were measured from the blank registered HSF form *tmplt/hsftmplt.pct*. These points are stored in the file *tmplt/hsfreg.pts* and correspond in order to the top-left corner of the leftmost 0 through 9 digit box, the top-left point on the H in the form's title "HANDWRITING SAMPLE FORM", the top-right corner of the CITY-STATE-ZIP box, the top-left corner of the Constitution box, the bottom-left corner of the Constitution box, and the bottom-right corner of the Constitution box. These registration points are annotated on the HSF form shown in Figure 11 (scaled 0.5X).



HANDWRITING SAMPLE FORM

NAME	DATE	CITY	STATE	ZIP
[redacted]	8-7-89	Allendale	MI	49401

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below.

0 1 2 3 4 5 6 7 8 9	0123456789
0 1 2 3 4 5 6 7 8 9	0123456789
0 1 2 3 4 5 6 7 8 9	0123456789
14	14
542	542
3309	3309
54308	54308
467077	467077
169	169
1293	1293
62346	62346
857238	857238
12	12
9588	9588
71711	71711
034264	034264
74	74
274	274
29279	29279
286106	286106
85	85
505	505
3597	3597
485969	485969
30	30
063	063
0589	0589
18160	18160
s v m g t i c e y a s k h o u w d p n b x q l f j r	zvmgticeyaskhouwdpnbxqlfjr
X Z Q U R P C A E F B T V D O K I L J Y S H G W M N	XZQURPCAEFBTVDOKILJYSHGWMN

Please print the following text in the box below:
We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

We, The People of the United States, in order to form a more
perfect Union, establish Justice, insure domestic
Tranquility, provide for The common Defense, promote the
general Welfare, and secure the Blessings of Liberty to
ourselves and our ~~pros~~ posterity, do ordain and establish
this CONSTITUTION for the United States of America

Figure 11. Registration marks on an HSF form.

The six registration marks on the HSF form were selected so that they are distributed across the entire form with a slight concentration of points in the top-left portion of the form. This concentration is due to *hsfsys* exhibiting sensitivities to the top-left of the form when conducting form removal. Some of this sensitivity is known to be caused by local distortions in the image where the form was folded when it was sent through the mail.

The file *tmplt/hsfreg.pts* is in a general NIST file format known as a Multiple Feature Set (MFS) file. MFS files are editable ASCII files designed to contain lists of single or multi-column data where the data values residing on the same line are strongly associated with each other. The first line in the file contains the number of subsequent lines in the file. In the case of the reference points file, there are 6 subsequent lines in *tmplt/hsfreg.pts*, each containing an (x, y) coordinate pair of numbers. A space or tab character is used to separate values within the same line, and lines are terminated with the line feed character 0x0A. The library *src/lib/mfs* contains a suite of routines designed to read and write MFS files and manipulate MFS structures.

Figure 12 lists the C definition of the MFS structure that is stored in *include/mfs.h*. The structure contains three members. **Values** references an array of character strings, **alloc** holds the number of allocated positions within **values**, and **num** holds the number of contiguous positions currently holding information in **values**. Each line after the first in an MFS file is read into a single string, which in turn is stored in the next available position within **values**. Multiple items on a single MFS file line are appended together in a single string. It is the responsibility of an application to parse the independent items from the strings stored in the **values** array. In the case of *tmplt/hsfreg.pts*, the file is read into an MFS structure and then the x and y coordinates are parsed into two separate integer arrays. The MFS file convention provides a common I/O interface when manipulating editable lists of ASCII values. The items listed in an MFS file can be integers, floating point numbers, names, and/or any sequence of printable ASCII characters.

```
typedef struct mfsstruct{
      int alloc;
      int num;
      char **values;
} MFS;
```

Figure 12. C definition for the MFS structure.

3.1.2.1.2 Locate Hypothesized Points; src/lib/hsf/hsfpoint.c; hsfpoints()

The amount of discrepancy between the registration marks on an ideal undistorted form and the position of the corresponding marks in an input form image are used to estimate the amount of distortion in the input image. *Hsfsys* uses spatial histogram projections to locate the position of these registration marks within the input HSF form image. The spatial histograms represent black pixel densities aggregated across an image region either in a horizontal or vertical orientation.

Figure 13 contains an image region containing the second registration mark in *tmplt/hsfreg.pts*, the top-left point on the H in the form’s title “HANDWRITING SAMPLE FORM”. The top image in the figure shows the sequence of subimages on which spatial histograms are computed in order to locate the registration mark. Each subimage has been assigned a number that corresponds to one of the spatial histograms displayed below the top image.

Horizontal histogram 1 is first computed on the entire image region. There are two bands of black in the histogram. The top band represents the characters in the form’s title. Vertical histogram 2 is computed on a subimage that is centered about the top of the title determined from histogram 1. Histogram 2 is used to locate the left end of the title. Horizontal histogram 3 is computed on a subimage that begins at the left edge of the title determined by histogram 2 and extends approximately the width of a single character. Histogram 3 is used to determine where the top the H, the first character in the title, begins. Vertical histogram 4 is computed on a subimage that begins at the top of the letter H and extends downward approximately the height of a single character. This final histogram is used to determine the left edge of the letter H, at which point the registration mark is located.

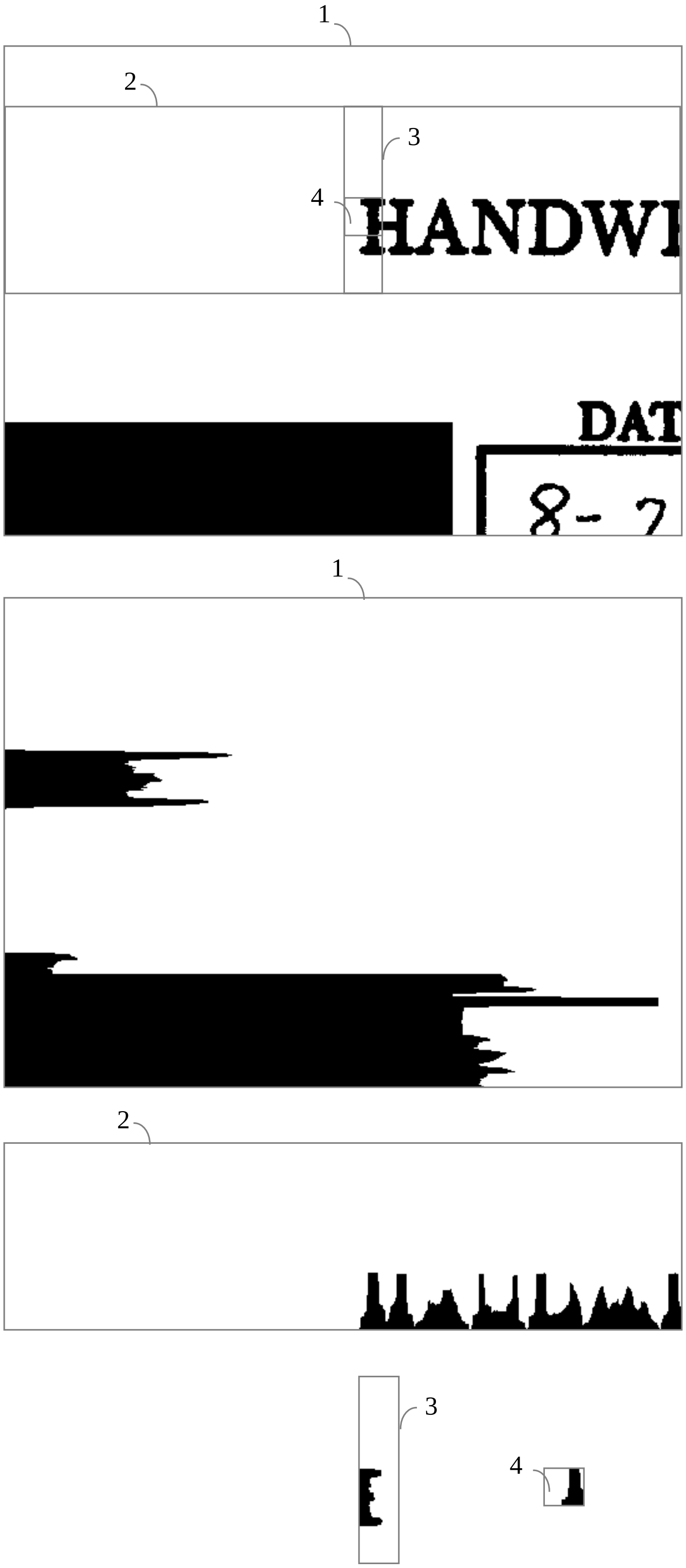


Figure 13. Locating a registration mark using spatial histograms.

As an image is increasingly rotated, the peaks in the histograms become shorter and they spread out wider making them decreasingly reliable and increasingly inaccurate. Therefore, the technique deployed in *hsfsys* carefully reduces the scope of successive histogram projections, alternating between horizontal and vertical projections, until the desired structure is accurately isolated. *Hsfsys* has been engineered and tested to tolerate up to 5 degrees of rotation in combination with 1.27 cm (0.5 inches) of translation.

3.1.2.1.3 Compute Distortion Parameters; src/lib/stats/lsq3.c; chknfindparam3()

Once the registration marks are located on a form, parameters estimating the amount of rotation, translation, and scale can be computed. The estimation of distortion parameters is embedded in a technique for detecting form registration failures. This section first describes how distortion parameters are used to detect form registration failures and then presents a method for deriving these distortion parameters using Linear Least Squares (LSQ).

Figure 14 contains pseudocode for an algorithm that detects form registration failures. The technique determines when registration points from within an input form image are incorrectly located. The recognition system can confuse or miss registration points for a number of different reasons. For example, a form may be so distorted that it cannot be corrected by the registration process. More frequently, an input form image has noise such as extraneous marks or writing in the vicinity of a registration mark, or worse yet, this noise may occlude the registration mark altogether. In the case where only one or two registration points are missed, if they can be detected, they can be removed from the LSQ computation. In most cases, using the remaining located registration points is sufficient for successful form registration.

```
input: located registration points - hyp_pts,
        ideal registration points - ref_pts
while (# hyp_pts > rm_limit)
    params = compute distortion parameters (hyp_pts, ref_pts)
    for each pt in hyp_pts
        trans_pt = apply distortion transformation (pt, params)
        errors[i] = distance (trans_pt, ref_pts[i])
    end for
    max_pt = find maximum error point (errors)
    max_err = find maximum error (errors)
    if (max_err > err_limit) then
        remove max_pt from hyp_pts
    else
        break from while
    endif
end while
if (# hyp_pts < rm_limit) then
    output: “form registration failed”
else
    output: “form registration successful”, params
endif
```

Figure 14. Pseudocode for *chknfindparam3()*, which detects form registration failures.

Walking through the algorithm in Figure 14, the procedure accepts as input the set of located registration points (hypothesis points) from an input form image. The procedure accepts a second set of corresponding points (reference points) extracted from the position of the registration marks on an ideal undistorted form. While the number of hypothesis points remaining in the analysis is more than a given threshold, in this case 3 points, the analysis continues. For each pass through the while loop, distortion parameters are computed from the remaining hypothesis points and their corresponding reference points. Then, using the new distortion parameters, each hypothesis point is transformed. If the parameters are a good estimate of the actual distortion, the transformed points will be very close to their corresponding reference points. An error distance is computed between each hypothesis and reference point pair. If the maximum error distance from all the points exceeds a given threshold, the hypothesis point contributing to the maximum error is removed from the analysis, and new distortion parameters are computed from the remaining hypothesis points. This process continues until either there are too few hypothesis points remaining to accurately compute distortion parameters, or the maximum error distances from all the remaining points falls below a specified threshold. If too few points remain, the form registration is determined to have failed. Otherwise, form registration is determined to be successful, and the last set of distortion parameters computed are used to transform the entire input form image. *Hsfsys* uses an error threshold of 4, which was derived empirically from a set of independent studies.

The procedure *chknfindparam3()* is an encapsulation of a lower level procedure *findparam3()* also located in *src/lib/stats/lsq3.c*. This lower level procedure is responsible for computing distortion parameters given the recognition system's located hypothesis points and their corresponding reference points. These distortion parameters are estimated using a method of LSQ and account for rotation, translation, and scale. A pair of linear equations using 3 unknowns can be defined to account for these distortions.

$$x_h = \Delta x + m_{x_x} x_r + m_{x_y} y_r \tag{1}$$

$$y_h = \Delta y + m_{y_y} y_r + m_{y_x} x_r \tag{2}$$

Equation (1) is used to estimate the translation, rotation, and scale in x using the three unknown quantities Δx, m_{x_x}, and m_{x_y}. Equation (2) is used to estimate the translation, rotation, and scale in y using the three unknown quantities Δy, m_{y_y}, and m_{y_x}. In the first equation, the hypothesized x-coordinate, x_h, is linearly dependent on the reference x-coordinate, x_r, and the reference y-coordinate y_r. The same is true for the hypothesized y-coordinates in the second equation. Here, reference points refer to the registration marks stored in the file *tmplt/hsfreg.pts* corresponding to the blank registered form *tmplt/hsftmplt.pct*. The reference points are where the marks should be located if the input image has absolutely no distortion whatsoever. Hypothesized points refer to the registration marks located within the input HSF form image using spatial histograms.

Applying the method of LSQ on Equation (1), the equation expands into the following system of three linear equations.

$$\sum_{i=1}^{n} x_h = n\Delta x + m_{x_x} \sum_{i=1}^{n} x_r + m_{x_y} \sum_{i=1}^{n} y_r \tag{3}$$

$$\sum_{i=1}^{n} x_h x_r = \Delta x \sum_{i=1}^{n} x_r + m_{x_x} \sum_{i=1}^{n} x_r^2 + m_{x_y} \sum_{i=1}^{n} x_r y_r \tag{4}$$

$$\sum_{i=1}^{n} x_h y_r = \Delta x \sum_{i=1}^{n} y_r + m_{x_x} \sum_{i=1}^{n} x_r y_r + m_{x_y} \sum_{i=1}^{n} y_r^2 \tag{5}$$

This system of three simultaneous linear equations is represented in matrix form as:

$$\mathbf{B} = \mathbf{AP} \tag{6}$$

where:

$$\mathbf{B} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} x_{h_i} \\ \sum_{i=1}^{n} x_{h_i} x_{r_i} \\ \sum_{i=1}^{n} x_{h_i} y_{r_i} \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} n & \sum_{i=1}^{n} x_{r_i} & \sum_{i=1}^{n} y_{r_i} \\ \sum_{i=1}^{n} x_{r_i} & \sum_{i=1}^{n} x_{r_i}^2 & \sum_{i=1}^{n} x_{r_i} y_{r_i} \\ \sum_{i=1}^{n} y_{r_i} & \sum_{i=1}^{n} x_{r_i} y_{r_i} & \sum_{i=1}^{n} y_{r_i}^2 \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} p_{11} \\ p_{21} \\ p_{31} \end{bmatrix} = \begin{bmatrix} \Delta x \\ m_{x_x} \\ m_{x_y} \end{bmatrix}$$

Solving for **P**, the following equation is derived:

$$\mathbf{P} = \mathbf{A}^{-1}\mathbf{B} \tag{7}$$

The inverse of the matrix **A** is defined to be:

$$\mathbf{A}^{-1} = \frac{1}{det\mathbf{A}} Adj\mathbf{A} \tag{8}$$

The determinant of **A** is defined to be:

$$det\mathbf{A} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12}$$

Using cofactors, the adjunct of **A** is defined to be:

$$Adj\mathbf{A} = \begin{bmatrix} (a_{22}a_{33} - a_{23}a_{32}) & (a_{13}a_{32} - a_{12}a_{33}) & (a_{12}a_{23} - a_{13}a_{22}) \\ (a_{23}a_{31} - a_{21}a_{33}) & (a_{11}a_{33} - a_{13}a_{31}) & (a_{13}a_{21} - a_{11}a_{23}) \\ (a_{21}a_{32} - a_{22}a_{31}) & (a_{12}a_{31} - a_{11}a_{32}) & (a_{11}a_{22} - a_{12}a_{21}) \end{bmatrix}$$

Multiplying **A**$^{-1}$ by **B**, using Equation (8) to compute **A**$^{-1}$, yields:

$$\mathbf{P} = \mathbf{A}^{-1}\mathbf{B} = \begin{bmatrix} p_{11} \\ p_{21} \\ p_{31} \end{bmatrix} = \begin{bmatrix} \left(\dfrac{b_{11}(a_{22}a_{33} - a_{23}a_{32}) + b_{21}(a_{13}a_{32} - a_{12}a_{33}) + b_{31}(a_{12}a_{23} - a_{13}a_{22})}{a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12}}\right) \\ \left(\dfrac{b_{11}(a_{23}a_{31} - a_{21}a_{33}) + b_{21}(a_{11}a_{33} - a_{13}a_{31}) + b_{31}(a_{13}a_{21} - a_{11}a_{23})}{a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12}}\right) \\ \left(\dfrac{b_{11}(a_{21}a_{32} - a_{22}a_{31}) + b_{21}(a_{12}a_{31} - a_{11}a_{32}) + b_{31}(a_{11}a_{22} - a_{12}a_{21})}{a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12}}\right) \end{bmatrix}$$

The LSQ parameter estimates for Equation (1) are derived by substituting the elements of **A** and **B** into the equations for **P**. The parameter estimates for Equation (2) are derived by substituting the following matrix elements.

$$\mathbf{B} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_{h_i} \\ \sum_{i=1}^{n} y_{h_i} y_{r_i} \\ \sum_{i=1}^{n} y_{h_i} x_{r_i} \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} n & \sum_{i=1}^{n} y_{r_i} & \sum_{i=1}^{n} x_{r_i} \\ \sum_{i=1}^{n} y_{r_i} & \sum_{i=1}^{n} y_{r_i}^2 & \sum_{i=1}^{n} x_{r_i} y_{r_i} \\ \sum_{i=1}^{n} x_{r_i} & \sum_{i=1}^{n} x_{r_i} y_{r_i} & \sum_{i=1}^{n} x_{r_i}^2 \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} p_{11} \\ p_{21} \\ p_{31} \end{bmatrix} = \begin{bmatrix} \Delta y \\ m_{y_y} \\ m_{y_x} \end{bmatrix}$$

This LSQ method computes a linear mapping that minimizes the total discrepancy (error) between all the reference points on a registered form and their corresponding hypothesized points measured on an input form containing distortion. Assuming that all points are reliably detectable, the error at any one point is decreased as the number of points used in the Least Squares calculation increases, causing the registration quality to improve.

3.1.2.1.4 Transform Form Image; src/lib/image/fitimage.c; f_fit_param3_image2()

Using the method of Linear Least Squares, the parameter estimates Δx, m_{x_x}, m_{x_y}, Δy, m_{y_y}, and m_{y_x} are substituted back into Equations (1) and (2) and the pixels in the input HSF form are transformed by computing (x_h, y_h).

Each black pixel in the input image is mapped or *pushed* to a new position within an initially blank output image. This approach is efficient because it only computes a transformation for those pixels in the image that are black. A trade-off to this approach is that the resulting output image may contain small amounts of speckle noise within dense black pixel regions. The small white pixel voids are caused by round-off when converting real-valued transformation addresses to discrete pixel locations. An alternative to pushing is *pulling*. In this case, an inverse transformation is computed for every pixel position in the output image, and pixels are *pulled* from the input image to the output image. This approach ensures complete coverage across the output image, and the speckle noise is avoided. Unfortunately, a transformation is computed for every pixel in the image making this approach computationally more expensive. In light of this, *hsfsys* uses the efficient pushing approach. Upon completion, the input HSF form has been transformed to fit the blank registered form *tmplt/hsftmplt.pct* and its spatial field template *tmplt/hsftmplt.pts*.

3.1.2.2 REMOVE FORM; src/lib/hsf/rmform.c; remove_form()

One approach to isolating the handprint entered on a form is to first remove the pixels comprising the form itself. Then, all that remains in the image is handprinted data in the presence of some amount of noise. Upon registration, the pixels making up the form in the input image are known to correspond to the pixels in *tmplt/hsftmplt.pct*. Therefore, the image in *tmplt/hsftmplt.pct* can be used as a mask so that, when laid over the registered input image, each pixel corresponding to a black pixel in the mask is erased from the input image.

3.1.2.2.1 Read Form Mask Image; src/lib/image/readrast.c; ReadBinaryRaster()

The LSQ method for form registration minimizes error, but does not absolutely remove all error. Detection of a registration mark even within an undistorted input image may be somewhat inaccurate and there is always a certain amount of discrete round-off error when implementing pixel-based transformations. Therefore, there will always be a small amount of discrepancy between a registered input image and the ideal mask. To compensate for these small amounts of error, the blank registered form image *tmplt/hsftmplt.pct* has been dilated[15] four times and stored in *tmplt/hsftmplt.d4*. This broadens all form structures in the blank form image so that coverage is improved when overlaid with the registered input image. The file *tmplt/hsftmplt.d4* is a binary IHead image and is loaded into *hsfsys* using the same routine *ReadBinaryRaster()* as is used to load the input HSF form image.

3.1.2.2.2 Subtract Form Pixels; lib/image/binlogop.c; nandbinimage()

The form is erased from the registered input image by applying the dilated blank form as a mask. A logical NAND (NOT followed by an AND) is used. For each pixel in the input image, an output pixel value is computed as follows:

$$o = r \mathbin{\&} (\sim m) \qquad (9)$$

where o is the output pixel, r is the pixel from the registered input image, and m is the corresponding pixel from the mask. In this way, o is set to black only when r is black and m is white. Remember that a black pixel has value 1 and a white pixel has value 0.

Upon image subtraction, characters in the registered input image may be left with holes and discontinuities. This occurs when the characters written in a field overlap with information already printed on the form or when strokes of characters extend across the form’s lines or instructions. At the time of this software release, NIST has not yet developed a complete solution to reconstructing disjoint strokes and holes in characters. However, initial experiments have been conducted to study this issue, and further research is required.

3.2 DO DIGIT FIELDS; src/lib/hsf/field.c; do_digit_fields()

This section describes how fields containing handprinted digits are processed by the standard reference recognition system. First, information must be loaded into the system to support feature extraction and the recognition of handprinted digit images. The handprint within a particular field is then extracted, segmented, size-normalized, and slant-normalized. Features are extracted from each segmented character image, and the features are classified. The results from the classification are stored field by field and include both the hypothesized digit identifications and their associated confidence values. Figure 15 lists the steps used to process digit fields, and the figure cross-references the steps to the software distribution according to file and subroutine name.

3.2.1 INITIALIZE FOR FIELDS; src/lib/hsf/field.c; init_field()

Three files are necessary to process fields that contain handprinted digits. The first file contains a set of basis functions that are used to compute feature coefficients from each segmented digit image. *Hsfsys* uses the Karhunen Loeve (KL) transform to compute these features.[6] KL basis functions have been computed off-line and stored in the file *weights/td3_d.evt*. The second file needed to process digit fields, *weights/td3_d.pat*, contains prototype KL feature vectors and a search tree used by an optimized Probabilistic Neural Network Classifier (PNN).[7] The third file, *weights/td3_d.med*, contains class-based median vectors computed from the prototypes in the *pat* file. If the small memory mode (the *-m* option), is used to invoke *hsfsys*, a smaller set of prototypes and their associated files are loaded instead. These files begin with the root file name *td3_d_s* in the top-level directory *weights*. This section describes how basis functions, prototype vectors, and median vectors are computed and how they are stored in their respective files.

3.2.1.1 READ BASIS FUNCTIONS; src/lib/nn/basis_io.c; read_basis()

The KL transform of a segmented character image is obtained by projecting the image onto the orthonormal eigenvectors of the covariance matrix of a large number of prototype images. The prototype images are representative of the types of images desired to be recognized by the recognition system, in this case, images of segmented digits. The KL transform requires computing the covariance matrix, and then diagonalizing it to produce the eigenvectors.[16] The resulting eigenvectors can be used as basis functions for feature extraction. Computing the KL transform is very expensive, but it is done once off-line, and the eigenvectors are stored in a basis function file for use in the recognition system. Appendix A documents the program *mis2evt* that computes the covariance matrix and eigenvectors from a training set of segmented and labeled character images. The output from this program is an *evt* file.

All elements of the basis function file occupy 4 bytes and are read-writable using the unformatted binary C functions *fread* and *fwrite*. The supplied basis function files found in the top-level distribution directory *weights* are assigned the extension *evt*. These files were written using C source code running on a computer that uses the Motorola (high-low) byte order format. Users of other computer architectures should be aware that byte orders may need to be changed for correct reading on their specific equipment.

The basis functions for the KL transform are eigenvectors, so these terms are used interchangeably in this document. The first element in the file is the integer number, n, representing the number of eigenvectors stored in the file. The second element is the integer dimensionality, D, of the eigenvectors. The remainder of the file consists of $n+2$ vectors, each with D elements. The first vector of length D contains the mean image vector of all of the images used to build the covariance matrix. The second vector exists for historical purposes only and has all elements equal to 1.0. The final n vectors are the eigenvectors of the covariance matrix, and they are stored in the order of decreasing eigenvalue.

The following items should be noted. The order of the elements within the eigenvectors corresponds to row major ordering of the image pixels. The ordering of the eigenvectors according to decreasing eigenvalue improves the efficiency of the PNN classifier. Finally, the images used in building the covariance matrix are 32 X 32 pixels in size, resulting in a dimensionality of $D = 1024$, which is fixed throughout the implementation of *hsfsys*.

3.2.1.2 READ PROTOTYPES & TREE; src/lib/nn/pat_io.c; readpatstreefile()

The features produced by projecting segmented character images onto the KL eigenvectors have been studied extensively by NIST.[17,18] A set of KL coefficients are computed by applying a set of eigenvectors to the same image. The image is then represented by the vector of coefficients rather than by its pixel data. The feature vectors are computed from a large training set of segmented character images and can be used to train classifiers such as PNNs. A large number of prototypes, tens of thousands, are required to train these classifiers, so they are computed off-line and stored in a prototype file.

3.2 DO DIGIT FIELDS	**3.2 src/lib/hsf/field.c; do_digit_fields()**
3.2.1 INITIALIZE FOR FIELDS	**3.2.1 src/lib/hsf/field.c; init_field()**
3.2.1.1 READ BASIS FUNCTIONS	3.2.1.1 src/lib/nn/basis_io.c; read_basis()
3.2.1.2 READ PROTOTYPES & TREE	3.2.1.2 src/lib/nn/pat_io.c; readpatstreefile()
3.2.1.3 READ MEDIAN VECTORS	3.2.1.3 src/lib/nn/median.c; readmedianfile()
For Each Digit Field	*For Each Digit Field*
3.2.2 PROCESS DIGIT FIELD	**3.2.2 src/lib/hsf/field.c; process_digit_field()**
3.2.2.1 ISOLATE 1-LINE FIELD	3.2.2.1 src/lib/hsf/isolate.c; iso_1line_field()
3.2.2.2 SEGMENT DIGIT FIELD	3.2.2.2 src/lib/hsf/segblob.c; segbinblobdigits()
3.2.2.2.1 Extract Blobs	3.2.2.2.1 src/lib/hsf/segblob.c; segbinblob()
3.2.2.2.2 Paste Digit Blobs	3.2.2.2.2 src/lib/hsf/segblob.c; paste_digit_blobs()
3.2.2.3 NORMALIZE CHARACTER IMAGES	3.2.2.3 src/lib/hsf/normaliz.c; norm_2nd_gen()
3.2.2.4 SHEAR CHARACTER IMAGES	3.2.2.4 src/lib/hsf/shear.c; shear_mis()
3.2.2.5 EXTRACT FEATURES	3.2.2.5 src/lib/nn/kl_mis.c; kl_transform_mis()
3.2.2.6 CLASSIFY FEATURE VECTORS	3.2.2.6 src/lib/nn/pnn.c; treepnnhypsconsC()
3.2.3 STORE FIELD RESULTS	**3.2.3 src/lib/fet/updatfet.c; updatefet()**
End Loop	*End Loop*
3.2.4 DEALLOCATE FOR FIELDS	**3.2.4 src/lib/hsf/field.c; free_field()**

Figure 15. Steps to process digit fields.

Like the basis function files, prototype files are read-writable using the unformatted C functions *fread* and *fwrite*. The supplied prototype files found in the top-level distribution directory *weights* are assigned the extension *pat*. These files were written using C source code running on a computer that uses the Motorola byte order format. Just as with basis function files, users of other computer architectures should be aware that byte orders may need to be changed for correct reading on their specific equipment. The prototype file format was originally implemented in FORTRAN, which embeds integer stream lengths at the beginning and end of a byte block of data. Therefore, the low-level reading routines in C are complicated by this feature.

The first element in a prototype file is a 4-byte integer always having a value of 24. It indicates that six 4-byte integer follow. These six integers constitute the file's header, and currently only four of the integers are actually used. The first of the six integer elements, *P*, represents the number of feature vectors stored in the file; the second element *n* signifies the dimensionality of the feature vectors; the third element *L* is the number of possible classes to which the vectors may belong; the fourth element is not used; the fifth element indicates the format used in the file and must be a value of 5151; and the sixth element is currently unused. After the six integers, the initial block size integer with value 24 is repeated.

The section following the header in a prototype file contains the class-set that identifies each class with a user-defined string. The data block starts with a 4-byte integer assigned the value $32 \times L$. The class set strings follow with *L* strings each of length 32 bytes (they do not need to be null terminated). The same integer data length of $32 \times L$ concludes the class-set section.

The largest section of the file follows the class set and contains KL feature vectors. The *n* elements of one vector are followed by the next for a total of *P* vectors. These floating point feature vectors are most conveniently input using a single *fread* of $4 \times P \times n$ bytes into a preallocated block of contiguous memory. The feature vectors are followed by a vector of integer indices on the range [0,*L*-1]. These indices identify the class of each feature vector stored in the file and can be read as a single block of $4 \times P$ bytes.

The final section in a prototype file contains a search tree that is used to minimize the computational intensity of a traditional PNN classifier. This tree contains indices pointing to the feature vectors stored in the file. Therefore, the ordering of these features is important and must remain fixed. For this reason, the tree is included in the prototype file. The tree section begins with two 4-byte integers. The first integer is the number of nodes, *N* in the tree. The second integer is the number of children per node, *C*. A matrix containing $N \times C$ 4-byte integers follows. The matrix is followed by five vectors each containing *N* 4-byte elements. The first three vectors contain integers, while the last two hold floating point values. Section 3.2.2.6 will discuss the use of this tree in more detail, and Appendix B documents the program *mis2pat* that generates prototype files.

3.2.1.3 READ MEDIAN VECTORS; src/lib/nn/median_io.c; readmedianfile()

The program *mis2pat* produces a second file that is needed for character classification. Median vector files are supplied in the top-level distribution directory *weights* and are assigned the extension *med*. A median file contains as many vectors as there are classes in an application. For example, there are 10 classes when recognizing digits as compared to 26 classes when recognizing upper case letters. Each median vector has the same dimensionality *D* as the feature vectors stored in a corresponding prototype file. The k^{th} element of the i^{th} vector contains the center value from the sorted list of the k^{th} elements from all the training feature vectors for class *i*. The median vectors are used during classification to initialize the search through the tree stored in a corresponding prototype file.

The median vectors are stored in an ASCII file format. The first line of the file contains two space-separated integers. The first integer specifies the number of vectors in the file, the second integer specifies the number of elements in each vector. The floating point vectors follow, one after another, with a blank line separating each vector. The order of these vectors correspond to the class indices stored in a corresponding prototype file.

3.2.2 PROCESS DIGIT FIELD; src/lib/hsf/field.c; process_digit_field()

With the input HSF form image registered and the form information removed, the handprint entered within each field can be extracted using the spatial field template. The field subimage is then segmented, separating each handprinted digit into its own image. Handprint varies widely in size and slant between different writers, so each segmented digit image is normalized so that the character is scaled to a consistent size, and the character is straightened to remove slant. Features are extracted from each character image so that an image is represented by a vector of floating point coefficients rather than by its binary pixels. These feature vectors are classified by a neural network, and the hypothesized digit classifications along with their associated confidence values are stored.

3.2.2.1 ISOLATE 1-LINE FIELD; src/lib/hsf/isolate.c; iso_1line_field()

Through the process of form registration, the input HSF form has been transformed to line up with the spatial field template stored in *tmplt/hsftmplt.pts*. This spatial template defines the location and spatial extent of each entry field on the HSF form. Each field region is represented by 4 pixel coordinate points representing the corners of a rectangle that is aligned with the raster grid in the input image. These rectangular template coordinates are used to extract subimages of the fields from the registered input image. Figure 16 contains an example of an extracted field (scaled up 4X). Notice that in addition to the handprint there is still a small amount of form information that was not erased during form removal. Spatial histograms similar to those used in locating registration marks are used to separate the handprinted data from the form data.

The techniques used works off the assumption that the entry field contains a single line of handprinted text, and the handprint can be distinguished from the edges of the form by examining pixel densities within a spatial histogram projection. A horizontal histogram computed on the example field image is displayed in Figure 17.

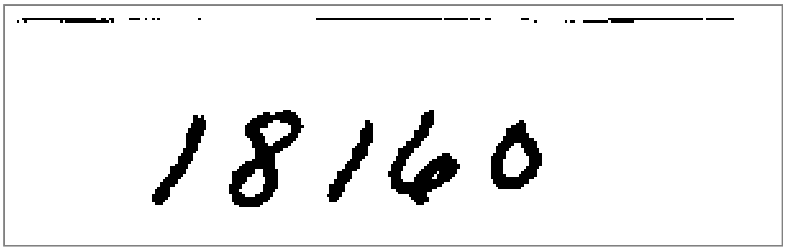


Figure 16. Form residue in an isolated field.

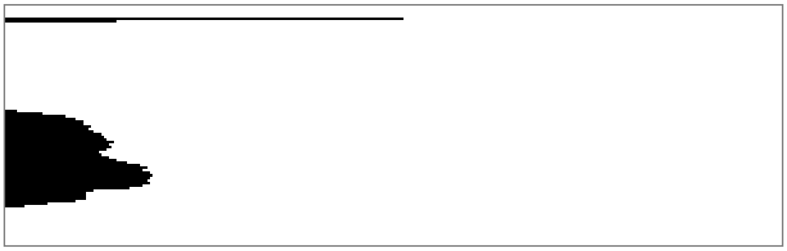


Figure 17. Horizontal histogram of image displayed in Figure 16.

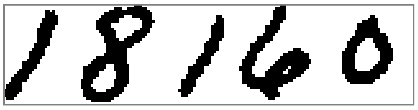


Figure 18. Cropped field image containing only handprint.

Index	Bins	Thresh	Runs	
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0	0	
5	156	1	1	
6	44	1	2	
7	0	0	0	
8	0	0	0	
9	0	0	0	
• • •	• • •	• • •	• • •	
38	0	0	0	
39	0	0	0	
40	0	0	0	← **top edge**
41	5	1	1	
42	15	1	2	
43	24	1	3	
44	28	1	4	
45	31	1	5	
46	31	1	6	
47	34	1	7	
48	33	1	8	
49	35	1	9	
50	38	1	10	
51	39	1	11	
52	40	1	12	
53	43	1	13	
54	41	1	14	
55	42	1	15	
56	40	1	16	
57	37	1	17	
58	38	1	18	
59	41	1	19	← **middle**
60	44	1	20	
61	48	1	21	
62	53	1	22	
63	56	1	23	
64	54	1	24	
65	57	1	25	
66	58	1	26	
67	57	1	27	
68	56	1	28	
69	57	1	29	
70	54	1	30	
71	49	1	31	
72	35	1	32	
73	32	1	33	
74	32	1	34	
75	32	1	35	
76	28	1	36	
77	17	1	37	
78	8	1	38	
79	0	0	0	← **bottom edge**
80	0	0	0	
81	0	0	0	
• • •	• • •	• • •	• • •	

Figure 19. Technique for locating the vertical center of a line of handprinted text.

In order to locate the handprinted text, the histogram values plotted in Figure 17 are converted to a vector of incremental run length values. This process is illustrated in Figure 19. The first column contains a list or row indices for the example image. The second column of numbers lists the histogram values whose positions within the list are called *bins*. Notice that the lists have been shortened at points of contiguous zeros so that they fit within the figure. All those bins having an accumulated value of more than 2 pixels is set to 1, and those bins having less than 2 pixels is set to 0. The resulting binary vector is listed in the third column of the figure. A run length counter is initialized for each contiguous group of binary values equal to 1, and each subsequent value of 1 in a run is replaced by the current run length counter, and the counter is then incremented. The results of this step are shown in the fourth column. Finally, the run with the longest length is selected, and the midpoint of the run is determined to represent the vertical middle of the handprinted text within the field. In the example shown in Figure 19, the second run with length 38 is selected, and raster row 59 is determined to approximate the middle of the handprinted text.

Given the approximate middle of the handprinted text line, the histogram bins in the second column of Figure 19 are searched to locate the edges of the text. One search starts at the approximate middle and proceeds upwards until a bin equal to 0 is encountered, and in a similar way a second search starts at the approximate middle and proceeds downwards until a bin equal to 0 is encountered. The two points at which the bins become zero are assumed to correspond to the top and bottom edges of the handprinted text line. Image rows between these two points are extracted, and any form residue is cropped.

The one-line fields on the HSF from are much wider than they are tall, so form residue is much more common along the horizontal sides of these fields than along the vertical sides. The longer horizontal sides permit small amounts of registration error to propagate until it becomes significant, whereas the error along the shorter vertical sides is seldom propagated to the extent that it is noticeable. Therefore, the process of thresholding the histogram bins and computing run length increments is done only for locating the top and bottom edges of the handprint within the field. Left and right edges are found by searching vertical histogram bins directly. Searching in from a left or right edge, the first histogram bin greater than 10 pixels is found, and then a reverse search from that point locates the first bin that equals zero. The use of a 10 pixel threshold avoids speckle noise in the field and locates the beginning of significant character data, while the reverse search locates the edge of the character data. *Hsfsys* extracts the subimage bounded by these left, right, top, and bottom edges, and the result of cropping the image in Figure 16 is shown in Figure 18.

3.2.2.2 SEGMENT DIGIT FIELD; src/lib/hsf/segblob.c; segbinblobdigits()

At this point, *hsfsys* has a subimage containing the handprint of one or more digits. The feature extraction and classification techniques used by the recognition system are designed to classify images containing a single character. Therefore, the field image of multiple characters must be segmented into plausible character images, one character per image. To do this, the system uses connected components or *blobs* to define these plausible character images. A blob is defined to be a group of pixels all contiguously neighboring or *connecting* each other. In general, each blob is extracted and assumed to be a separate character, although a blob is not guaranteed to be a single and complete character. This is frequently the case with handprinted fives. Often a writer will print the top horizontal stroke of a 5 so that it does not connect the bottom portion of the digit. In this case, the two pieces of the same five will be treated incorrectly as two independent characters. To avoid this type of problem, a blob pasting step has been developed.

Based on experience gained from creating and manipulating large on-line image databases such as SD3, NIST has developed a number of diversified structures and file formats. Storing character images in individual files has proven to be very inefficient, especially when manipulating databases containing hundreds of thousands of characters. Devoting a separate file node for each character image creates enormous file system overhead, and unreasonably large directory tables must be allocated. Rarely do recognition system components process only a single character image in isolation. Rather, most components are designed to process a large sample of characters. Experience has shown that the gathering of a large sample of characters from a file system where the images have been stored in individual files greatly burdens the computer's disk controller. This results in slow experiment loading times as well as limiting the access of other applications to data stored on the same storage device.

In addition to creating large directory tables, storing segmented character images in individual files results in sparse usage of the storage device. This sparseness is even more exaggerated when the images are compressed. For example, segmented character images in SD3 have been centered within a 128 by 128 binary pixel image. The resulting image size is 2,344 bytes, 296 bytes for the IHead header and 2,048 bytes of image data. These files when CCITT Group 4 compressed average 360 bytes in size, 296 bytes for the IHead header and only 64 bytes of compressed image data. Storing these compressed image files onto CD-ROM for example, which uses a 2,048 byte block size, would be extremely wasteful. Only 18% of each block containing image data would be used.

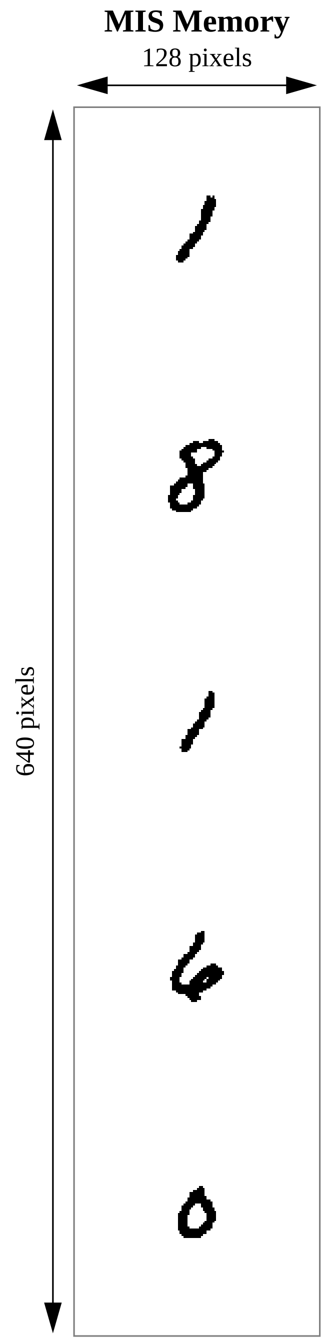
In light of these observations the segmentor used by *hsfsys* creates a memory structure called a Multiple Image Set (MIS). The MIS structure and file format have been developed to manage large volumes of segmented character images. The MIS format allows multiple images of homogeneous dimensions and depth to be stored in one file. MIS is a simple extension or encapsulation of the IHead format described in Section 3.1.1.1. It can be seen in Figure 20 that the IHead structure is included as a member within the MIS definition. The library *src/lib/mis* contains a suite of routines designed to read and write MIS files and manipulate MIS structures.

An MIS file contains one or more individual images stacked vertically within the same contiguous raster memory, the last pixel row or *scanline* of the previous image is concatenated to first scanline of the next image. The individual images that are concatenated together are referred to as MIS *entries*. The resulting contiguous raster memory is referred to as the MIS *memory*. The MIS memory containing one or more entries of uniform width, height, and depth is stored using the IHead file format. The IHead attribute fields are sufficient to describe the MIS memory. The IHead structure's **width** attribute specifies the width of the MIS memory, and likewise the IHead structure's **height** attribute specifies the height of the MIS memory. In this way, the MIS memory can be stored just like any normal IHead image, including possible compression. An example of an MIS memory is displayed in Figure 21 (scaled up 3X). In this example, each extracted character is centered within a 128 by 128 MIS entry.

Due to the uniform dimensions of MIS entries, the IHead structure's **width** attribute also specifies the width of the entries in the MIS memory. What is lacking from the original IHead definition is the uniform height of the MIS entries and the number of MIS entries in the MIS memory. Notice that, given the uniform height of the MIS entries, the number of entries in the MIS memory can be computed by dividing the entry height into the total MIS memory height. The interpretation of two of the IHead attribute fields, **par_x** and **par_y**, changes when the IHead header is being used to describe an MIS memory. The **par_x** field is used to hold the uniform width of the MIS entries, and the **par_y** field is used to hold the uniform height of the MIS entries. In other words, **width** and **height** represent MIS memory width and MIS memory height respectively, while **par_x** and **par_y** represent MIS entry width and MIS entry height respectively. Using this convention, an MIS file is treated like an IHead file.

```
/************************************************************
        Filename: Mis.h
        Author: Michael D. Garris
        Date: 7/18/90
************************************************************/
typedef struct misstruct{
    IHEAD *head;
    unsigned char *data;
    int misw;
    int mish;
    int entw;
    int enth;
    int ent_num;
    int ent_alloc;
} MIS;
```

Figure 20. C definition for the MIS structure.



Parent (x, y)	Blob (w, h)
0, 2	21, 35
30, 0	29, 38
68, 4	18, 32
92, 0	28, 37
132, 4	20, 27

Figure 21. An example of an MIS memory segmented from the field image in Figure 18.

Figure 20 lists the MIS structure definition written in C and found in *include/mis.h*. The structure contains an IHead structure, **head**, and an MIS memory, **data**. In addition, there are six other attribute fields that hide the details of the IHead interpretation from application programs that manipulate MIS memories. The MIS attributes **misw** and **mish** specify the width and height of the MIS memory. These values are the same as the **width** and **height** attributes contained in the IHead structure pointed to by **head**. The MIS attributes **entw** and **enth** specify the uniform width and height of the MIS entries. These values are the same as the **par_x** and **par_y** attributes contained in the IHead structure pointed to by **head**. The MIS attribute, **ent_alloc**, specifies how many MIS entries of dimension **entw** and **enth** have been allocated to the MIS memory **data**. The MIS attribute, **ent_num**, specifies how many entries out of the possible number allocated are currently and contiguously contained in the MIS memory **data**.

In addition to extracting character images, the segmentor used by *hsfsys* computes the location of where each character was extracted form the field image, and also computes the characters width and height. These statistics are listed with the MIS memory in Figure 21.

3.2.2.2.1 Extract Blobs; src/lib/hsf/segblob.c; segbinblob()

A serial implementation of a connected component algorithm called *findblob()* has been developed that is relatively inexpensive to compute and is included in this software distribution in the file *src/lib/image/findblob.c*. This utility finds a single blob from the input image and returns a subimage containing the blob. By calling the utility repeatedly, one can obtain all the blobs in the input image, or if desired, just some of the blobs. Each call to *findblob()* initiates a search that begins at a specified starting point in the input image and proceeds to scan the input image in column-major (top-to-bottom and left-to-right) order for a black pixel. Once found, the black pixel is grown into a complete blob region. The caller may leave the current point of the scan unchanged between calls, thereby making a complete scan that finds all blobs upon subsequent calls, or the caller may change the starting point so as to direct the search to specific regions within the input image.

Findblob() is extremely flexible and has been designed with a number of different options. These options control the clearing of blobs from the input image, the allocation of memory for the output image, and the format of the output image. Input image pixels that are members of a detected blob can either be left alone or erased. The caller can either provide the space needed for the output image or let the utility allocate the required amount of memory. Finally, there are three available output format options. In the first format, the blob is returned in an output image the same size and dimension as the input image. In this case, the blob occupies the same position in the full-size output image as it did in the input image. The second format returns the blob centered in an image whose dimensions are specified by the caller. The final format option causes the blob to be returned in an image allocated just large enough to contain it. In other words, the output image is defined to be the bounding box around the blob.

Starting at a specified pixel position, the utility scans the input image for a black pixel. When a black pixel is found, it is grown into a *run*. Here, a run is a maximal horizontal segment of contiguous black pixels. The run is then grown into a maximal set of connected runs, which constitutes an entire blob. During the growth process, bytes representing pixels of the blob are changed from ones to zeros, and structures representing the runs are stored in an array. The pixels must be changed to avoid finding them again. (If the caller opts not to have these pixels changed upon return, then they are set back just prior to exiting the utility.)

The array of runs is treated as a queue. One growth step consists of reading a run from the head of the queue, producing new runs if there are any black pixels connected to its top or bottom, and appending these new runs to the queue tail. The queue is initialized to just the seed run that is grown from the position of the first black pixel encountered during the column-major scan. The growth steps continue until the queue becomes empty. The tail of the queue does not wrap around so as to recycle array positions (as is typical with most queue implementations). Instead, head and tail pointers both move toward higher addresses, so that when the growth is finished, the array contains all elements that have ever been in the queue. The routine then systematically goes through all the run structures and sets their corresponding pixels to black in the output image. The output image, representing one blob, is then returned to the caller.

The routine is efficient because it localizes processing to only the black pixels in the image, and it does so one blob at a time. In addition, the algorithm’s implementation generally requires an amount of working memory that is small compared to the memory occupied by the input image. The blobs returned by this utility are treated as plausible character images. If a blob is too small it is thrown away and ignored. If a blob is too big it currently is also thrown away. A future refinement to this segmentor would be to try to break any large blob down into smaller pieces because it is likely to contain multiple characters connected to each other. In *hsfsys*, if a blob has less than 100 black pixels it is considered too small, and if a blob has more than 1750 black pixels it is considered too big.

3.2.2.2.2 Paste Digit Blobs; src/lib/hsf/segblob.c; paste_digit_blobs()

Unfortunately, using connected components for segmentation has some significant pitfalls. A blob is not guaranteed to be a single and complete character. If two characters touch, then a single blob will contain both characters as a single composite image. A blob may also contain only one stroke of a character that is comprised of several disjoint pieces. For example, the top of the letter T may not be connected to the vertical stroke, causing the algorithm to over-segment the character into two blobs.

Figure 22 shows a field containing “DAuGhter” in which connected component labeling over-segments and under-segments the field. The extracted field image is shown at the top, and the resulting blobs are listed below it. The first blob is a vertical stroke that when viewed independently looks like a 1, *l*, or I. This blob is the vertical stroke representing the left potion of the first letter D. This is an example of over-segmenting. The remaining three blobs are examples of under-segmenting. The second blob contains portions of D, A, and u. In this example, the single blob is assigned a class of X by the recognition system’s character classifier because the blob is assumed to be a single character. The third blob contains both the G and h and is assigned a class of G. The h is deleted from the field. The fourth blob contains t, e, and a portion of a clipped r. This blob is incorrectly assigned a class of W. Due to segmentations errors introduced by using connected components, the field is recognized as “HXGW” rather than “DAUGHTER”.

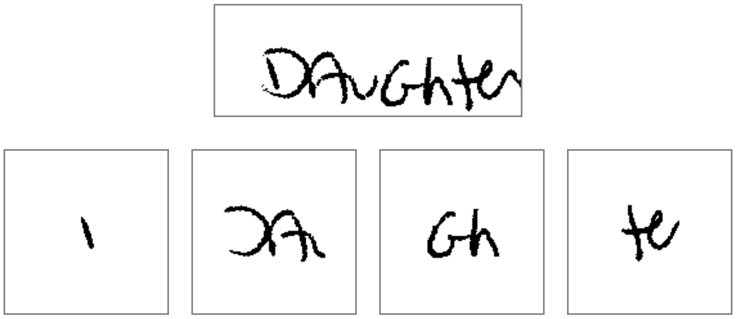


Figure 22. An example of over and under-segmenting using connected components.

The problem of over-segmentation does occur when using connected components to segment digits. For example, a writer will often print the top horizontal stroke of a 5 so that it does not connect to the bottom portion of the digit. The two pieces of the same 5 will be treated incorrectly as two independent characters. To avoid this type of problem, a method of blob pasting has been developed.

The connected component utility extracts blobs in a column-major order, so blobs are extracted left-to-right within a one-line text field. In addition to the blob images, the utility returns the location from where the blob was extracted in the field along with the blob’s width and height approximated by a bounding rectangle. Examples of these statistics are listed in Figure 21. At times it becomes necessary to join two blobs together as with the top and bottom pieces of a disjoint five. The decision to join two blobs is based on a simple heuristic that tests neighboring blobs. The heuristic tests the *current* blob with it neighbor, the *next* blob. If the difference between the next blob’s bottom minus the current blob’s top is less than half the current blob’s height, then the two blobs are pasted back together as a new plausible character image. This simple heuristic works very well at putting the tops back on disjoint fives.

3.2.2.3 NORMALIZE CHARACTER IMAGES; src/lib/hsf/normaliz.c; norm_2nd_gen()

To improve the classification performance of character images, a size-normalization technique referred to as *second generation normalization* was developed. Handprinted characters come in all different shapes, sizes, and slants. Some people will use all the space provided on a form, and others will use as little space as possible. It has been observed that the same characters printed by the same writer can vary greatly in size. As a writer begins to run out of room, he will do all types of curious things to fit the remainder of his answer in the field.

Second generation normalization attempts to remove size variations in handprint by scaling all segmented character images to a consistent size. The scaling is handled by an efficient serial *zoom()* utility provided with the software distribution in *src/lib/image/zoom.c*. The normalization method bounds the character data within a segmented image with a box, and that box is scaled to fit exactly within a 20 by 32 pixel region, and the aspect ratio of the original character is *not* preserved. The resulting 20 by 32 pixel character is then centered within a 32 by 32 pixel image.

Tests at NIST have shown that size-normalization improves recognition performance when recognizing handprinted characters. The technique also applies a simple morphological operator in an attempt to normalize the stroke width within the character image. If the pixel content of a character image is significantly high, then the image is eroded (strokes are thinned). If the pixel content of a character image is significantly low, then the image is dilated (strokes are widened). The left image in Figure 23 shows an original segmented character (scaled up 4X) centered within a 128 by 128 image. The same character spatially normalized using second generation normalization is displayed in the right image.

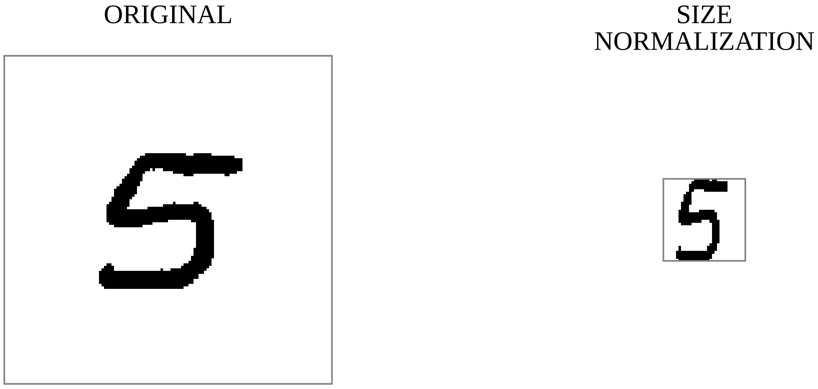


Figure 23. Results of size-normalizing a segmented character image.

3.2.2.4 SHEAR CHARACTER IMAGES; src/lib/hsf/shear.c; shear_mis()

As mentioned earlier, not only does the shape and size of handprinted characters vary, but their slant can also be significantly different. As size-normalization attempts to reduce character variations due to scale, slant-normalization attempts to reduce character variations due to slant. By effectively reducing these two sources (size and slant) of variation, a character classifier is left to deal primarily with variations due to character shape.

The slant is removed by a technique that uses horizontal shears in which rows in the image are shifted left or right in an attempt to straighten the character in the image. Given a segmented character image, the top and bottom image rows containing black pixels are located. The leftmost black pixel is located in each of the two rows, and a linear shifting function is calculated to shift the rows in the image so that when finished the leftmost pixels in the top and bottom rows line up in the same column. The rows between the top and bottom are shifted in lesser amounts based on the linear shifting function.

A slope factor *f*, defining the linear shifting function is calculated:

$$f = \frac{t_l - b_l}{b_r - t_r} \qquad (10)$$

where t_r is the vertical position of the top row, b_r is the vertical position of the bottom row, t_l is the horizontal position of the leftmost black pixel in the top row, and b_l is the horizontal position of the leftmost black pixel in the bottom row. The slope factor is used to compute a shift coefficient as follow:

$$s = (r - m) f \qquad (11)$$

with *r* being a vertical row index in the image and *m* equal to the vertical middle of the image. This causes the shifting to be centered about the middle of the image. A positive value of the shift coefficient causes a row to be shifted *s* pixel positions to the right, and a negative value causes a row to be shifted *s* pixel positions to the left.

This slant-normalization technique is applied after size-normalization. The results of shearing a size-normalized handprinted 4 in order to remove the character's slant is shown (scaled up 8X) in Figure 24. The technique is very inexpensive to compute and it works very well. This technique occasionally fails to remove the slant from the character when the top-leftmost black pixel in the image and the bottom-leftmost black pixel in the image do not lie on the same vertical stroke. This is more likely to happen with characters such as H and M where there are two equally likely peaks at the top and bottom of the character. In these cases, the slant may not be removed, although the results of the shearing do cut down on character variations, which is the underlying goal of this process.

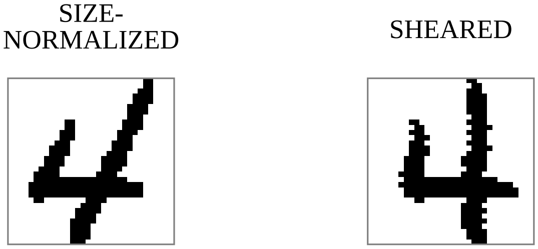


Figure 24. Slant removed from a character image via shearing.

3.2.2.5 EXTRACT FEATURES; src/lib/nn/kl_mis.c; kl_transform_mis()

The Karhunen Loeve (KL) transform has many optimal properties and is widely used in the pattern recognition field.[19] The KL transform is a linear transform and corresponds to the projection of images onto the eigenvectors of a covariance matrix, where the covariance matrix is created from images of training data such as those distributed with SD3 and in the top-level directory *train*. The production of this transform is also known as *principal factor* or *principal components* analysis. The creation of the covariance matrix and its eigenvectors is conducted off-line, and the computed eigenvectors are stored in a basis function file described in Section 3.2.1.1.

The pixels of a segmented character image define a vector whose elements are obtained by considering the 2-dimensional *N* by *N* image as a vector of N^2 elements. This vector is formed by concatenating the rows of the image together, and each binary element is converted according to Equation 13. Black pixels are represented as 1 and white pixels are represented as -1. The segmented character images have been size-normalized to be 32 by 32 pixels; therefore, $N = 32$ and $N^2 = 1024$.

$$\mathbf{u} = (u_{11}, u_{12}, \ldots, u_{1N}, u_{21}, \ldots, u_{2N}, \ldots, u_{NN}) \qquad (12)$$

$$u_{ij} = \left\{ \begin{array}{l} 1 \text{ if black pixel} \\ -1 \text{ if white pixel} \end{array} \right. \qquad (13)$$

The mean vector in Equation (14) is computed from all the training images.

$$\mu = \frac{1}{P} \sum_{i=1}^{P} \mathbf{u}^{(i)} \qquad (14)$$

This mean vector is subtracted from all the training images forming a set of *sample* vectors. Each sample vector comprises a column in the sample matrix, **U**. The covariance matrix R is symmetric and is formed as the outer product of the *P* sample vectors as in Equation (15).

$$\mathbf{R} = \frac{1}{P}\mathbf{U}\mathbf{U}^T \tag{15}$$

The covariance matrix is diagonalized using standard FORTRAN linear algebra routines such as those in EISPACK[20], producing the eigenvalues and corresponding eigenvectors in descending order of largest eigenvalue. The covariance matrix **R** has N^2 eigenvectors as the columns of Ψ defined in the equation

$$\mathbf{R}\Psi = \Psi\Lambda \tag{16}$$

and the only nonzero elements of Λ are the eigenvalues on its diagonal. The KL transform **v** of a vector **u** is the projection of the vector minus the mean vector μ onto the eigenvector basis Ψ.

$$\mathbf{v} = \Psi^T(\mathbf{u} - \mu) \tag{17}$$

Typically, only a subset of the eigenvectors corresponding to the largest eigenvalues are used in the transformation. The initial dimensionality of **u** is N^2. By selecting only the top k eigenvectors, the dimensionality of the transformed feature vector **v** is reduced to k. In *hsfsys*, k is selected to be 64. For a more complete discussion of the effect of feature dimensionality please refer to Reference 22.

Several steps have been taken to increase the efficiency of the KL transform when applied to binary images. The first improvement is a pre-multiplication step in which certain factors that are not dependent on the image data are computed once up front, then these factors are reused over a set of segmented character images. The second optimization takes advantage of the binary nature of the image data. The details of the implementation can be examined in the source code file *src/lib/nn/kl.c*.

3.2.2.6 CLASSIFY FEATURE VECTORS; src/lib/nn/pnn.c; treepnnhypsconsC()

It has been our experience that Probabilistic Neural Networks (PNNs), outperform Multi-Layer Perceptrons (MLPs) and other popular classifiers in terms of accuracy.[21,22] The PNN algorithm, in its traditional implementation[7], takes a large training set of prototype vectors and uses euclidean distances from an unknown vector to each of the training vectors. These distances are computed each time an unknown vector is classified. Similar methods are used in k-nearest neighbor classifiers. This computation is very expensive, so up till now, the slow processing times incurred by software implementations of PNN have outweighed the accuracy benefits of the classification.

Hsfsys uses a new optimized version of PNN that was developed at NIST. This new software implementation achieves a factor of 20 improvement in processing time over the traditional PNN when running in the standard reference recognition system, and the speed improvement is realized without any loss in classification accuracy.

In the traditional PNN, each training vector (or prototype) $\mathbf{x}_j$ becomes the center of a kernel function that takes its maximum at the vector and decreases gradually as one moves away from the vector in feature space. An unknown feature vector **y** is classified by computing, for each class i containing M_i prototype vectors, the sum of the values of the class-i kernels at **y**, multiplying these sums by factors involving the estimated *a priori* probabilities, and finding which of L classes has the highest resulting discriminant value. PNN assigns the class with the highest discriminant value to the unknown vector **y**.

Many forms are possible for the kernel functions; we have obtained our best results using radially symmetric Gaussian kernels. The resulting discriminant functions are of the form:

$$D_i(\mathbf{y}) = \frac{p(i)}{M_i}\sum_{j=1}^{M_i} exp\left(-\frac{1}{2\sigma^2}d^2\left(\mathbf{y}, \mathbf{x}_j^{(i)}\right)\right) \tag{18}$$

where σ is a smoothing parameter that may be optimized by conducting experiments on a testing set. In this study, σ is assigned 2.0 when classifying digits and 3.0 when classifying alphabetic characters. The *a priori* probability of class i is $p(i)$, and as mentioned earlier, M_i is the number of training prototypes in class i. The euclidean distance between two vectors is defined as

$$d^2(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^{\mathrm{T}} (\mathbf{x} - \mathbf{y}) \qquad (19)$$

The current implementation of the discriminant functions used in *hsfsys* does not use the leading term in Equation (18). Further research is needed to determine if the natural frequencies of character occurrence (in the Constitution box for example) would make good *a priori* probability estimates and improve classification accuracy. If the number of training prototypes within each class are approximately balanced, the denominator of the leading term becomes redundant. Specht[7] shows that the discriminant values can be converted to estimate *a posteriori* probabilities by dividing each discriminant value by their sum such that they add up to 1.0.

Several optimizations have been added by NIST to the traditional PNN implementation in order to decrease computational intensity and improve processing times. The first optimization takes advantage of pruning those prototypes that do not significantly contribute to the computation of discriminant values, and a second optimization utilized a search tree to reduce the number of prototypes used in the discriminant value summation. Due to the presence of the exponential in Equation (18), the closer a training prototype is to the unknown vector, the more significant the prototype's contribution to its discriminant value. In light of this, Equation (18) can be approximated by not including prototypes whose exponential term contributes less than $10^{-\lambda}$ times the largest term. Formally, the j^{th} prototype of any given class can be deleted if:

$$\exp\left(-\frac{1}{2\sigma^2} d^2(\mathbf{y}, \mathbf{x}_j)\right) < 10^{-\lambda} \exp\left(-\frac{1}{2\sigma^2} d^2(\mathbf{y}, \mathbf{x}_c)\right) \qquad (20)$$

where the subscript c denotes the closest training prototype. By taking logs and changing sign, the condition in Equation (20) can be rearranged without the need for computing the exponential function. The resulting test in the distance domain is

$$d^2(\mathbf{y}, \mathbf{x}) > 2\lambda\sigma^2 \ln 10 + d^2(\mathbf{y}, \mathbf{x}_c) \qquad (21)$$

This technique can be used to approximate the traditional PNN in Equation (18). The associated error can be constrained by setting λ to a sufficiently large positive number. This parameter should not be less than $\log(P/L)$, where P is the number of prototypes, and L is the number of classes. The value used in *hsfsys* is $\lambda = 4$. This ensures that classification results will not change between the optimized and traditional PNN implementations.

Two items of importance make using Inequality (21) efficient. First, it is important to note that the training prototype with smallest distance to the unknown vector (thus contributing the maximum exponential term to its discriminant value) is not known *a priori*. The determination of $\mathbf{x}_c$ can be done on the fly, and distances to each prototype only need to be computed once. During the computation of the distances, a list of eligible prototypes (prototypes not yet deleted) can be maintained. Eligible prototypes include the closest to the unknown vector found so far together with all other training prototypes sufficiently close that they do not satisfy the deletion criterion. The deletion test is conducted by substituting $\mathbf{x}_c$ in Inequality (21) with the closest prototype found so far. As the distances between the unknown vector $\mathbf{y}$ and each training prototype $\mathbf{x}_j$ are computed, the new prototype will at times be closer to $\mathbf{y}$ than the closest prototype seen to that point. When this happens, the current prototype $\mathbf{x}_j$ is assigned to be the new $\mathbf{x}_c$ and all eligible prototypes are retested using Inequality (21). Using this single pass technique, the distance $d^2(\mathbf{y}, \mathbf{x}_c)$ can only decrease throughout the process, so prototypes can be safely deleted along the way and, once deleted, they can never become eligible again.

The second item related to the efficiency of Inequality (21) takes advantage of the fact that the distance calculations can be preempted once they become sufficiently large to trigger the deletion criterion. Inequality (21) implies complete calculation of

$$d^2(\mathbf{y},\mathbf{x}) = \sum_{i=1}^{k} (x_i - y_i)^2 \qquad (22)$$

If the distance summation exceeds the deletion criterion, the computation of Equation (22) can stop with $i < k$, as remaining terms contribute nothing more to the outcome of the test. A useful property of the KL transform is that the features are ranked in order of decreasing variance. Therefore, the first few features of a training prototype contribute the most to the distance summation. Typically, only four KL coefficients are required to delete a prototype, and only about 1% of the prototypes are sufficiently close to remain eligible for use in computing discriminant values. Applying these two improvements (on the fly determination of $\mathbf{x}_c$ and preempting distance calculations) makes pruning prototypes very efficient, which in turn greatly reduces the computation of discriminant values. The first optimization step of pruning prototypes achieves a factor of 4 speed up in *hsfsys*.

A further optimization has been integrated into the PNN classifier provided in this distribution. This step utilizes a search tree to reduce the number of prototypes used in the PNN calculations. As stated earlier, PNN discriminant values require the same distance calculations as those used in nearest-neighbor methods. Nearest-neighbor methods for character classification have been shown to be competitive with neural network methods.[22]

In nearest-neighbor methods, we have *N* characters with known identities; each character has been reduced to a feature vector (or point) in *k* dimensions. In practical applications, *N* is large (perhaps 10^6 or so) and *k* is typically in the range 24 to 64. To classify an unknown character, first reduce it to a feature point using a technique such as the KL transform. Then calculate the distances between its point and each of the *N* known points. Any function of the *N* distances and the *N* known classes can be used to classify the unknown character. For example, the unknown character could be assigned the class of the nearest of the *N* known points. PNN itself is another example of such a function and includes some optimal properties. Other functions are described in Reference 22.

This method of classification is expensive; the time is proportional to N, since N distances must be calculated. There is a large literature on faster methods.[23] Among the best are the *k-d tree* methods[24,25] of Bentley, which often have average searching time proportional to log(*N*). For our case, *k* is large and *N* is relatively small (*N* is much smaller than 2^k) and the training points are sparse in *k*-dimensional space. Therefore, the logarithmic behavior is not found. Some slight variations on the k-d tree give searching time proportional to sqrt(*N*), even for large *k*. While not as good as log(*N*), this search time is a substantial improvement over time proportional to *N*. A brief description of this search method is presented here; details may be found in Reference 26.

Construction of the k-d tree is done recursively. The top node contains all *N* points. The two children of this node each contain *N*/2 points. The left child node contains those points whose first feature component has values less than the median of all first components, x_1; the right child node has the remainder. Each of these child nodes is then divided in half using the medians of the second components of the points in the node, and so on. The depth of the tree is $\log_2(N)$, which is less than *k* for our applications. Construction of the tree takes time proportional to *N*log(*N*), but it is done once off-line and stored in a prototype as described in Section 3.2.1.2.

Searching for *m*-nearest neighbors in the k-d tree achieves speed because of being able to avoid calculating distances for entire sub-trees. In k-d trees, rather than searching for the *m* closest points, it is more natural to search for points within distance *d* of the unknown point, *t*, as follows. Start at the top node, and let the unknown point have components t_i. Suppose $t_1 < x_1$. Put the left child node on a list to look at later. All the points in the right child node are at least x_1-t_1 distant from *t*. If x_1-$t_1 > d$, ignore the right child node; otherwise put it on the list. Continue searching by taking one node at a time off the list. If the node has no children, look at the distances of the point or points in the node and remember the *m* smallest distances. If the node has children, look at both child nodes and put one or both of them on the list.

If the distance *d* is excessively large, too few sub-trees will be discarded and too many distances calculated, leading to a long search time. If *d* is excessively small, too many sub-trees will be discarded and too few nearest neighbors will be found, but this calculation is fast. A reasonable approach is to estimate *d*, preferably on the small side. If not enough nearest neighbors are found, *d* is increased and a new search is made. Also, after *m* distances less than *d* have been calculated, *d* can be reduced to the m^{th} largest distance.

To estimate the distance *d*, we use the centroids of each class of known points as trial points and calculate the distances of the unknown point to the trial points. Then we use a fraction, usually around 0.5, of the smallest such distance as an estimate for *d*. In *hsfsys*, the k-d tree is traversed, possibly several times, using increasing factors to widen *d*. These factors are stored in global arrays beginning with the name “tree_cuts”. A different set of cut-off values are used for classifying digits, alphabetic characters, and the mixed upper and lower case Constitution box. These cut-off arrays are found at the top of the distribution file *src/lib/hsf/field.c* and were obtained empirically over a testing set of KL prototypes (feature vectors).

KL prototype vectors and their indexed k-d tree are calculated off-line using the program *mis2pat* discussed in Appendix B. The two optimizations discussed in this section (prototype pruning and k-d tree searching) have been integrated into an optimized PNN procedure *treepnnhypsconsC()* found in the distribution file *src/lib/nn/pnn.c*. This procedure traverses the k-d tree producing a relatively small yet viable set of prototypes. This small set of prototypes is then used to calculate approximated PNN discriminant values according to the deletion criterion defined in Inequality (21). In very rare cases, no close prototypes are found in the tree search. When this occurs, all the training prototypes are used in the approximated PNN calculation. The PNN exponential activations are normalized to estimated probabilities by dividing by their sum and used as classification confidence values.

The optimized version of PNN described in this section runs a factor of 20 times faster than the traditional PNN code, and tests have shown that the gain in speed has not reduced classification accuracy. The optimizations introduced by NIST now enable applications to capitalize on the robustness of the PNN algorithm without compromising processing time.

3.2.3 STORE FIELD RESULTS; src/lib/fet/updatefet.c; updatefet()

The results of character classification are stored in Feature (FET) data structures. Upon completion, *hsfsys* writes the contents of two of these structures to FET files. One FET structure and file hold the system’s hypothesized character classifications and a second FET structure and file hold the confidence values associated with each character classification. FET files are editable ASCII files similar to MFS files and are designed to contain a list of names and a multi-column set of data values that are associated with each name. Every line in an FET file contains a name followed by zero or more values. The names of each entry field on the HSF form comprise the names in the system’s hypothesis and confidence files. For hypothesis files, the values that follow each name are the characters recognized by the system concatenated together without space separators as a single field value. A line in the file that has no value after the field name represents a field that was either not processed or was recognized to be empty. There is a corresponding confidence value reported in the system’s confidence file for every character classified by the recognition system that is reported in the system’s hypothesis file. The confidence values for the characters in a field are space-separated on each line in the FET file. These separators may be a space character 0x20 or a tab character 0x09. Lines are terminated with the line feed character 0x0A. The library *src/lib/fet* contains a suite of routines designed to read and write FET files and manipulate FET structures.

```
typedef struct fetstruct{
    int alloc;
    int num;
    char **names;
    char **values;
} FET;
```

Figure 25. C definition of the FET structure.

Figure 25 lists the C definition of an FET structure that is stored in *include/fet.h*. The structure contains four members. **Names** references an array of character strings corresponding to the names listed in the first column of an

FET file. In the case of *hsfsys*, the names are entry field identifiers. **Values** references an array of character strings holding the values associated with each entry field. For a hypothesis FET structure (one that stores the system's hypothesized character classification), each string in **values** contains the characters recognized by the system for that specific field. For a confidence FET structure (one that stores confidence values), each string in **values** contains the space-separated list of confidence values corresponding to the field value stored in the hypothesis FET. The structure member **alloc** holds the number of allocated positions within **names** and **values**, and **num** holds the number of contiguous positions currently filled in **names** and **values**. In *hsfsys*, the primary routine responsible for manipulating an FET structure is *updatefet()* found in *src/lib/fet/updatfet.c*. It is the responsibility of an application to parse the independent confidence values from a string stored in the **values** array. The FET file convention provides a common I/O interface when manipulating lists of ASCII values that are associated with a common attribute or feature (name). The contents of an FET structure or file can be integers, floating point numbers, names, and/or any sequence of printable ASCII characters.

Hypothesis File

data/f0000_14/f0000_14.nhy

```
hsf_0
hsf_1
hsf_2
hsf_3 0123456789
hsf_4 0123456789
hsf_5 0123456789
hsf_6 86
hsf_7 506
hsf_8 8941
hsf_9 95309
hsf_10 891405
hsf_11 01
hsf_12 707
hsf_13 60170
hsf_14 689547
hsf_15 98
hsf_16 6081
hsf_17 77132
hsf_18 314200
hsf_19 78
hsf_20 464
hsf_21 93849
hsf_22 256369
hsf_23 63
hsf_24 224
hsf_25 6902
hsf_26 551339
hsf_27 78
hsf_28 722
hsf_29 5798
hsf_30 21313
hsf_31 bavxujdyohsmzfcwgiakrezpln
hsf_32 FSHUXTEZRQMLABGVIYPUCOJWH
hsf_33 WE THE PEOPLE THE UNITED
STATES IN FORM A MORE PERFECT UNION
ESTABLISH JUSTICE INSURE DOMESTIC
TRANQUILITY PROVIDE FOR THE COM-
MON DEFENSE OUR THE GENERAL WEL-
FARE AND SECURE THE BLESSINGS OF
LIBERTY TO OURSELVES OUR POSTERITY
DO ORDAIN ESTABLISH THE CONSTITU-
TION FOR THE UNITED STATES AMERICA
```

Confidence File

data/f0000_14/f0000_14.nco

```
hsf_0
hsf_1
hsf_2
hsf_3 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00
hsf_4 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00
hsf_5 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 0.99 1.00
hsf_6 1.00 1.00
hsf_7 1.00 1.00 1.00
hsf_8 1.00 1.00 1.00 1.00
hsf_9 1.00 1.00 1.00 1.00 0.82
hsf_10 1.00 1.00 1.00 1.00 1.00 1.00
hsf_11 1.00 1.00
hsf_12 0.88 1.00 1.00
hsf_13 1.00 1.00 1.00 1.00 1.00
hsf_14 1.00 1.00 1.00 1.00 1.00 1.00
hsf_15 1.00 1.00
hsf_16 1.00 1.00 1.00 1.00
hsf_17 1.00 1.00 1.00 1.00 1.00
hsf_18 1.00 1.00 1.00 1.00 1.00 1.00
hsf_19 1.00 1.00
hsf_20 1.00 1.00 1.00
hsf_21 1.00 1.00 1.00 1.00 0.64
hsf_22 1.00 1.00 1.00 1.00 1.00 1.00
hsf_23 1.00 1.00
hsf_24 1.00 1.00 1.00
hsf_25 1.00 1.00 1.00 1.00
hsf_26 1.00 1.00 1.00 1.00 1.00 1.00
hsf_27 1.00 1.00
hsf_28 1.00 1.00 1.00
hsf_29 1.00 1.00 1.00 1.00
hsf_30 1.00 1.00 1.00 1.00 1.00
hsf_31 1.00 1.00 0.99 1.00 0.99 1.00 1.00 0.99 1.00
1.00 1.00 0.85 0.90 1.00 0.91 1.00 0.50 0.68 0.68 0.98
1.00 0.95 1.00 0.66 0.57 0.99
hsf_32 0.97 1.00 1.00 0.90 1.00 1.00 1.00 1.00 0.99
1.00 1.00 1.00 1.00 0.94 1.00 1.00 1.00 1.00 0.99 0.98
1.00 1.00 1.00 1.00 0.70
hsf_33
```

Figure 26. Example of a system hypothesis and corresponding confidence file.

There are 10 completed HSF forms provide in this distribution under the top-level distribution directory *data*. Hypothesis and confidence files created by *hsfsys* at NIST have been included. Figure 26 lists the contents of the hypothesis file *data/f0000_14/f0000_14.nhy* on the left, and on the right, lists the corresponding confidence file *data/f0000_14/f0000_14.nco*. Notice that confidence values are provided for every field processed except for the Constitution field (hsf_33), which had dictionary-based postprocessing applied. Also note that any line continuations and hyphenation within the values for a single field have been inserted by this document's text formatter and do not actually exist in the files.

3.2.4 DEALLOCATE FOR FIELDS; src/lib/hsf/field.c; free_field()

This step simply deallocates all the memory containing data dependent on the type of field being processed. These are the data items loaded into the system by the INITIALIZE FOR FIELDS step. The memory allocated to all the basis functions and intermediate calculations supporting feature extraction and all the prototypes and class information needed for character classification are deallocated.

3.3 DO LOWER CASE FIELD; src/lib/hsf/field.c; do_alpha_field()

This section describes how *hsfsys* processes fields containing handprinted lower case characters such as field hsf_31 on the HSF form. Figure 27 lists the steps used to process lower case fields and cross-references the steps to the software distribution according to file and subroutine name. Notice that many of the step used in digit field processing are applied here as well. Those steps reused are referenced with heading numbers pointing to the previous sections and will not be discussed in this section.

The difference between processing digit fields and lower case fields are in what feature extraction and character classification files are loaded, and how segmentation is conducted. Lower case feature extraction requires loading the basis function file *weights/td13_l.evt*, and lower case classification requires loading the prototype file *weights/td13_l.pat* and the median vector file *weights/td13_l.med*. Unlike processing digits, the same basis function, prototype, and median vector files are loaded when processing lower case characters regardless if the small memory mode option *"-m"* is specified or not. The differences in segmentation are discussed below.

3.3.1 PROCESS ALPHABETIC FIELD; src/lib/hsf/field.c; process_alpha_field()

Lower case field and digit field processing only differ slightly in how characters are segmented. Otherwise, there is nearly no difference between in the way recognition system processes lower case fields and digit fields. The handprint entered in each field is extracted using the same one-line text isolation routine. The segmented character images are normalized in terms of size and slant using the same techniques applied to digit images. Features are extracted from the lower case character images using the same techniques, and the same PNN classifier is used to classify the feature vectors. Finally, the results of classification are added to the same hypothesis and confidence FET structures that hold the digit field results.

3.3.1.1 SEGMENT ALPHABETIC FIELD; src/lib/hsf/segblob.c; segbinblob()

As can be seen from Figure 27, the general blob extraction routine *segbinblob()* used in segmenting digit fields is the only step used to segment lower case fields. There is no added step of pasting blobs back together after the lower case field is separated into its connected components. This leaves the system vulnerable to the issues of under and over-segmenting discussed in Section 3.2.2.2.2. A lower case segmentor that attempts to subdivide blobs that are too large and merge blobs that are too small would likely improve the system.

3.3 DO LOWER CASE FIELD	**3.3 src/lib/hsf/field.c; do_alpha_field()**
3.2.1 INITIALIZE FOR FIELDS	**3.2.1 src/lib/hsf/field.c; init_field()**
3.3.1 PROCESS ALPHABETIC FIELD	**3.3.1 src/lib/hsf/field.c; process_alpha_field()**
3.2.2.1 ISOLATE 1-LINE FIELD	3.2.2.1 src/lib/hsf/isolate.c; iso_1line_field()
3.3.1.1 SEGMENT ALPHABETIC FIELD	3.3.1.1 src/lib/hsf/segblob.c; segbinblob()
3.2.2.2.1 Extract Blobs	3.2.2.2.1 src/lib/hsf/segblob.c; segbinblob()
3.2.2.3 NORMALIZE CHARACTER IMAGES	3.2.2.3 src/lib/hsf/normaliz.c; norm_2nd_gen()
3.2.2.4 SHEAR CHARACTER IMAGES	3.2.2.4 src/lib/hsf/shear.c; shear_mis()
3.2.2.5 EXTRACT FEATURES	3.2.2.5 src/lib/nn/kl_mis.c; kl_transform_mis()
3.2.2.6 CLASSIFY FEATURE VECTORS	3.2.2.6 src/lib/nn/pnn.c; treepnnhypsconsC()
3.2.3 STORE FIELD RESULTS	**3.2.3 src/lib/fet/updatfet.c; updatefet()**
3.2.4 DEALLOCATE FOR FIELDS	**3.2.4 src/lib/hsf/field.c; free_field()**

Figure 27. Steps to process the lower case field.

3.4 DO UPPER CASE FIELD	**3.4 src/lib/hsf/field.c; do_alpha_field()**
3.2.1 INITIALIZE FOR FIELDS	**3.2.1 src/lib/hsf/field.c; init_field()**
3.3.1 PROCESS ALPHABETIC FIELD	**3.3.1 src/lib/hsf/field.c; process_alpha_field()**
3.2.3 STORE FIELD RESULTS	**3.2.3 src/lib/fet/updatfet.c; updatefet()**
3.2.4 DEALLOCATE FOR FIELDS	**3.2.4 src/lib/hsf/field.c; free_field()**

Figure 28. Steps to process the upper case field.

3.5 DO CONSTITUTION FIELD	**3.5 src/lib/hsf/field.c; do_const_field()**
3.2.1 INITIALIZE FOR FIELDS	**3.2.1 src/lib/hsf/field.c; init_field()**
3.5.1 PROCESS CONSTITUTION FIELD	**3.5.1 src/lib/hsf/field.c; process_const_field()**
3.5.1.1 ISOLATE MULTIPLE LINE FIELD	3.5.1.1 src/lib/hsf/isolate.c; iso_nline_field()
3.3.1.1 SEGMENT ALPHABETIC FIELD	3.3.1.1 src/lib/hsf/segblob.c; segbinblob()
3.2.2.3 NORMALIZE CHARACTER IMAGES	3.2.2.3 src/lib/hsf/normaliz.c; norm_2nd_gen()
3.2.2.4 SHEAR CHARACTER IMAGES	3.2.2.4 src/lib/hsf/shear.c; shear_mis()
3.2.2.5 EXTRACT FEATURES	3.2.2.5 src/lib/nn/kl_mis.c; kl_transform_mis()
3.2.2.6 CLASSIFY FEATURE VECTORS	3.2.2.6 src/lib/nn/pnn.c; treepnnhypsconsC()
3.5.1.2 BUILD PHRASE LISTS	3.5.1.2 src/lib/phrase/bld_pis.c; build_pi_lists()
3.5.1.2.1 Find Phrases	3.5.1.2.1 src/lib/phrase/find_pis.c; find_pi_lists()
3.5.1.2.2 Merge Phrases	3.5.1.2.2 src/lib/phrase/merg_pis.c; merge_pi_lists()
3.5.1.2.3 Sort Phrases Top To Bottom	3.5.1.2.3 src/lib/phrase/sort_pis.c; sort_pi_lists_on_y()
If Dictionary Option ON	*If Dictionary Option ON*
3.5.1.3 CORRECT AND IDENTIFY WORDS	3.5.1.3 src/lib/phrase/spellphr.c; spell_phrases2()
For Each Phrase	*For Each Phrase*
3.5.1.3.1 Spell-Correct Line Of Text	3.5.1.3.1 src/lib/dict/line.c; spell_line2()
3.2.3 STORE FIELD RESULTS	**3.2.3 src/lib/fet/updatfet.c; updatefet()**
3.2.4 DEALLOCATE FOR FIELDS	**3.2.4 src/lib/hsf/field.c; free_field()**

Figure 29. Steps to process the Constitution field.

3.4 DO UPPER CASE FIELD; src/lib/hsf/field.c; do_alpha_field()

The steps used to process the upper case field hsf_32 on the HSF form are nearly identical to those used to process lower case fields. The only difference is in what feature extraction and character classification files are loaded. Upper case feature extraction requires loading the basis function file *weights/td13_u.evt*, and upper case classification requires loading the prototype file *weights/td13_u.pat* and the median vector file *weights/td13_u.med*. When processing upper case characters the small memory mode option "*-m*" is ignored. The heading numbers referenced in Figure 28 point back to steps discussed in previous sections.

3.5 DO CONSTITUTION FIELD; src/ib/hsf/field.c; do_const_field()

The steps required to process the Constitution field (hsf_33) are listed in Figure 29. Again, there are a number of steps that are used here that have already been discussed in previous sections. Two things make processing the Constitution field different from the digit, lower case, and upper case fields. They are the processing of multiple lines of text within the same field and the optional dictionary-based postprocessing. All other steps apply the same techniques. Feature extraction requires loading the basis function file *weights/ul.evt*, and character classification requires loading the prototype file *weights/td13_ul.pat* and the median file *weights/td13_ul.med*. If the small memory mode option is used to invoke *hsfsys*, a smaller set of prototypes and their associated files are loaded instead. These files begin with the root file name *td3_ul_s* in the top-level directory *weights*. The basis functions and prototype files used to process the Constitution field have been designed to assign both lower and upper case instances of the same character with a single upper case classification. For example, an H and an h are both classified as H. This was done because people frequently switch between lower and upper case when handprinting textual information.

3.5.1 PROCESS CONSTITUTION FIELD; src/lib/hsf/field.c; process_const_field()

Processing the Constitution field is different from the previous types of fields because it involves handling multiple lines of text within the same field, and contextual postprocessing at the word-level is possible. Even with these differences, there is still a large overlap with the steps already discussed. The character segmentor simply extracts blobs using the connected component utility. The segmented character images are normalized in terms of size and slant using the same techniques discussed earlier. Features are extracted from the segmented character images using the KL transform, the same optimized PNN classifier is used to recognize the feature vectors, and the results of classification are added to the same hypothesis and confidence FET structures that hold the previous field results. Confidence values are reported for the raw OCR results, but no confidence values are reported when dictionary-based postprocessing is performed.

3.5.1.1 ISOLATE MULTIPLE LINE FIELD; src/lib/hsf/isolate.c; iso_nline_field()

Three of the six registration marks used to register the image lie on corners of the Constitution box at the bottom of the HSF form. Because of this, the form removal is quite accurate at removing the black pixels comprising the Constitution box from the input image. The spatial field template stored in *tmplt/hsftmplt.pts* is used to extract the field subimage. The handprinted data within the field is then isolated using spatial histogram techniques like the ones used to locate the left and right ends of the handprinted text in a one-line text field. Left, right, top, and bottom edges are found by searching horizontal and vertical histogram bins directly. Searching inward from the beginning or end of the histogram bins, the first bin greater than 10 pixels is located, and then a reverse search from that point locates the first bin that equals zero. The use of a 10 pixel threshold avoids speckle noise in the field and locates the beginning of significant character data, while the reverse search locates the edge of the character data. *Hsfsys* extracts the subimage bounded by these left, right, top, and bottom edges.

3.5.1.2 BUILD PHRASE LISTS; src/lib/phrase/bld_pis.c; build_pi_lists()

The connected component utility used to segment the isolated Constitution field returns blobs in column-major order. This section describes the steps used to sort the blobs into correct reading order. Initially it was anticipated that a simple sort of the x and y center coordinates of each blob would be sufficient to organize the blobs into reading order (left-to-right and top-to-bottom). Unfortunately, it was found that the handprint in the Constitution field often fluctuates significantly within lines as well as across lines, and this fluctuation is exaggerated by the use of punctuation marks, causing techniques that use global line statistics to fail.

A localized point-to-point technique was developed to organize the segmented blobs into phrases (text lines). The method is divided into three steps. First, the extracted blobs are collected into segments of text lines. Second, the phrase segments are merged into complete lines of text. Finally, the text lines are sorted top-to-bottom so that the order of the blobs within the lines correctly reconstruct the sequence of characters in the text paragraph.

3.5.1.2.1 Find Phrases; src/lib/phrase/find_pis.c; find_pi_lists()

The connected component utility produces segmented character images. Each segmented image has assigned to it the position where it was extracted from the isolated field image. The location of each blob is identified by computing the geometric center of the smallest rectangle bounding the blob. Adding each blob's center to the location from where the blob was extracted, produces a 2-dimensional grid of blob centers that can be used to reconstruct the line trajectories of the handprinted text.

The process of organizing the blob centers into text lines is referred to as Adaptive Sequence Reconstruction. This technique searches the 2-dimensional grid of blob centers taking into account local writing fluctuations to sort the blobs into correct reading order. A point-to-point search is conducted based on a local search space defined by the function:

$$S = a\cos(b\theta) \qquad -\frac{\pi}{2b} < \theta < \frac{\pi}{2b} \qquad (20)$$

This function, which is similar to an antenna sensitivity model, forms a tear-drop shaped bubble that is desirable for this application because it is horizontally biased. The interior of the function is used as a locally constrained search space. Through empirical study a technique for controlling the shape of the *S* function was developed. At values of *b* near 0.1, the function's shape is circular, and as *b* increases the shape continuously forms into a tear-drop. The variable, *a*, controls the length of the bubble along its horizontal axis of symmetry. By increasing *a*, the length of the bubble is increased and the search is extended in the horizontal direction.

A linear control function, *b=L(a)*, is used to modify the shape of the bubble, *b*, as the length of the bubble, *a*, is increased. This function is defined by the slope of the line connecting two empirically derived points. One point used to define the control line, (*a*1, *b*1), is calculated:

$$a1 = h \times 0.375 \qquad b1 = 0.1 \qquad (21)$$

where *h* is the average blob height for the writer. If for example the writer's average blob height is 32 pixels, this point on the linear control function defines a circular bubble with a radius of 12 pixels (1mm). The second point used to define the control line, (*a*2, *b*2), is calculated:

$$a2 = h \times 4.7 \qquad b2 = 2.0 \qquad (22)$$

If the writer's average blob height is 32 pixels, this second point defines a tear-shaped bubble with a horizontal length of 150 pixels. If a writer's handprint is small, the bubbles used in the search are adapted to be smaller, and if a writer's handprint is large, the bubbles used in the search are adapted to be larger. In addition, the bubble defining the search space continuously changes from circular to tear-drop in shape as the extent of the search increases.

Using the linear control function *L*, the size and shape of the bubble can be continuously modified as shown in Figure 30. In these three examples, average blob heights of 16, 32, and 48 are used, respectively. If a search is to be conducted relative to the right of a blob's center, then only the portion of the function with x>0 is used. If a search is to be conducted relative to the left of a blob's center then the portion of the function with x<0 is used. The search is conducted by initializing *a* to a starting length and then testing to see if any other blob centers are located within the boundary of the bubble. If points are found, then the nearest blob is selected. Otherwise, *a* is incremented and the bubble is enlarged and lengthened and a test for blobs in the new bubble is conducted. This continues until a center point of a neighboring blob is found, or *a* exceeds some threshold. In Figure 30, successive bubbles are overlaid from a common center point where *a* is initialized to 12, incremented by 20, and terminated at 300.

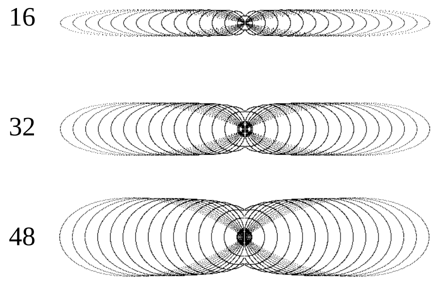


Figure 30. Adaptation of bubbles to the writer's average blob height.

The search begins from the blob center closest to the top-left corner of the field. The blob is added to an empty list, and a bubble is initialized, tested, and then grown via incrementing *a* until either a neighboring blob center is found or *a* exceeds a threshold. The threshold used in this system is 300 which is equal to 1 inch (12 pixels per millimeter equals 300 pixels per inch). In general, the new blob is added to the list and the search resumes from the center of the new blob. However, if the new blob's center does not meet given criteria, then its center is added to the list, but the search continues from the current blob center and does not advance to the center of the new blob. This way the system does not naively follow erratic line trajectories, minimizing the chances of crossing over into adjacent lines, such as may happen when a comma is found.

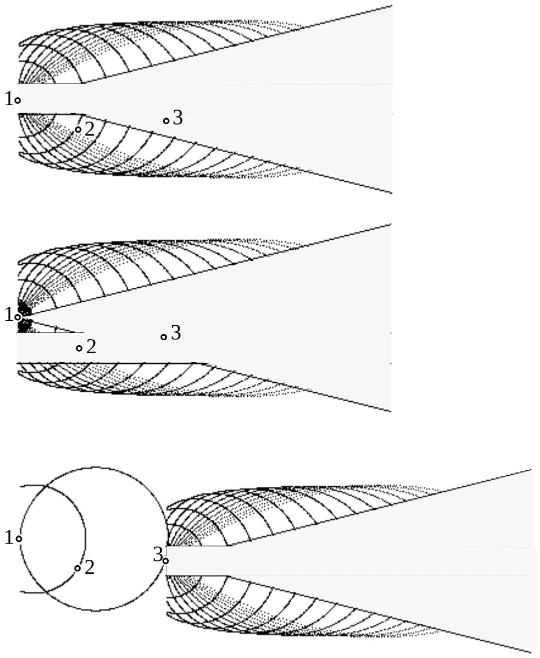


Figure 31. Heuristic used to control the advancement of the search.

Figure 31 illustrates this heuristic as it is used to control the advancement of the search. In order to advance, the new blob center must be within the area defined by the union of two region. The first region is the area bounded by two lines with slope -0.25 and +0.25 projecting from the current blob center. The second region is the area bounded by two horizontal lines with y-intercepts at -(0.25**h*) and (0.25**h*), centered about the last blob added to the list. The top diagram in Figure 31 show bubbles projected from the current blob center (1). The closet neighboring blob center is (2), however (2) is not within the given criteria represented by the region filled with gray. Therefore, (2) is added to the list with (1), but the current bubble position does not advance to (2). The middle diagram in Figure 31 shows the search continuing with bubbles being projected from (1). The next closest blob center is (3). Notice that the gray region has changed from the top diagram. The triangular slope-based region remains anchored to (1), but the horizontal region, based on the writer's average blob height, is now defined in relationship to (2). Blob center (3) is added to the list with (1) and (2), and because (3) is within the new gray region, the current blob center advances to (3) as shown in the bottom diagram in Figure 31. Note that once a blob center is added to the list, it is not considered again in the search process.

It was observed during the development of this approach that the heuristic described above, when tuned to handle isolated cases, did not yield proper results in other cases. It was determined that as local fluctuations in the hand-print become excessive, rather than force the system to make a *guess*, the point-to-point search should be preempted. The search is restarted from a blob not yet included in any lists and closest to the top-left of the image. This action is also taken at the end of a text line when no new neighboring blob centers are found to the right of the current blob. Each restart involves starting a new list, and the entire search process is terminated when every blob in the image has been assigned to a list.

The criterion for preempting the search and beginning a new list is illustrated in Figure 32. In this illustration, the search is currently being conducted from blob center (2), and (3) has been located as the next nearest neighbor. In the previous step, (2) was found by searching from (1), and both (1) and (2) have been added to the current list. The distance, *d1*, is the vertical distance between (1) and (2), and the distance, *d2*, is the vertical distance between (2) and (3). The two parallel horizontal lines in the diagram represent the area bounded by -*h* and +*h* centered about the previous blob center (1). The sum of ($d1+d2$), the vertical distance between (1) and (3), exceeds the limit, *l*, representing the region bounded by the horizontal lines; therefore the search is preempted and (3) is not added to the current list. Blob (3) is left unassigned so that it can be added to a list later in the search process.

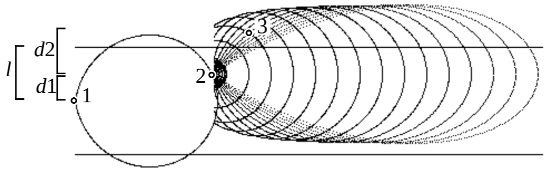


Figure 32. Heuristic used to preempt the search.

It is surprising how well this preemptive heuristic works. At times, more frequently with some writers than with others, the local writing fluctuations become excessive and the search is restarted. Often the restart resumes on the next line and the point-to-point search is successful in tracking the next line. The search is top-down by nature, so that the blobs in the line above the area of excessive fluctuation are likely to be assigned to a previous list. Eventually the left-most blob involving the fluctuation is the closest remaining blob to the top-left of the field, and the point-to-point search resumes from that blob. All neighboring blobs from the line above and the line below have been previously assigned leaving only the blobs comprising the fluctuation exposed. This greatly reduces the system's *guesswork* and thereby reduces system errors. Figure 33 shows the results of segmenting the Constitution field in Figure 1 and using the bubble technique to sort the blobs into lines. A bubble is traced from each point where a neighboring blob was found, and each bubble reflects the actual size and the shape of the search space used to locate the neighbor.

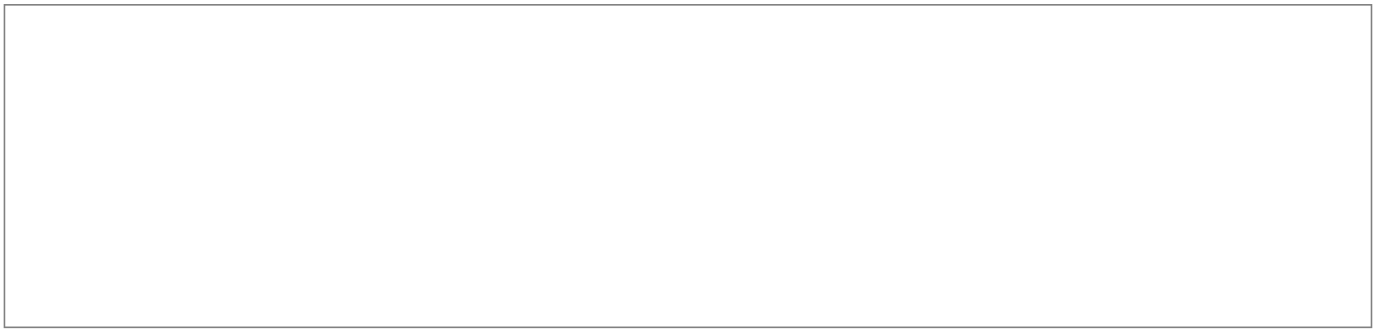


Figure 33. Traces of the bubbles used to sort the blobs into lists.

3.5.1.2.2 Merge Phrases; src/lib/phrase/merg_pis.c; merge_pi_lists()

The point-to-point search produces multiple lists of blobs. Some of the lists represent complete lines of text, and other lists may represent only fragments of the text lines printed. A final merging process is required so that, upon completion, only lists containing complete text lines remain. Two heuristics are used for merging the blob lists; they are illustrated in Figure 34 and Figure 35. The lists are sorted in descending order according to the number of blobs in each list. The longest list is first compared against all other lists, applying the first heuristic and then the second to each comparison. If two lists meet the merging criteria, they are merged and the looping process is restarted by resorting the lists. Otherwise, the next longest list is compared to the remaining shorter lists, and so on until all the lists are looped through and no merging takes place. When two lists are merged, their blob centers are appended into one larger list and then sorted on their x-coordinates. Figure 34 illustrates the merging of two when the end point of the shorter list is within a vertical distance of -(0.75**h*) and +(0.75**h*) of the longer lists's corresponding end point.

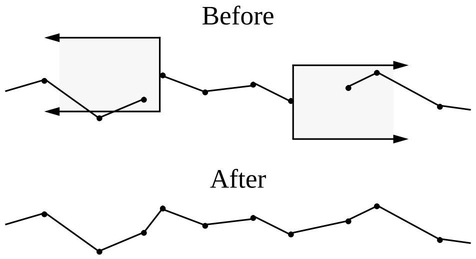


Figure 34. Heuristic for merging blob lists based on end point positions.

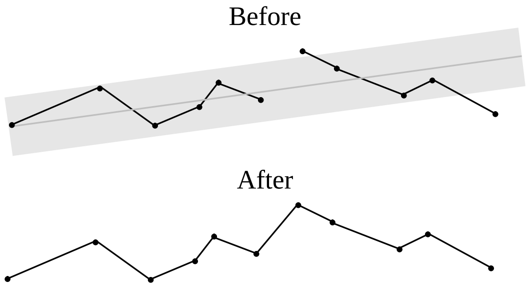


Figure 35. Heuristic for merging blob lists based on line trajectories.

The second merge heuristic is illustrated in Figure 35. In this case, the blob centers comprising the longer list are fitted using linear least squares to produce a slope and y-intercept. A perpendicular distance is computed between each blob center in the shorter list and the line fitted to the longer list, and the distances less than (0.5**h*) are counted.

This area along the fitted line is represented by the gray region in the diagram. The two lists are merged if the count is greater than (0.1**n*), where *n* is the number of blobs in the longer list. This facilitates merging of lists that lie along the same line trajectory, but whose end points are somewhat erratic.

3.5.1.2.3 Sort Phrases Top To Bottom; src/lib/phrase/sort_pis.c; sort_pi_lists_on_y()

As a result of the previous two steps, the blobs segmented from the Constitution field are now gathered and sorted into lines. The last step is to sort the resulting lines vertically. The lines are sorted based on the y-coordinate values of the first blob in each line. When finished, the correct reading order has been reconstructed.

To conduct the various sorts in *hsfsys*, a multiple-indexed recursive quick sort utility has been provided with this software distribution. The utility is *multisort()* and is found in *src/lib/util/multsort.c*. The utility is capable of sorting on up to 5 integer keys (primary, secondary, etc.), and the values sorted can be an array of integers or an array of pointers. Each of the 5 keys can be sorted independently increasing or decreasing. A macro-based interface has been developed to help the caller access the flexible capabilities of this utility. The macro definitions are stored in *include/multsort.h* and examples of how they are used can be seen in *src/lib/util/sortindx.c*.

3.5.1.3 CORRECT AND IDENTITY WORDS; src/lib/phrase/spellphr.c; spell_phrases2()

No contextual information has been use up to this point by the recognition system other than knowing the type of each field (digit, lower case, upper case, or Constitution). Of these field types, only the Constitution box has data that can be processed using language or word models. A dictionary-based postprocessing capability has been integrated into the system, and it can be optionally selected from the command line as described in Section 2.4.

If dictionary-based postprocessing is not selected, the system stores the raw hypothesized character classifications and their corresponding confidence values to the output FET structures, which upon completion, are written to hypothesis and confidence files. If dictionary-based postprocessing is selected then the raw character classifications are further processed. The system's hypothesized classifications are prone to errors. These errors are introduced when the connected components over and under-segment the field image and when the classifier assigns an incorrect class to a properly segmented character image. By using a dictionary, many of these errors can be corrected.

3.5.1.3.1 Spell-Correct Line Of Text; src/lib/dict/line.c; spell_line2()

The dictionary-based postprocessing described in this section is referred to as Word Identification using Fanout Signals. Up to this point in the system, the handprinted text within the Constitution box has been isolated, the extracted field image has been segmented into blobs, and the resulting blob images have been sorted into correct reading order. Each blob image has also been size and slant-normalized, having its features extracted and classified. These steps were applied to the image in Figure 33 producing the text shown in Figure 36. Notice that the character classifications are all upper case due to the merged upper and lower case classes in the prototype file *weights/ul.kl;* notice the large number of errors contained in classifier output; and also notice there are no inter-word spaces recognized at this point in the process. The dictionary-based postprocessing has been developed to correct these classification errors and to detect word boundaries within text lines like the ones shown in Figure 36.

```
Line 1:    ILAUTHEPEOPLEOFTHEUHLTEASTCTESLNORDERTOFORMAMORE
Line 2:    PTRFECTUHOHLOTABLLSHJUSTICELLNSUREDOMESTLC
Line 3:    TRSNQUILHTYJPROIDEFURTHECOMAONDEFMSEPROLNURERHE
Line 4:    GENIRALWWMRCANDSEEURTTHEBIISSLNGSOFLLBEHTYTO
Line 5:    OURSEIVOSANDOURPOSTLRITYDOORDALNCNDNTABIISH
Line 6:    MIJUNSTITUTIONFORGEUNLTEDSRARESMFAMERDCA
```

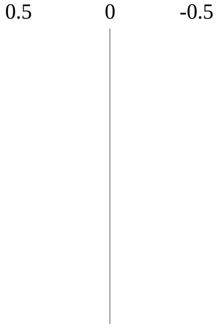
Figure 36. Example of classifier output prior to contextual processing.

The Preamble to the U.S. Constitution is comprised of 38 unique words, and these words are used to construct the dictionary (lexicon) shown in Figure 37. The lexicon is used to detect words within text lines, identifying word boundaries and correcting any segmentation and classification errors existing within the text lines.

A
AMERICA
AND
BLESSINGS
COMMON
CONSTITUTION
DEFENSE
DO
DOMESTIC
ESTABLISH
FOR
FORM
GENERAL
IN
INSURE
JUSTICE
LIBERTY
MORE
OF
ORDAIN
ORDER
OUR
OURSELVES
PEOPLE
PERFECT
POSTERITY
PROMOTE
PROVIDE
SECURE
STATES
THE
THIS
TO
TRANQUILITY
UNION
UNITED
WE
WELFARE

Figure 37. Lexicon constructed from the text contained in the Preamble to the U. S. Constitution.

The technique is illustrated by the example shown in Figure 38. In this example, a portion of the first line of text in Figure 36, “STCTESLNORDE”, is being processed. The graph plots the floating point numbers listed in the first column. These numbers form a signal which is processed in order to locate words within the text. The generation of these signals will be discussed later. The second column is a fan-out of hypothesized words beginning with the character S and adding one successive character from the text line forming a new hypothesized word on each row down the column. The maximum length of a hypothesized word is 12 characters, which is the length of the longest word in the lexicon, “CONSTITUTION”. The third column lists the best match from the lexicon for each hypothesized word in the second column. The fourth column lists alignments that are produced using the Levenstein Distance to match the hypothesized word to the lexicon match. In the alignments, 0 represents a correct character, 1 represents a substituted character, 2 represents an inserted character, and 3 represents a deleted character. These alignments are used to generate the signals listed in the first column and plotted in the graph.



Signal	Hypothesis	Match	Alignment
-0.230	S	THIS	2220
-0.127	ST	STATES	022022
-0.174	STC	STATES	022021
-0.031	STCT	STATES	001022
0.111	STCTE	STATES	001002
0.254	STCTES	STATES	001000
0.096	STCTESL	STATES	0010003
0.040	STCTESLN	STATES	00100033
-0.005	STCTESLNO	STATES	001000333
-0.043	STCTESLNOR	STATES	0010003333
-0.075	STCTESLNORD	STATES	00100033333
-0.103	STCTESLNORDE	STATES	001000333333

Figure 38. Signals generated from a fan-out of hypothesized words.

A signal value, *s*, is computed from two terms, *e* and *t*. The first term, *e*, is an error term and is computed:

$$e = \frac{n}{l-g} \tag{23}$$

where *n* is the number of errors (1’s, 2’s, and 3’s) in a hypothesized word’s alignment, *l* is the total number of characters in the alignment, and *g* is the number of contiguous groupings of 1’s and 3’s. The Levenstein Distance strictly minimizes the amount of error in the alignment without regard for the resulting configuration of alignment elements. The variable *g* is used to favor hypothesized words whose alignments contain contiguous groupings of correct characters (0’s) over alignments containing many discontinuities.

The second term used to compute the signal is *t*. This is a translation term based on the linear function, *T*, that biases longer hypothesized words over shorter ones. In this way, matches to the word “DOMESTIC” are favored over matches to the word “DO”, and “INSURE” is favored over the word “IN”. The linear translation function used in this study is defined by the empirically derived points (2, 0.5) and (12, 0.4); such that *t*=0.5 for hypothesized words of length 2, and *t*=0.4 for hypothesized words of length 12. The translation term is determined by locating the point on the line at the position corresponding to the length of the hypothesized word's dictionary match. If *p* is the length of the dictionary match, then *t=T(p)*. Signal value, *s*, is then computed:

$$s = 1.0 - e - t \tag{24}$$

The signals listed in the first column of Figure 38 are searched top to bottom. Only those hypothesized words with *s*>0 are considered to contain possible words. All other hypothesized words in the fan-out are ignored. The hypothesized word with the largest signal strength is selected. If this word is a substring of a hypothesized word further down the list, such as "DO" in "DOMAIN", and the word containing the substring has a signal strength, *s*>0, then the longer word is selected in place of the word with maximum signal.

Once a hypothesized word is selected from the fan-out, the lexicon match for that word is pushed onto a stack and the alignment is used to synchronize the processing. As can be seen in Figure 38, characters that match between the hypothesized word and the lexicon match are represented by 0's. The alignment elements between the left-most 0 and the right-most 0 comprise an *alignment span*, and it corresponds to the characters in the lexicon match. The ends of this alignment span demarcate the boundaries of the word within the original line of text. Processing the signals in this fashion is done recursively. If a portion of the fan-out remains to the left of the selected word's alignment span, then the remaining piece of fan-out may contain another word. Remember the maximum hypothesized word is 12 characters which is long enough to hold 3 or 4 small words from the lexicon simultaneously. The remaining left portion is processed recursively, recalculating new signal values and searching for words within that piece. As lexicon matches are selected, they are pushed onto the stack.The recursion continues until all of the fan-out to the left of the top-level selected word have been exhaustively processed. The selected lexicon matches are then popped off the stack in correct reading order, and a new fan-out is rebuilt beginning with the first character to right of the top-level selected word's alignment span. For example in Figure 38, the hypothesized word, "STCTES" is selected with a maximum signal of 0.254. The next fan-out will begin with L, starting from the position in the text line, "LNORDERTOFOR". If no hypothesized words are selected within the current fan-out, then the processing advances one character in the text line, and the fan-out begins from that point.

Through this approach, segmentation and classification errors are corrected, and word boundaries are automatically identified. The results of the dictionary-based postprocessing are stored to the hypothesis FET structure. All the words recognized by processing fanout signals are concatenated together into a single string separated by spaces and stored as hsf_33 field's value in the hypothesis FET. No confidence values are stored in the confidence FET structure. An example of the results of dictionary-based postprocessing can be seen in Figure 40. These results were obtained from the raw classifications shown in Figure 39.

```
WETHEPEOPIEOPTHRUNIIEDSTATESIINOTTORMAMOREIPEHECZUNONIESEEBLIIHJUS
TICEIINJUREDOMESIICTRANGUIICPROVIHFORTHECTMMONDETENEIPKOMRCETHEY
TENERALWELFUEZNDSEWRETHCBKSSINDJOFLLBERTTOJOVRSELVUIDOURPOSTERI
YRIDOORJAINMDESTZBLISLTHOCONSTITUTIONFORTHTUNZEDSTNTESOTAMMICA
```

Figure 39. The raw classifications from running *hsfsys* on the form image *data/f0000_14/f0000_14.pct*.

```
WE THE PEOPLE THE UNITED STATES IN FORM A MORE PERFECT UNION ESTABLISH JUS-
TICE INSURE DOMESTIC TRANQUILITY PROVIDE FOR THE COMMON DEFENSE OUR THE
GENERAL WELFARE AND SECURE THE BLESSINGS OF LIBERTY TO OURSELVES OUR POS-
TERITY DO ORDAIN ESTABLISH THE CONSTITUTION FOR THE UNITED STATES AMERICA
```

Figure 40. The results of dictionary-based postprocessing on the raw classifications shown in Figure 39.

4. PERFORMANCE AND TIMING STATISTICS

Running *hsfsys* produces a hypothesis file and a confidence file. The hypothesis file contains the characters recognized in each field on the input HSF form, and the confidence file contains the confidence values produced for each character classification reported in the hypothesis file. Both of these are FET files and are compatible as input files to the NIST Scoring Package.[27-30]

In a sample of the first 500 writers in SD1, the system achieves a character output accuracy of 92.9% (59308/63830) on digit fields with no character rejections, where character output accuracy CHAR8[31] is defined to be:

$$CHAR8 = \frac{AC_{char}^{chrrec}}{total_{refchr}} \tag{25}$$

Running the recognition system using the small memory mode option, *hsfsys* achieves a character output accuracy of 92.3% (58916/63830) with one third the training prototypes used by default. The numerator represents all the segmented character images correctly classified by the recognition system that are not rejected. The denominator represents the total number of characters that can possibly be recognized on the completed forms. The system achieves a character output accuracy of 75.3% (9611/12766) on lower case fields and 84.5% (10784/12766) on upper case fields without the use of context-based postprocessing.

Hsfsys achieves a character decision accuracy of 95.4% (59308/62167) with no rejections, where character decision accuracy CHAR3 is defined to be:

$$CHAR3 = \frac{AC_{char}^{chrrec}}{AC_{char}^{chrrec} + AI_{char}^{chrrec}} \tag{26}$$

Equation (26) has the same numerator as Equation (25), but the denominator represents the total number of segmented character images presented to the system's classifier that are not rejected. Equation (26) does not include character deletions within the system. At a rejection rate of 4.6%, the system achieves a character decision accuracy of 97.4% (57671/59217).

The system achieves a field accuracy of 79.1% (10878/13748) with no characters rejected, where field accuracy CHRFLD1 is defined to be:

$$CHRFLD1 = \frac{AC_{chrfld}^{fldrec}}{total_{chrfld}} \tag{27}$$

The numerator of Equation (27) represents the total number of fields correctly recognized by *hsfsys*. In order for a field to be considered correctly recognized, no remaining characters in the field value after rejection can be substituted, inserted, or deleted. The denominator represents the total number of fields requested to be recognized by the system.

The recognition system achieves a word accuracy of 60.5% (15439/25532) when applying a limited-size dictionary to the character classifications made on the Constitution paragraph. Running the recognition system using the small memory mode option, *hsfsys* achieves a word accuracy of 59.0% (15076/25532) with one half the training prototypes used by default. The word accuracy is computed by tokenizing each word, using the Scoring Package to align the word tokens, and then accumulating the number of substituted, inserted, and deleted words.

A timing option can also be selected when invoking *hsfsys*, in which case a timing file is produced upon system completion. Figure 41 shows the collective time spent on each major task within the recognition system. The times recorded in the timing file are actually reported at a much finer detail, so that many useful tables can be compiled for conducting various analyses. These timing results were achieved on an SGI Challenge (IP19) computer listed in Figure 42. Notice that most of the time is spent classifying characters (29.2%) and conducting dictionary-based postprocessing (26.0%). Figure 42 lists all the different computers on which the recognition system was successfully compiled and tested. The last column in the table shows the average user time required by each machine to process a single form. Theses averages were compiled from the times produced on the 10 HSF form provided in the top-level directory *data*.

FORM TIMES

Task	**Sec.**	**%**
initialize :	0.55	2.5
register :	2.71	12.5
remove :	0.41	1.9

FIELD TIMES

Task	**Sec.**	**%**
initialize :	0.28	1.3
isolate :	0.52	2.4
segment :	2.02	9.3
normalize :	0.43	2.0
shear :	0.07	0.3
feature :	2.58	11.9
classify :	6.32	29.2
sort :	0.07	0.3
spell :	5.62	26.0
total :	21.630	100.0

Figure 41. Report compiled from timing statistics generated by *hsfsys* on an SGI Challenge (IP19).

Man.	**Model**	**O.S.**	**# Proc.***	**RAM**	**Time**
DEC	Alpha	OSF/1 V1.3	1	32 Mb	28.3
HP	Model 712/80	HP-UX 9.03	1	64 Mb	31.4
IBM	RS6000	AIX 3.2.5	1	128 Mb	27.4
SGI	Challenge (IP19)	IRIX 5.2	8	512 Mb	22.9
SGI	Indigo 2 (IP22)	IRIX 4.0.5H	1	128 Mb	26.4
SGI	Onyx (IP19)	IRIX 5.1.1.3	4	512 Mb	22.4
Sun	SPARCserver 4/470	SunOS 4.1.1	1	32 Mb	125.9
Sun	SPARCstation IPC	SunOS 4.1.2	1	8 Mb	169.5**
Sun	SPARCstation 2 (Weitek 80MHz CPU)	SunOS 4.1.3	1	64 Mb	81.8
Sun	SPARCstation 10	SunOS 4.1.3	1	32 Mb	63.0
Sun	SPARCstation 10	SunOS 5.2 (Solaris)	2	128 Mb	39.6

Figure 42. Table of timings from different computers on which the standard reference recognition system has been successfully ported and tested.

*All computers, including those with multiple processors, were compiled and tested serially.

**The Sun IPC was run using the small memory mode option due to its limited memory.

5. FINAL COMMENTS

A number of NIST Internal Reports (NISTIRs) have been referenced in this document. These reports are provided in PostScript format in the top-level directory *doc*. The file *doc/hsfsys.ps* contains this specific document. These reports along with many other NIST Image Recognition Group publications are available in PostScript format across the network via anonymous ftp on *sequoyah.ncsl.nist.gov.* To request a paper copy of any of these NISTIRs, please contact:

CSL Publications
NIST
225/B151
Gaithersburg, MD 20899
voice: (301) 975-2821

This report documents the NIST standard reference recognition system *hsfsys* in terms of its installation, organization, and functionality. The system has been successfully compiled and tested on a number of different vendors' UNIX workstations. It is the responsibility of the distribution recipient to port the software to their specific computer architecture. The source code is written primarily in C with two supporting utilities containing FORTRAN components. The standard reference recognition system is organized into 11 libraries. In all, there are approximately 19,000 lines of code supporting more than 550 subroutines. Source code is provided for a wide variety of utilities that have application to many other types of problems.

Distributions of the NIST standard reference recognition system can be obtained free of charge on CD-ROM by sending a letter of request to the primary author. Requests for distribution made by electronic mail will not be accepted; however, electronic mail is encouraged for technical questions once the distribution has been received. This system or any portion of this system may be used without restrictions because it was created with U.S. government funding. Redistribution of this standard reference system is strongly discouraged as any subsequent corrections or updates will be sent to registered recipients only. This software was produced by NIST, an agency of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibilities associated with its operation, modification, and maintenance.

6. REFERENCES

1. Department of Defense, “Military Specification - Raster Graphics Representation in Binary Format, Requirements for, MIL-R-28002,” 20 Dec 1988.

2. CCITT, “Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus, Fascicle VII.3 - Rec. T.6,” 1984.

3. M. D. Garris. Design and Collection of a Handwriting Sample Image Database, *Social Science Computer Review*, Vol. 10, pp. 196-214, Duke University Press, 1992.

4. M. D. Garris, Design, Collection, and Analysis of Handwriting Sample Image Databases, *Encyclopedia of Computer Science*, to be published, 1994.

5. M. D. Garris and D. L. Dimmick. Evaluating Form Designs for Optical Character Recognition. Technical Report NISTIR 5364, National Institute of Standards and Technology, February 1994.

6. P. J. Grother. Karhunen Loeve Feature Extraction for Neural Handwritten Character Recognition. In *Proceedings: Applications of Artificial Neural Networks III*, Vol. 1709, pp. 155-166. SPIE, Orlando, April 1992.

7. D. F. Specht. Probabilistic Neural Networks. *Neural Networks*, Vol. 3(1), pp 109-119, 1990.

8. H. G. Zwakenberg. Inexact Alphanumeric Comparison. *The C Users Journal*, pages 127-131. May 1991.

9. M. D. Garris. Unconstrained Handprint Recognition Using a Limited Lexicon. In Proceedings: Document Recognition, Vol. 2181, pp. 36-46, SPIE, February 1994.

10. C. L. Wilson and M. D. Garris, “Handprinted Character Database,” *NIST Special Database 1*, **HWDB**, April 18, 1990.

11. M. D. Garris and R. A. Wilkinson. Handwritten Segmented Characters Database. Technical Report Special Database 3, **HWSC**, National Institute of Standards and Technology, February 1992.

12. J. Geist, R. A. Wilkinson, S. Janet, P. J. Grother, B. Hammond, N. W. Larsen, R. M. Klear, M. J. Matsko, C. J. C. Burges, R. Creecy, J. J. Hull, T. P. Vogl, C. L. Wilson. The Second Census Optical Character Recognition Systems Conference. Technical Report NISTIR 5452, National Institute of Standards and Technology, May 1994.

13. ISO-9660. Information Processing - Volume and File Structure of CD-ROM for Information Interchange. Standard by the International Organization for Standardization, 1998.

14. C. R. Wyle. *Advanced Engineering Mathematics*, Second Edition, pp. 175-179, McGraw-Hill, New York, 1960.

15. A. K. Jain. *Fundamentals of Digital Image Processing*, pp. 384-389, Prentice-Hall, New Jersey, 1989.

16. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes, The Art of Scientific Computing (FORTRAN Version)*. pp. 349-363, Cambridge University Press, Cambridge, 1989.

17. M. D. Garris, C. L. Wilson, J. L. Blue, G. T. Candela, P. Grother, S. Janet, and R. A. Wilkinson. Massively Parallel Implementation of Character Recognition Systems. In *Conference on Character Recognition and Digitizer Technologies*, Vol. 1661, pp. 269-280, SPIE, San Jose, February 1992.

18. P. J. Grother and G. T. Candela. Comparison of Handprinted Digit Classifiers. Technical Report NISTIR 5209, National Institute of Standards and Technology, June 1993.

19. K. Fukunaga. *Introduction to Statistical Pattern Recognition.* New York, Academic Press, second edition, 1990.

20. B. T. Smith, et al., Matrix Eigensystem Routines - EISPACK Guide, 2nd ed., Vol. 6 of *Lecture Noted in Computer Science*, Springer-Verlag, New York, 1976.

21. C. L. Wilson. Evaluation of Character Recognition Systems. In *Neural Networks for Signal Processing III*, pp. 485-496, IEEE, New York, September 1993.

22. J. L. Blue, G. T. Candela, P. J. Grother, R. Chellappa, and C. L. Wilson. Evaluation of Pattern Classifiers for Fingerprint and OCR Applications. *Pattern Recognition*, Vol. 27, No. 4, pp. 485-501, 1994.

23. For a bibliography and selected papers, see, for example, B. V. Dasarathy, “Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques,” IEEE Computer Society Press, 1991.

24. J. L. Bentley, “Multidimensional Binary Search Trees Used for Associative Searching,” Communications of the ACM, Vol. 18, pp. 509-517, 1975.

25. J. H. Friedman, J. L. Bentley, and R. A. Finkel, “An Algorithm for Finding Best Matches in Logarithmic Expected Time,” ACM Transactions on Mathematical Software, Vol. 3, pp. 209-226, 1977.

26. J. L. Blue, “Fast Nearest-Neighbor Searches in High Dimensional Spaces,” National Institute of Standards and Technology, to be published.
27. M. D. Garris and S. A. Janet. Scoring Package Release 1.0, Technical Report Special Software 1, **SP**, National Institute of Standards and Technology, October 1992.
28. M. D. Garris and S. A. Janet. NIST Scoring Package User’s Guide, Release 1.0. Technical Report NISTIR 4950, National Institute of Standards and Technology, October 1992.
29. M. D. Garris. Methods for Evaluating the Performance of Systems Intended to Recognize Characters from Image Data Scanned from Forms. Technical Report NISTIR 5129, National Institute of Standards and Technology, February 1993.
30. M. D. Garris, NIST Scoring Package Certification Procedures in Conjunction with NIST Special Databases 2 and 6. Technical Report NISTIR 5173, National Institute of Standards and Technology, April 1993
31. M. D. Garris. NIST Scoring Package Cross-Reference for use with NIST Internal Reports 4950 and 5129. Technical Report NISTIR 5249, National Institute of Standards and Technology, August 1993.

APPENDIX A. UTILITY *mis2evt*

The NIST standard reference recognition system uses the Karhunen Loeve (KL) transform to extract features for classifying segmented character images. This transform is obtained by projecting a character image onto eigenvectors of the covariance computed from a set of training images. The mathematical details of the KL transform are provided in Section 3.2.2.5.

The eigenvectors are computed off-line and stored in a basis function file (described in Section 3.2.1.1) because computing the eigenvectors of a large matrix is very expensive. The standard reference recognition system *hsfsys* reads the basis function file during its initialization, and then reuses the eigenvectors across all the character images segmented from fields of a specified type (digit, lower case, upper case, or Constitution box). The program *mis2evt* applies the KL transform to segmented character images and generates a basis function file. The program's main routine and FORTRAN subroutines are located in the distribution directory *src/bin/mis2evt*. The command line usage of *mis2evt* is as follows:

```
# mis2evt
Usage: mis2evt:
    -n   for 128x128 input write normed+sheared 32x32 intermediate misfiles
    -v   be verbose - notify completion of each image
    nrequiredevts evtfile mfs_of_misfiles
```

Arguments:

- The first argument *nrequiredevts* specifies the number of eigenvectors to be written to the output file. It is also the number of KL features that will ultimately be extracted from each binary image using the associated utility *mis2pat*. This integer determines the dimensionality of the feature vectors that are produced for classification. Its upper bound is the image dimensionality (which is 32x32 = 1024). Typically, this argument is specified to be much smaller than 1024 because the KL transform optimally compacts the representation of the image data into its first few coefficients (features). *Hsfsys* uses a value of 64. Reference 22 documents an investigation of the dependency of classification error on feature vector dimensionality.
- The second argument *evtfile* specifies the name of the output basis function file.
- The third argument *mfs_of_misfiles* specifies a text file that lists the names of all the MIS files containing images that will be used to calculate the covariance matrix. This argument is an MFS file with the first line containing an integer indicating the number of MIS files that follow. The remaining lines in the MFS file contain MIS file names, one name per line.

Options:

- The option "*-n*" specifies the storing of intermediate normalized character images. *Mis2evt* can process binary images that are either (128 by 128) or (32 by 32). In the case of the former, the program invokes a size normalization utility to produce 32 by 32 images and then applies a shear transformation to reduce slant variations. If the input images are already 32 by 32, this flag has no effect. If normalization does occur, the resulting normalized images are stored to MIS files having the same name as those listed in the MFS file, with the additional extension *32* appended. These intermediate files offer computational gains because usually the same images are used with *mis2pat*.
- The option "*-v*" produces messages to standard output signifying the completion of each MIS file and other computation steps.

This program is computationally expensive and may require as long as 60 minutes to compute the eigenvectors for a large set (50,000 characters) of images. The program *mis2evt* was used to generate the basis function files provided with this distribution in the top-level directory *weights* and ending with the extension *evt*. These files contain eigenvectors computed from the images provided in the top-level directory *train*. The MFS files used as arguments to *mis2evt* are also provided in *weights* and end with the extension *ml*. For example, the basis function file *td13_l.evt* was generated with the following command:

```
# mis2evt -v 64 td13_l.evt td13_l.ml
```

APPENDIX B. UTILITY *mis2pat*

A second supporting utility is provide with this distribution. *Mis2pat* takes a set of training images along with the eigenvectors generated by *mis2evt* and creates feature vectors that can be used as prototypes for training classifiers (in this case PNN). Typically, the same images used to compute the eigenvectors are used here to generate prototype vectors. The program *mis2pat* also builds a kd-tree as described in Section 3.2.2.6. The prototypes along with their class assignments and kd-tree are stored in a prototype file (described in Section 3.2.1.2). In addition, *mis2pat* computes median vectors from the prototype vectors and stores them in a median vector file. The program's main routine and FORTRAN subroutines are located in *src/bin/mis2pat*. The command line usage of *mis2pat* is as follows:

```
# mis2pat
Usage: mis2pat:
     -h   accept hexadecimal class files
     -n   with 128x128 images write normed+sheared 32x32 intermediate misfiles
     -v   be verbose - notify completion of each image
     classset evtfile outfile mfs_of_clsfiles mfs_of_misfiles
```

Arguments:

- The first argument *classset* specifies the name of a text file (MFS file) containing the labels assigned to each class. The integer on the first line of the file indicates the number of classes following, and the remaining lines contains one class label per line. For example, a digit classifier uses ten classes labeled 0 through 9.
- The second argument *evtfile* specifies the basis function file containing eigenvectors computed by *mis2evt*. The number of features in each output vector is determined by the number of eigenvectors in this file.
- The third argument *outfile* specifies the name of the output prototype file. The name of the output median vector file is the same except with a second extension *med* appended to the *outfile* argument.
- The final arguments are the names of text files (MFS files) that contain listings of file names. The argument *mfs_of_clsfiles* lists file names containing class assignments corresponding to the images in the MIS files listed in the argument *mfs_of_misfiles*. Each class assignment file must have the same number of class assignments as there are images in its corresponding MIS file, and the classes assigned must be consistent with those listed in the argument *classset*.

Options:

- The option “-h” specifies that the class labels listed in the *classset* file are to be converted to ASCII characters values represented in hexadecimal. All the class assignments in the files listed in the argument *mfs_of_clsfiles* use the convention where [30-39] represent digits, [41-5a] represent upper case, and [61-7a] represent lower case. If the *classset* file contains alphabetic representations such as [0-9], [A-Z], and [a-z], then this flag must be used to effect conversion of these labels to their hexadecimal equivalents.
- The option “-*n*” specifies the storing of intermediate normalized character images. *Mis2pat* can process binary images that are either (128 by 128) or (32 by 32). In the case of the former, the program invokes size and slant normalization utilities to produce 32 by 32 images. If the input images are already 32 by 32, this flag has no effect. If normalization does occur, the resulting normalized images are stored to MIS files having the same name as those listed in *mfs_of_misfiles*, with the extension *32* appended.
- The option “-*v*” produces messages to standard output signifying the completion of each MIS file.

This program was used to generate the prototype files provided with this distribution in the top-level directory *weights* and ending with the extension *pat*. These files contain KL feature vectors, their associated classes, and a kd-tree as described in Section 3.2.1.2. The feature vectors were computed using the eigenvectors found in the same directory and from the images provided in the top-level directory *train*. The MFS files used as arguments to *mis2pat* are also provided in *weights*, as are the *classset* files which end with the extension *set*. The class assignment files are listed in files ending with the extension *cl*, whereas the MIS files are listed in files ending with the extension *ml*. For example, the prototype file *td13_l.pat* was generated with the following command:

```
# mis2pat -vh l.set td13_l.evt td13_l.pat td13_l.cl td13_l.ml
# mv td13_l.pat.med td13_l.med
```

The second command renames the generated median vector file from its default name *td13_l.pat.med* to the distribution name *td13_l.med*.

APPENDIX C. 2nd Census OCR Systems Conference

In February of 1994, the Second Census Optical Character Recognition Systems Conference was sponsored by the Bureau of the Census and run by NIST. Ten different organizations submitted recognition results to NIST for scoring. The task of the conference was to read, via machine, a small handprinted portion of the 1990 Census Long Form. This part of the form contains three questions related to occupation. Three rectangular regions were provided on the form in which people were instructed to write their answers. Images were obtained from both microfilm and paper, and the images were cropped creating *miniforms* containing just the test questions. The details of the conference and the conclusions drawn from the results are presented in Reference 12. This is the most comprehensive test of recognition systems of this type done to date.

The NIST standard reference optical character recognition system is similar to the NIST system used in the conference. Both systems use connected components for segmentation; they use the same size and slant-normalization techniques; they both use the KL transform to compute feature vectors; and a PNN classifier is used in both systems.

Despite their similarities, there are some significant differences between the standard reference recognition system and the conference system. The standard reference recognition system conducts form removal prior to conducting field isolation, whereas the conference system simply registered the image, extracted the fields, and removed form artifacts. Systems that conducted some type of form removal in the conference achieved better results than those that did not. Also, the dictionary-based postprocessing used in the standard reference recognition system is substantially different than that used in the conference system. The standard reference recognition system uses the word-based process described in Section 3.5.1.3 to process the handprint written in the Constitution box of Handwriting Sample Forms (HSF forms). The conference system used phrase-based dictionary matching where the system corrected errors using lists of multiple-word phrases rather than using single-word dictionaries.

The application of these two systems is also significantly different. First, the image quality of the HSF forms distributed with SD1 and SD3 is better than the quality of images scanned from the Census Long Forms. Second, the standard reference recognition system uses a limited-size dictionary (38 words) when processing the Constitution box. This is in contrast to the conference where dictionaries of more than 60,000 multiple-word phrases were used. Although the dictionary is restricted when reading the Constitution box, the segmentation of the characters is complicated. The segmentation of this multiple-line text paragraph requires an elaborate solution, such as the sequence reconstruction described in Section 3.5.1.2. This is considerably more difficult than segmenting the single-line (occasionally more than one line) fields on the conference miniforms.

It is difficult to compare the performance of the standard reference recognition system to the conference system due to the differences between the two systems and their applications. However, some comparisons can be made at the word recognition level. The word accuracy of the standard reference recognition system was 61% on the Constitution box. The average field in the conference contained two words so that this level of accuracy, if sustained on the conference test, would have resulted in a 37% field accuracy rate. In the conference, NIST achieved a 25% field accuracy. Based on these numbers it is probable that the standard reference recognition system is better than the conference system.

The expected word accuracy for the best conference system was about 76%, so on a word basis we would expect the the standard reference recognition system to have about 15% (76%-61%) more errors than the best conference system. This is comparable to the median system reported at the conference. The difference between the best conference systems and the NIST standard reference recognition system is primarily due to segmentation. Unlike the best conference systems, the standard reference recognition system does not use any techniques for oversegmenting characters and reconstructing words.