1.0 INTRODUCTION

This document describes the NIST fingerprint database, *NIST Special Database 10*. The database provides a large sample of patterns for transitional fingerprint classes and classes with a low natural frequency of occurrence in *NIST Special Database 9*. The 552 fingerprint cards in *NIST Special Database 10* are non-mated cards archived on a set of three CD-ROM's with the first CD-ROM containing 2160 fingerprint images and the last two CD-ROMs containing 1680 fingerprints each. All fingerprints are stored in NIST's IHead raster data format and compressed using a non-standard implementation of the JPEG lossless [1] compression algorithm. The prints are 832 (w) X 768 (h) pixels (see Appendix A). Image data stored on the first CD-ROM requires approximately 690 megabytes of storage. The second and third CD-ROMs require approximately 590 megabytes of storage. The average compression ratio for all the images is 1.9 : 1.

The data was collected by selecting non-mated fingerprint cards, from the FBI's Technical Master File, which contained the most occurrences of the desired fingerprint patterns. Since the entire card was digitized, there is a mix of other classes within the specific class groupings. The data also includes a significant number of referenced fingerprints. Appendix C shows the exact distribution of the classes and referencing for each major class group that was collected.

The specific classes being collected were Tented Arch, Arch, Low Ridge Count Loops, Central Pocket Whorls, Double Loop Whorls, Plain Whorls and Accidental Whorls. The fingerprints are classified using the National Crime Information Center (NCIC) classes assigned by the FBI [2]. All classes and references are stored in the NIST IHead id field of each file.

2.0 NON-STANDARD IMPLEMENTATION OF JPEG LOSSLESS COMPRESSION

The compression used was developed from techniques outlined in the WG10 "JPEG" (draft) standard [1] for 8 bit grey scale images with modifications to the compressed data format. This is the same code used in *NIST Special Database 4 and 9*. The NIST IHead format already contained most of the information needed in the decompression algorithm, so the JPEG compressed data format was modified to contain only the information needed when reconstructing the Huffman code tables.
and identifying the type of predictor used in the coding process. Codes used to compress and decompress the images are still developed per the draft standard, but only applied to 8-bit gray scale images.

The standard uses a differential coding scheme and allows for seven possible ways of predicting a pixel value. Tests showed that predictor number 4 provided the best compression on the fingerprint images; therefore, this predictor was used to compress all of the images.

3.0 DATABASE REFLECTANCE CALIBRATION

The reflectance values for the fingerprint database, NIST Special Database 10, were calibrated using a reflection step table [3]. A plot of the reflectance values obtained using this step table is shown in Appendix B. Also shown on the plot and below is an equation used to predict the reflectance of a given datapoint. The plot in Appendix B shows that this predicted reflectance closely follows the actual reflectance obtained using the reflection step table.

\[
\text{predicted \% reflectance} = 5.1 + (.36 \times \text{grayscale pixel value})
\]

4.0 FINGERPRINT FILE FORMAT [4][5]

Image file formats and effective data compression and decompression are critical to the usefulness of image archives. Each fingerprint was digitized in 8-bit gray scale form at 19.6850 pixels/mm (500 pixels/inch), 2-dimensionally compressed using a modified JPEG lossless algorithm, and temporarily archived onto computer magnetic mass storage. Once all prints were digitized, the images were mastered and replicated onto ISO-9660 formatted CD-ROM discs for permanent archiving and distribution.

After digitization, certain attributes of an image are required to correctly interpret the 1-dimensional pixel data as a 2-dimensional image. Examples of such attributes are the pixel width and pixel height of the image. These attributes can be stored in a machine readable header prefixed to the raster bit stream. A program which manipulates the raster data of an image is able to first read the header and determine the proper interpretation of the data which follow it.

Numerous image formats exist, but most image formats are proprietary. Some are widely supported on small personal computers and others on larger workstations. A header format named IHead has been developed for use as a general purpose image interchange format. The IHead header is an open image format which can be universally implemented across heterogeneous computer architectures and environments. Both documentation and source code for the IHead format are publicly available and included with this database. IHead has been designed with an extensive set of attributes in order to adequately represent both binary and gray level images, to represent images captured from different scanners and cameras, and to satisfy the image requirements of diversified applications including, but not limited to, image archival/retrieval, character recognition, and fingerprint classification. Figure 1 illustrates the IHead format.
<table>
<thead>
<tr>
<th>Header Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII Format Image Header</td>
</tr>
<tr>
<td>8-bit Gray Scale Raster Stream</td>
</tr>
<tr>
<td>110101001101001111010010110...</td>
</tr>
</tbody>
</table>

- Representing the digital scan across the page left to right, top to bottom.
- 8 bits to a pixel
- 256 levels of gray
- 1 Pixel is packed into a single byte of memory.

Figure 1: An illustration of the IHead raster file format.

Since the header is represented by the ASCII character set, IHead has been successfully ported and tested on several systems including UNIX workstations and servers, DOS personal computers, and VMS mainframes. All attribute fields in the IHead structure are of fixed length with all multiple character fields null-terminated, allowing the fields to be loaded into main memory in two distinct ways. The IHead attribute fields can be parsed as individual characters and null-terminated strings, an input/output format common in the 'C' programming language, or the header can be read into main memory using record-oriented input/output. A fixed-length field containing the size in bytes of the header is prefixed to the front of an IHead image file as shown in Figure 1.
Figure 2: The IHead ‘C’ programming language structure definition.

The IHead structure definition written in the ‘C’ programming language is listed in Figure 2. Figure 3 lists the header values from an IHead file corresponding to the structure members listed in Figure 2. This header information belongs to the database file an000001.pct (see Figure A.1 in Appendix A). Referencing the structure members listed in Figure 2, the first attribute field of IHead is the identification field, id. This field uniquely identifies the image file, typically by a file name. The identification field in this example not only contains the image’s file name, but also the sex of the individual, if the image was scanned from an inked or live scan printed image, and the NCIC classification of the fingerprint, with any references to another classification (see Figure 8 and Section 5.4 for an example of class referencing). This convention enables an image recognition system’s hypothesized classification to be automatically scored against the actual classification.
The attribute field, created, is the date on which NIST received the digitized image. 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, 19.6850 pixels/mm (500 pixels/inch). The next two fields deal with compression.

In the IHead format, images may be compressed with virtually any algorithm. Whether the image is compressed or not, the IHead 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 complete field must be used in addition to the image’s pixel dimensions. For example, the images in this database have a compression code of 6 signifying that modified JPEG lossless 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 complete gives the size of the compressed block of image data.
Once the compressed image data has been loaded into memory, JPEG lossless decompression can be used to produce an image which has the pixel dimensions consistent with those stored in its header. Using JPEG lossless compression and this compression scheme on the images in this database, an average compression ratio of 1.9 to 1 was achieved.

The attribute field, align, stores the alignment boundary to which scan lines of pixels are padded. Pixel values of 8-bit gray scale images are stored 1 byte (or 8 bits) to a pixel, so the images will automatically align to an even byte boundary.

The next three attribute fields identify data interchanging issues among heterogeneous computer architectures and displays. The unitsize field specifies how many contiguous bits are bundled into a single unit by the digitizer. The s8bit 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, data incompatibilities across computer hardware and data 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 gray scale image described in Figure 3 is gray print on a white background and the value of the white pixel is 255. This field is particularly useful to image display routines. The signed 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 pixel 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 images used in this database were renamed from their original filenames, given by the FBI, and the 'link' to the original filename was stored in the parent field as well as the size of the ten print image (columns x rows) before the individual fingerprints were segmented. The FBI filename consists of three fields separated by periods. The first field contains a tape number (i.e. tape501, tape502, ..., tape523), indicating the FBI streamer tape the file was stored on. The second field contains 5 characters. The first two characters in the second field indicate the specific fingerprint class group being collected. The next three characters are the card sequence number for that class grouping. The last field indicates the finger number of the file (01-10).

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.
5.0 DATABASE CONTENT AND ORGANIZATION

NIST Special Database 10 contains 5520 8-bit gray scale fingerprint images which are distributed on three ISO-9660 formatted CD-ROMs and compressed using a non-standard implementation of the JPEG lossless compression algorithm [1]. Included with the fingerprint data are software and documentation.

![Diagram of NIST Special Database 10 directory structure]

Figure 4: Top level directory tree for each CD-ROM in NIST Special Database 10.

5.1 Database File Hierarchy

The top level of the file structure contains three directories src, man, and data (see Figure 4). The code needed to decompress and use the image data is contained in the src directory with man pages for the source code stored in the man directories. The data directory contains the fingerprint images stored in two levels of subdirectories for easier access and clarity (see Figure 5). The first level of subdirectories indicates the specific class collected within that group of data. The specific class groupings on each CD-ROM are: disc 1 -> arches (aa), central pocket whorls (cw) and double loop whorls (dw), disc2 -> low ridge count loops (sl) and plain whorls (pw), and disc 3 -> tented arches (tt) and accidental whorls (xw). The next level has a subdirectory for each fingerprint card which contains the ten segmented fingerprint images for each card. The aa, sl and tt class groups have 120 cards each (card_001 - card_120) and the other class groups have 48 cards each.

Fingerprints are stored with filenames containing two letter, six digits and a "pct" (picture) extension. The first two characters in the filename are the same as the specific class being collected (aa, tt, sl, cw, dw, pw or xw). The next six characters indicate the sequence number (000001 - 001200 for aa, sl and tt groups). The finger number is given by the last of the six digits, with zero representing digit ten on the fingerprint card (see Figure 6 for fingerprint card layout). Every ten prints in sequential order are a group of prints from the same card (i.e. digits 1-10).
5.2 Low Count Loops

The main low count loops collected were ulnar loops (see statistics in Appendix C). The user can create low count radial loop data by flipping the ulnar loop image about a vertical axis centered on the image. This allows for more samples of both low count loop classes.

5.3 NCIC Classifications

The classes stored in the NIST Ihead ld field are the NCIC classes [2], including any references, that were assigned by the FBI Identification Division Automated System (IDAS). A listing of the
possible class codes is given below. Note that the classification ac (approximate class) means that the classification immediately after the ac is the best classification that can be assigned to the print given the information shown in the image.

Classification name (class code)
-------------------------------
Arch (aa)
Tented Arch (tt)
Ulnar Loop Ridge Counts (01 - 49)
Radial Loop Ridge Counts (51 - 99)
Plain Whorl Inner Ridge Tracing (pi)
Plain Whorl Outer Ridge Tracing (po)
Plain Whorl Meeting Ridge Tracing (pm)
Central Pocket Whorl Inner Ridge Tracing (ci)
Central Pocket Whorl Outer Ridge Tracing (co)
Central Pocket Whorl Meeting Ridge Tracing (cm)
Double Loop Whorl Inner Ridge Tracing (di)
Double Loop Whorl Outer Ridge Tracing (do)
Double Loop Whorl Meeting Ridge Tracing (dm)
Accidental Whorl Inner Ridge Tracing (xi)
Accidental Whorl Outer Ridge Tracing (xo)
Accidental Whorl Meeting Ridge Tracing (xm)
Approximate Class (ac) followed by a valid class
Amputation or Missing (xx)
Scar or Mutilation (sr)

Figure 7: Classification codes for NIST Special Database 10.

5.4 Class Referencing

The referencing of a fingerprint is caused by a variety of ambiguities such as a scar occurring in the fingerprint, the quality of the print rolling, or the print having ridge structures characteristic of two different classes. The referenced prints could easily cause a wrong classification when used in testing an automated classification system but could provide a challenge in the later stages of development. The id field of the prints in NIST Special Database 10 contain the primary fingerprint class followed by any references. An example IHead header is shown in Figure 8 (aa000042.pct) and the corresponding print is shown in Figure A.2. The fingerprint is classified as an aa and referenced to a tt.
Figure 8: The IHead values for the fingerprint data file aa000042.pct.

5.5 Segmenting the Fingerprint Images

The individual fingerprint images were obtained by scanning all ten prints on a card into one large image (4096 X 1536 pixels) and then segmenting each individual image from that larger image. The individual images were segmented at the same exact points on all the larger (full card) images. The image size of 832 X 768 pixels was selected to allow the user the capability of reconstructing the fingerprint card image and resegmenting the individual fingerprint images if desired. The images overlap by 32 pixels with horizontally adjacent images and by 18 pixels with vertically adjacent images which must be accounted for when reconstructing the fingerprint card image.

5.6 Inked and Live Scan Printed

The fingerprints were scanned from two types of prints. The first type were fingerprint patterns created by rolling the individuals ink covered finger on the fingerprint card (denoted by an i in the id field). The second type were patterns taken with a live scanning device and then printed onto a fingerprint card (denoted by an l in the id field). This is important in that the quality of the image is affected by the resolution of the printer used to print the live scanned image onto the fingerprint card.
6.0 SOFTWARE FOR ACCESSING DATABASE

Included with the fingerprint images are documentation and software written in the 'C' programming language. The software was developed on a SUN sparc station and has only been tested on that platform. Four programs are included in the src directory: dumpihdr, ihdr2sun, sunalign, and dcpljjpg. These routines are provided as an example to software developers of how IHead images can be manipulated and used. Descriptions of these programs and their subroutines are given below as well as in the included man pages located in the man directory. Copies of the manual pages are also included in Appendix D.

6.1 Compilation

The CD-ROMs used for NIST Special Database 10 are read only storage medium. The files in the src directory must be copied to a read-writable partition prior to compiling. After copying these files, executable binaries can be produced by invoking the UNIX utility make to execute the included makefile. An example of this command follows.

    # make -f makefile.mak

6.2 Dumpihdr <Ihead file>

Dumpihdr is a program which reads an image's IHead data from the given file and formats the header data into a report which is printed to standard output. The report shown in Figure 3 was generated using this utility. The main routine for dumpihdr is found in the file dumpihdr.c and calls the external function readihdr().

readihdr() is a function responsible for loading an image's IHead data from a file into main memory. This routine allocates, reads, and returns the header information from an open image file in an initialized IHead structure. This function is found in the file ihead.c. The IHead structure definition is listed in Figure 2 and is found in the file ihead.h

6.3 Ihdr2sun <Ihead file>

Ihdr2sun converts an image from NIST IHead format to Sun rastfile format. Ihdr2sun loads an IHead formatted image from a file into main memory and writes the raster data to a new file appending the data to a Sun rastfile header. The main routine for this program is found in the file ihdr2sun.c and calls the external function ReadIheadRaster() which is found in the file rastério.c.

ReadIheadRaster() is the procedure responsible for loading an IHead image from a file into main memory. This routine reads the image's header data returning an initialized IHead structure by calling readihdr(). In addition, the image's raster data is returned to the caller uncompressed. The images in this database have been 2-dimensionally compressed using a modified JPEG lossless compression algorithm, therefore ReadIheadRaster() invokes the external procedure jpegIddc() which is responsible for decompressing the raster data. Upon completion, ReadIheadRaster()
returns an initialized IHead structure, the uncompressed raster data, the image’s width and height in pixels, and pixel depth.

Jpg1Idcp() accepts image raster data compressed using the modified JPEG lossless compression algorithm and returns the uncompressed image raster data. Jpg1Idcp() was developed using techniques described in the WG10 “JPEG” (draft) standard [1] and adapted for use with this database. Source code for the algorithm is found in jpg1Idcp.c.

6.4 Dcpljjpg <lossless JPEG compressed file>

Dcpljjpg is a program which decompresses a fingerprint image file (approximately 10 seconds per image, for images from this database, on a scientific workstation) that was compressed using the modified JPEG compression routine. The routine accepts a compressed image in NIST IHead format and writes the uncompressed image to the same filename using the NIST IHead format. The main routine is found in dcpljjpg.c and calls the external functions ReadIheadRaster() (see section 6.3 for ReadIheadRaster description) and writeihdrfile().

Writeihdrfile() is a routine that writes an IHead image into a file. This routine opens the passed filename and writes the given IHead structure and corresponding data to the file. Writeihdrfile() is found in the src file rasterio.c.
References


Appendix A: Database Fingerprint Image Samples
Figure A.1: Fingerprint file aa000001.pct from NIST Special Database 10.
Figure A.2: Fingerprint file aa000042.pct from NIST Special Database 10.
Appendix B: Database Reflectance and Resolution Calibration
FINGERPRINT DATABASE REFLECTANCE VALUES

GRAYSCALE PIXEL VALUES (0 = BLACK, 255 = WHITE)
KEY: SOLID: SCANNED  DASHED: PREDICTED
PREDICTED % REFLECTANCE = -5.1 + (0.36 * GRAYSCALE PIXEL VALUE)
Database Resolution Calibration

The resolution of the scanner used to create the database was calibrated using a NBS 1010A resolution chart\(^1\). A portion of this image is shown below (Figure B.2) after being magnified for viewing (Note: Printing has significantly reduced the quality of this image.). The scan of the table showed that the vertical resolution was approximately 10.5 line pairs/mm and the horizontal resolution was approximately 11.0 line pairs/mm.

![Resolution Calibration Table](image)

**Figure B.2: Scan of Resolution Calibration Table.**

---

Appendix C: Fingerprint Class Distribution Statistics
Class Distribution

The data in Figure C.1 shows the exact class distribution of NIST Special Database 10. Since the data was collected by storing all ten fingerprints from a card some of the specific class groupings contain numerous fingerprints from other classes.

Note:

| ?L (1-5 ridge count loops) | All whorl ridge tracings (L,M,O) for a whorl class are counted together (i.e. PL,PM,PO are counted in PW). |
| ?M (6-30 ridge count loops) | |
| ?H (>30 ridge count loops) | |

### Specific Class Groupings

<table>
<thead>
<tr>
<th>Actual Classes Collected</th>
<th>aa</th>
<th>sl</th>
<th>tt</th>
<th>cw</th>
<th>dw</th>
<th>pw</th>
<th>xw</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>1199</td>
<td>88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1287</td>
</tr>
<tr>
<td>UL</td>
<td>843</td>
<td>149</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>994</td>
</tr>
<tr>
<td>UM</td>
<td>164</td>
<td>86</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>321</td>
</tr>
<tr>
<td>RL</td>
<td>189</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>212</td>
</tr>
<tr>
<td>RM</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>TT</td>
<td>842</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>842</td>
</tr>
<tr>
<td>PW</td>
<td></td>
<td>153</td>
<td>138</td>
<td>228</td>
<td>250</td>
<td></td>
<td></td>
<td>769</td>
</tr>
<tr>
<td>CW</td>
<td></td>
<td>308</td>
<td></td>
<td>94</td>
<td>71</td>
<td></td>
<td></td>
<td>409</td>
</tr>
<tr>
<td>DW</td>
<td></td>
<td>19</td>
<td>342</td>
<td>79</td>
<td>124</td>
<td></td>
<td></td>
<td>564</td>
</tr>
<tr>
<td>XW</td>
<td></td>
<td>6</td>
<td></td>
<td>99</td>
<td>105</td>
<td></td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>SR</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>1200</td>
<td>1200</td>
<td>1200</td>
<td>480</td>
<td>480</td>
<td>480</td>
<td>480</td>
<td>3520</td>
</tr>
</tbody>
</table>

Figure C.1: Class distribution statistics for NIST Special Database 10.
The data in Figure C.2 shows the class referencing for *NIST Special Database 10*. The referencing statistics are given for each specific class grouping. All primary classes without references (UL, UM, ...) show the number of unreferenced prints for that particular class. See the note on page 21 for an explanation of the L, M, and H as used with the ulnar and radial loops. If a primary class has more than one reference the ordering is insignificant, meaning UL/AA/TT is the same as UL/TT/AA and both would be counted as UL/AA/TT.

**Figure C.2: Class referencing statistics for NIST Special Database 10.**

<table>
<thead>
<tr>
<th>Class/Reference</th>
<th>aa</th>
<th>sl</th>
<th>tt</th>
<th>cw</th>
<th>dw</th>
<th>pw</th>
<th>xw</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL</td>
<td>812</td>
<td>178</td>
<td></td>
<td></td>
<td></td>
<td>69</td>
<td></td>
<td>1059</td>
</tr>
<tr>
<td>UL/UM</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>UL/TT</td>
<td>184</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>232</td>
</tr>
<tr>
<td>UL/UM/TT</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>UL/PW/DW</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>UL/CW/TT</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>UL/AA/TT</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>UM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UM/UL</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>UM/CW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>UM/TT</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>UM/UL/TT</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>UM/CW/DW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RL</td>
<td>115</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>133</td>
</tr>
<tr>
<td>RL/TT</td>
<td>70</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>RL/RM/TT</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RM/RH</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RM/CW</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RM/TT</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>RM/PW/CW</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RM/DW/XW</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PW/CW</td>
<td>145</td>
<td>120</td>
<td>198</td>
<td>220</td>
<td>683</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PW/DW</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>21</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22
### Specific Class Groupings

<table>
<thead>
<tr>
<th>Class/Reference</th>
<th>aa</th>
<th>sl</th>
<th>tt</th>
<th>cw</th>
<th>dw</th>
<th>pw</th>
<th>xw</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CW</td>
<td>216</td>
<td>81</td>
<td>1</td>
<td>1</td>
<td>79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/UM</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/RM</td>
<td>6</td>
<td>1</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/PW</td>
<td>70</td>
<td>8</td>
<td>1</td>
<td>79</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/DW</td>
<td>2</td>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/PW/DW</td>
<td>1</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/PW/XW</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CW/DW/XW</td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW</td>
<td>11</td>
<td>268</td>
<td>67</td>
<td>105</td>
<td>451</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW/UM</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW/PW</td>
<td>1</td>
<td>65</td>
<td>10</td>
<td>16</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW/CW</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW/XW</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW/PW/CW</td>
<td>2</td>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XW</td>
<td></td>
<td></td>
<td>5</td>
<td>77</td>
<td>82</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XW/RM</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XW/PW</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XW/CW</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XW/DW</td>
<td></td>
<td></td>
<td>1</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XW/PW/DW</td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AA</td>
<td>1160</td>
<td>68</td>
<td></td>
<td></td>
<td>1228</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AA/TT</td>
<td>32</td>
<td>20</td>
<td></td>
<td></td>
<td>52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AA/UL/TT</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT</td>
<td></td>
<td></td>
<td>341</td>
<td></td>
<td>341</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/UL</td>
<td></td>
<td></td>
<td>310</td>
<td></td>
<td>310</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/UM</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/RL</td>
<td></td>
<td></td>
<td>53</td>
<td></td>
<td>55</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/RM</td>
<td></td>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/CW</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/AA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/UL/UM</td>
<td></td>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/UL/RL</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/UL/AA</td>
<td></td>
<td>22</td>
<td></td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/RL/RM</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT/RL/AA</td>
<td></td>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total scars</td>
<td>7</td>
<td>15</td>
<td>10</td>
<td>1</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix D: Manual Pages for Database Source Code
NAME
dcplljpg - non-standard JPEG lossless decompression for
8 bit gray scale images

SYNOPSIS
dcplljpg ihdrfile

DESCRIPTION
Dcplljpg takes an 8 bit gray scale ihdr image, which was
compressed using jpegcomp4, and decompresses it using tech-
niques from the committee draft ISO/IEC CD 10196-1 for
"Digital Compression and Coding of Continuous-tone Still
images" with modifications to the draft image header.

NOTE: dcplljpg does not allow more than 8 bits/pixel input
precision.

OPTIONS

ihdrfile
Any 8 bit gray scale ihdr raster image (previously
compressed using jpegcomp4).

EXAMPLES
dcplljpg foo.pct

FILES

ihead.h NIST's raster header include file
jpeg.h Include file for jpeg algorithm

SEE ALSO
dumpihdr(1), ihdr2sun(1), ReadIheadRaster(3),
writeihdrfile(3)

DIAGNOSTICS
dcplljpg exits with a status of -1 if an error occurs.

BUGS
dcplljpg only handles gray scale images up to 8 bits per
pixel precision.
NAME
dumpihdr - takes a NIST IHead image file and prints its header content to stdout

SYNOPSIS
dumpihdr ihdrfile

DESCRIPTION
Dumpihdr opens a NIST IHead rasterfile and formats and prints its header contents to stdout.

OPTIONS
ihdrfile
 any NIST IHead image file name

EXAMPLES
dumpihdr foo.pct

FILES
ihead.h NIST’s raster header include file

SEE ALSO
ihdr2sun(1), ReadIheadRaster(3), writeihdrfile(3), writeihdr(3), readihdr(3), printihdr(3)

DIAGNOSTICS
Dumpihdr exits with a status of -1 if opening ihdrfile fails.

BUGS
ihdr2sun - takes a NIST ihdr image and converts it to a Sun rasterfile

SYNOPSIS

ihdr2sun [ -o outfile ] ihdrfile [ mapfile ]

DESCRIPTION

Ihdr2sun converts a NIST ihdr rasterfile to a Sun rasterfile. If the optional argument mapfile is included on the command line and the input image is multiple bitplane, the colormap in mapfile will be inserted into the Sun rasterfile, otherwise a default colormap gray.map will be used when necessary. The Sun image file created will have the root name of ihdrfile with the extension .ras appended, unless an alternate outfile is specified.

OPTIONS

    ihdrfile
      any ihdr raster image

    mapfile
      optional colormap file

EXAMPLES

ihdr2sun foo.pct gray.map

FILES

/usr/include/rasterfile.h
  sun’s raster header include file

ihead.h
  NIST’s raster header include file

SEE ALSO

dumpihdr(1), sunalign(1), rasterfile(5)

DIAGNOSTICS

Ihdr2sun exits with a status of -1 if opening ihdrfile fails.

BUGS

Ihdr2sun does not currently support multiple bit levels per pixel other than depth 8.
NAME
sunalign - takes a sun rasterfile and word aligns its scan-lines

SYNOPSIS
sunalign sunrasterfile

DESCRIPTION
Sunalign takes the file sunrasterfile and determines if the stored scan lines in the file require word alignment. If so, the command overwrites the image data making scan lines word aligned. This command is useful when taking clipped images from the HP Scan Jet and importing them into Frame Maker.

OPTIONS
sunrasterfile
any sun rasterfile image

EXAMPLES
sunalign foo.ras

FILES
/usr/include/rasterfile.h
sun’s raster header include file

SEE ALSO
rasterfile(5)

DIAGNOSTICS
Sunalign exits with a status of -1 if opening sunrasterfile fails.

BUGS
NAME
jplldcpl - takes a JPEG lossless compressed input data
buffer (with modified data header) and writes the
uncompressed data to the passed output buffer

SYNOPSIS
void jplldcpl(indata, width, height, depth, outbuffer)
unsigned char *indata, *outbuffer;
int width, height, depth;

DESCRIPTION
jplldcpl() takes the input buffer indata and decompresses it
writing the uncompressed data into the output buffer out-
buffer with length equal to the original image dimensions
given. This procedure was developed using techniques from
the committe draft ISO/IEC CD 10198-1 for "Digital Compre-
sion and Coding of Continuous-tone Still Images" with modifi-
cations to the draft image header. The source is found in
the source code file jplldcpl.c.

indata
- the compressed data input buffer

width
- the pixel width of the image from which the input
data came

height
- the pixel height of the image from which the input
data came

depth
- the pixel depth of the image from which the input
data came

outbuffer
- the output buffer in which the uncompressed data is
to be returned

SEE ALSO
dcp11jpg(1), ReadlheadRaster(3), writeihdrfile(3)

BUGS
NOTE: jplldcpl will only work with gray-scale images that
were compressed using a modified data header (not the stan-
dard lossless JPEG data header).

Sun Release 4.1  Last change: 14 January 1992
NAME
printihdr - prints an ihead structure to the passed file pointer

SYNOPSIS
#include <ihead.h>

printihdr(head, fp)
IHEAD *head;
FILE *fp;

DESCRIPTION
Printihdr() takes a pointer to an ihead structure and prints
the ihead structure to the file pointed to by fp. The source
is found in the source code file ihead.c.

fp - an open file pointer

head - a pointer to an initialized ihead structure

SEE ALSO
writeihdrfile(3), writeihdr(3), readihdr(3), ReadIheadRaster(3), dumpihdr(1)

BUGS
NAME
readihdr - allocates and reads header information into an ihead structure and returns the initialized structure

SYNOPSIS
#include <ihead.h>

readihdr(fp)
FILE *fp;

DESCRIPTION
readihdr() takes a file pointer to an ihead structured file. Then allocates and reads the header information from the file into an ihead structure. The source is found in the source code file ihead.c.

fp - an open file pointer

SEE ALSO
ReadIheadRaster(3), writeihdrfile(3), printihdr(3), writeihdr(3), dumpihdr(1)

BUGS
NAME

ReadlheadRaster - loads into memory an ihead structure and corresponding image data from a file

SYNOPSIS

#include <ihead.h>

ReadlheadRaster(file, head, data, width, height, depth)
char *file;
IHEAD **head;
unsigned char **data;
int *width, *height, *depth;

DESCRIPTION

readlheadRaster() opens a file named file and allocates and loads into memory an ihead structure and its corresponding raster image data. If the image data is compressed, ReadlheadRaster will uncompress the data before returning the data buffer. This routine also returns several integers converted from their corresponding ASCII entries found in the header. The source is found in the source code file rasterio.c.

file - the name of the file to be read from

head - a pointer to where an ihead structure is to be allocated and loaded

data - a pointer to where the array of binary raster image data is to be allocated and loaded

width - integer pointer containing the image's pixel width upon return

height - integer pointer containing the image's pixel height upon return

depth - integer pointer containing the image's Bits Per Pixel upon return

SEE ALSO

printihdr(3), readihdr(3), writeihdrfile(3), writeihdr(3), dumpihdr(1)

DIAGNOSTICS

ReadlheadRaster() exits with -1 when opening file fails.

BUGS

Sun Release 4.1 Last change: 05 March 1990 1
NAME
writeihdrfile - writes an ihdr structure and corresponding image data to a file

SYNOPSIS
#include <ihdr.h>

writeihdrfile(file, head, data)
char *file;
IHEAD *head;
unsigned char *data;

DESCRIPTION
Writeihdrfile() opens a file name file and writes an ihdr structure and its corresponding image data to it. The source is found in the source code file rasterio.c.

file - the name of the file to be created
head - a pointer to an initialized ihdr structure
data - the array of raster image data

SEE ALSO
writeihdr(3), printihdr(3), ReadIheadRaster(3), readihdr(3), dumpihdr(1)

DIAGNOSTICS
Writeihdrfile() exits with -1 when opening file fails.

BUGS
NAME
  writeihdr - writes an ihead structure to an open file

SYNOPSIS
  #include <ihead.h>

  writeihdr(fp, ihead)
  FILE *fp;
  IHEAD *ihead;

DESCRIPTION
  Writeihdr() takes a pointer to an ihead structure and writes
  it to the open file pointed to by fp. The source is found in
  the source code file ihead.c.

  fp    - an open file pointer
  ihead - a pointer to an initialized ihead structure

SEE ALSO
  writeihdrfile(3), printihdr(3), readihdr(3), ReadIheadRas-
  ter(3), dumpihdr(1)

BUGS