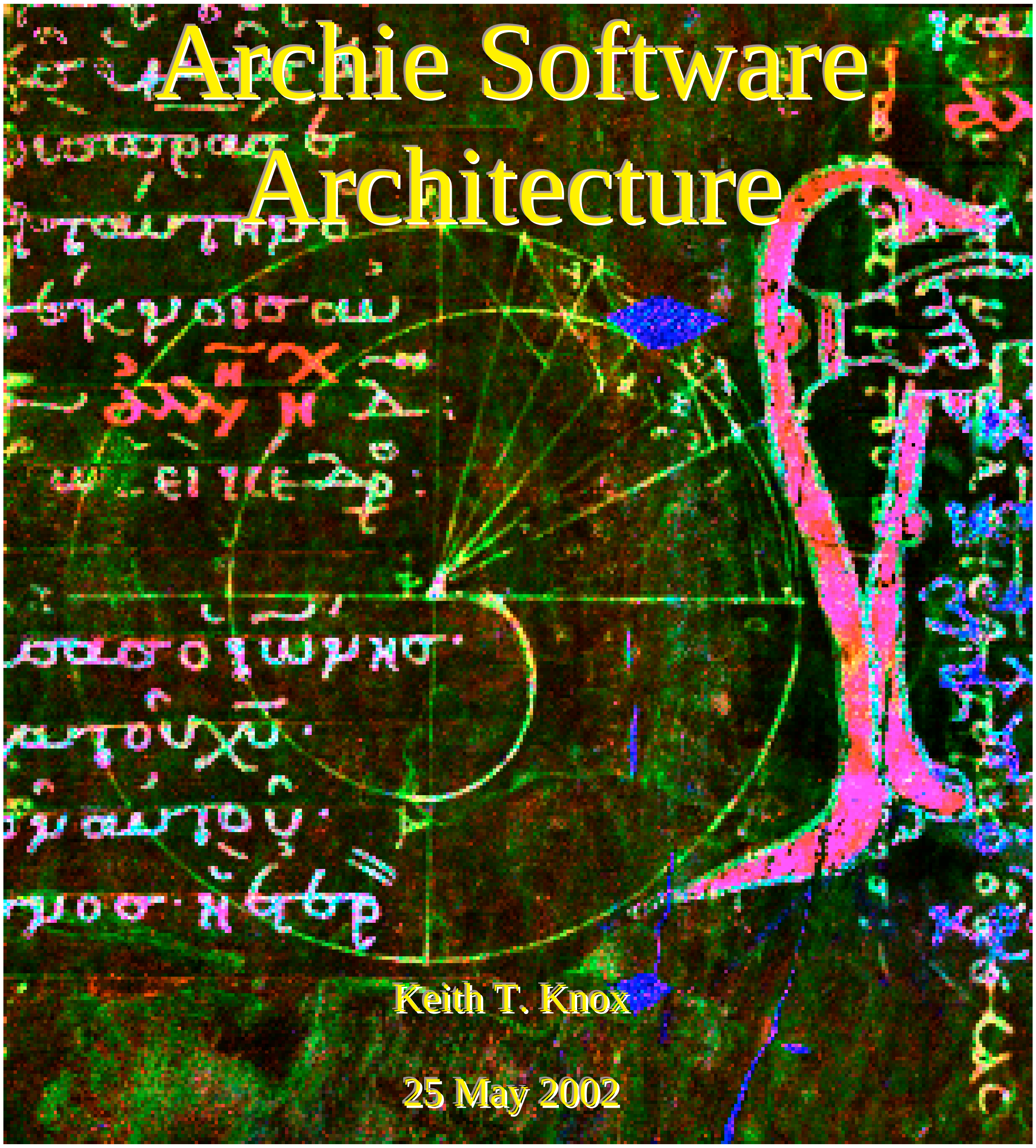
Archie Software Architecture



Keith T. Knox

25 May 2002

1.0 Overview

This report describes Archie, an image processing software package written at RIT for the Archimedes Palimpsest project. The software is written in the C programming language and is designed to run on the UNIX operating system.

The architecture of the package allows it to be used for general purpose image processing tasks. Individual algorithms can be easily implemented and included in the package, with no modifications required to the rest of the package. This enables quick experimentation and testing of new image processing algorithms.

The software package has no interactive graphic user interface. Tasks are intended to be run in batch mode and to be applied to a large number of images, so only a command line interface is provided. A complete history of the commands used to process an image are stored within the image file header for later review.

Instructions for compiling the package and for constructing new modules are included in sections 6 and 7.

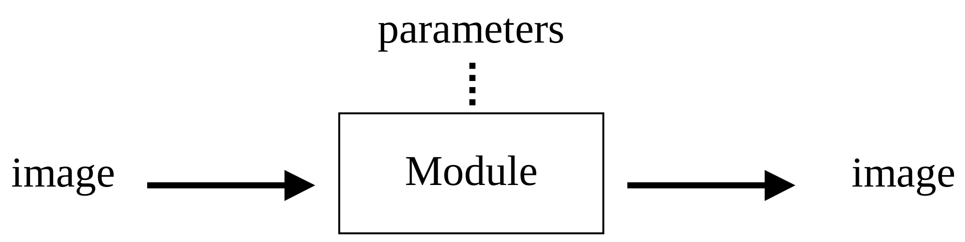
2.0 Archie Software Architecture

2.1 Modularity

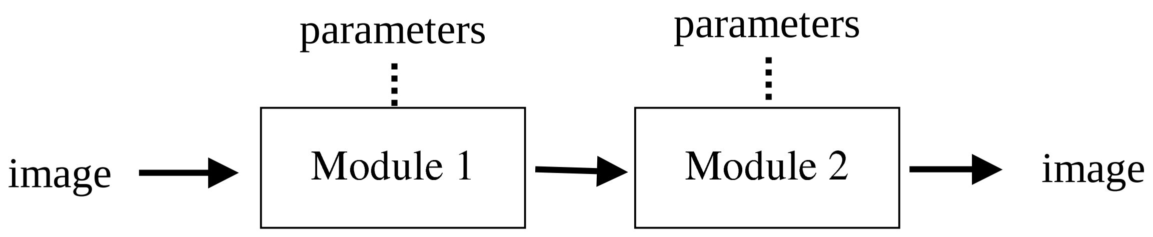
Individual algorithms in Archie are implemented in separately compiled modules that implement one particular algorithm. Typically, each module takes an input image on the standard input pipe (standard in), processes the image and writes the image to the standard output pipe (standard out). The image is stored in a simple form of the TIFF 6.0 image file format.

The details of what the input image looks like are stored the TIFF header of the image file. This header includes information such as the number of pixels, the number of scanlines, the number of bits/sample, the number of samples/pixel, etc. It also includes other TIFF tags that can be used to store metadata related to the image, see section 6.4.

Each module can be called with options on the command line. These options specify parameters important to the specific algorithm, such as window sizes, types of processing, scale factors, etc.



Just like puzzle pieces, individual modules can be joined together by piping the output of one module to the input of another. If a module changes the size of the image, then its output image header is changed to reflect the new size. The next module down the pipeline reads the changed size from the new header.



For example, if Module 1 above is the command "scale –factor 2", then it will magnify the image by replication by a factor of 2. An input image of 512 x 512 pixels will be converted to an output image of 1024 x 1024 pixels. Module 2 will only see the larger 1024 x 1024 image.

2.2 Image Streaming

Each module processes the data that it receives in a prescribed order. First, it reads the parameters from the command line. These parameters tell the module what the user would like done to the input image.

Next, it reads and processes the header of the input TIFF image. From the header it learns the characteristics of the input image. The header and the command line parameters are then compared and the module decides if it can complete that task as requested. If it cannot, it issues an error and halts processing.

After deciding that it can do the task, the module creates a new TIFF image header, one that describes the characteristics of its output image. This new header is then written to the output and is passed on to the next module in line that is waiting to hear what to expect.

Only after having informed the next module what to expect, does this module start to process the image. In this software package, the preferred mode of processing the image is a few scanlines at a time, maybe even one at a time. This mode is called streaming.

In streaming, a scanline is read from the input image, processed to produce a corresponding output scanline, which is then written to the output pipe. In this way, the image is streamed through the many modules in the pipeline a few scanlines at a time.

If the processing algorithm requires a window or neighborhood in which to do the processing, then a few input scanlines can be buffered and cycled through. Of course, in a few cases of algorithms like the FFT, the whole input image may need to be buffered.

2.3 Advantages of the Architecture

There are several advantages of this architecture. It makes it easy to use multiple processors, it is memory efficient (enabling larger images to be processed) and it allows errors to be dealt with before any processing is started.

Multiprocessor UNIX machines are becoming commonplace. These computers automatically divide tasks between available processors. In this architecture, the image processing task is specified by the user as computed by different modules that can be run on different processors. As long as the individual modules are written with a good balance of I/O and computation, a multiprocessor operating system can utilize all of its available processors on any given image processing task.

Memory is inexpensive these days and getting cheaper. One might argue that since memory is so cheap, it is not necessary to write code that is memory efficient. The counter argument, however, is that everything else being equal, a memory-efficient module, that streams an image through, can process a much larger image more efficiently that can a module that reads the whole image into memory before processing. Memory will continue to get cheaper, but images will continue to get larger.

This architecture also allows errors to be dealt with before any processing is started. Errors are generated as a result of inconsistencies between the input image and the parameters passed to the module. By processing the image headers first and passing them down the pipeline of modules, every module has a chance to inspect what it is about to do and to halt the process before any processing is started. The user of the package would not be happy if the first 5 modules spent an hour completely processing their version of the image and then the last module quit without doing anything because its parameters had a small typographical error. Handling the headers first (and therefore the errors) gets them out of the way before the processing is begun.

3.0 Image File Format

Each module in the architecture needs to understand the same image file format, so that the modules can be combined in any order and they will all communicate with each other.

There are several desirable characteristics of the image file format for Archie. It should:

- allow a broad range of images types
- be readable by other software, such as Photoshop
- lend itself to a streaming architecture
- be a recognized external standard
- provide the ability to record metadata within the file.

After looking over several image file format standards, it was concluded that a restricted version of TIFF 6.0 would meet all of the above requirements.

The details of the file format are presented here, but they do not need to be understood to read and write the image files within an individual module's source code. Special library routines are provided that handle reading and writing both the TIFF headers and the raster data. The only part of the structure that needs to be understood by someone writing code is how the raster data is stored within the scanlines in memory. The library routines and the structure of the scanlines are discussed in detail in sections 6.3 and 7.2.

3.1 Image File Restrictions

The TIFF 6.0 format is very broad, but a restricted version is implemented in the Archie library routines. The restricted version allows the image file to be streamed through a pipeline of modules and still be readable by Photoshop. Here is the list of restrictions:

Byte Order. Either Intel (IBM) or Motorola (Mac) byte order is allowed. Typically, the byte order of the input is used on the output, but it can be changed within a module.

Bits/Sample. Currently restricted to 1, 8, or 16 for unsigned integers and to 32 for floating point, see Sample Format. Complex floating point 64 is planned for the future.

Sample/Pixel. There can be any number of samples/pixel, only subject to photometric interpretation or a soft matte. All samples must have the same number of bits/sample.

Sample Format. Pixel data is interpreted as unsigned integer, signed integer, floating point, or undefined. The latter may be used for complex floating point, in the future.

Soft Matte. An extra sample, attached to each image pixel, that multiplies the pixel to mask the image. It must have the same number of bits/sample as the image.

Photometric Interpretation. There are two photometries defined, both where larger numbers are brighter. Gray has one monochrome sample and RGB has three.

Sub Images. A TIFF file can contain more than one image. Archie only reads the first, or high resolution, image if more than one is present. See the module readtif.

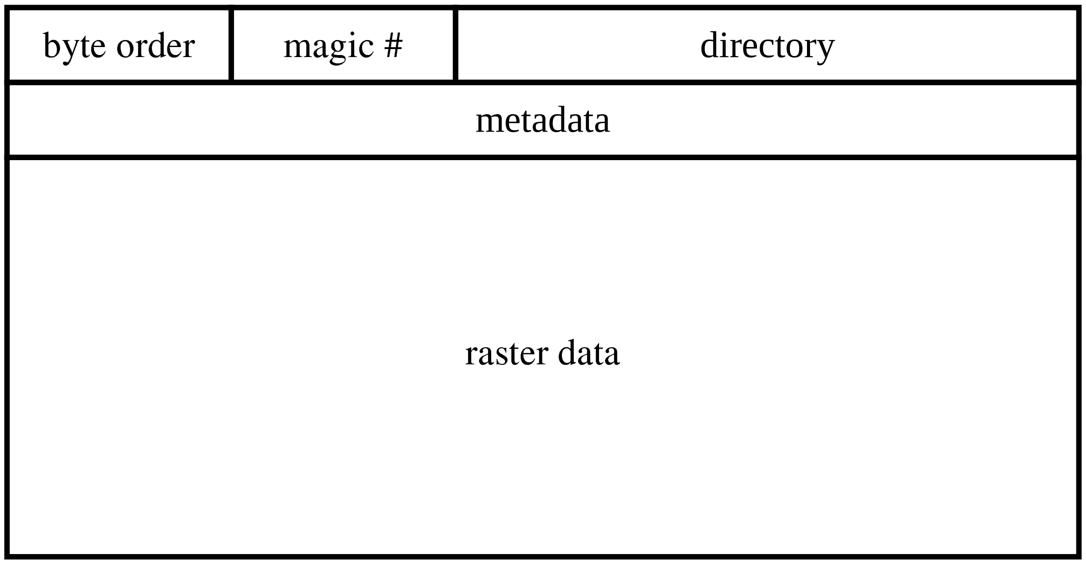
TIFF Directory. In the Archie image file, the directory, any directory entries and any metadata must come before all of the raster image data.

Image Strips. There can only be one image strip, which contains all the image data.

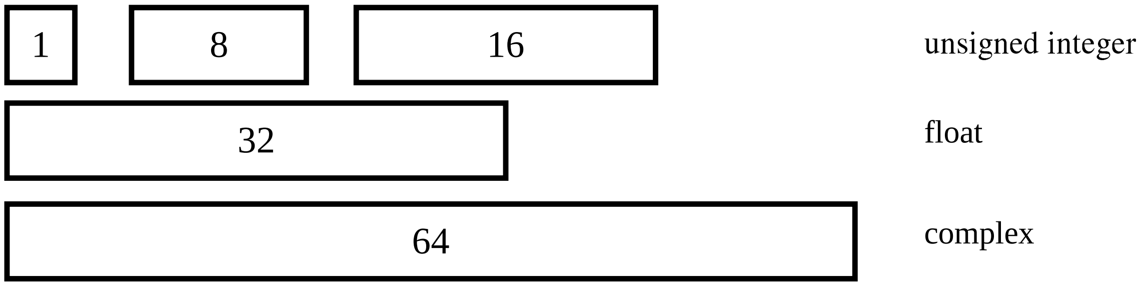
Planar Configuration. There can only be one image plane, in which multiple samples of a given pixel are continguous.

Comments. Comments can be added to the TIFF header in the form of character strings. The command line parameters are automatically recorded in the header. See section 6.4.

Archie Image File Format



A restricted TIFF 6.0 file format that can be streamed through Archie modules and still be read in Photoshop.



The interpretation of the pixel data depends on the number of bits/sample.

G_0 G_1 G_2 . . .	gray
G_0 S_0 G_1 S_1 G_2 S_2. . .	gray w/ soft matte
R_0 G_0 B_0 R_1 G_1 B_1 . . .	color
R_0 G_0 B_0 S_0 R_1 G_1 B_1 S_1 . . .	color w/ soft matte

Multiple samples of a pixel are positioned next to each other.

4.0 Released Modules

Several modules have been implemented to date and more will be added in the future. Brief descriptions of the modules, the algorithms that they implement and the command lines to call them are given in this section.

4.1 Command Line Arguments

A flag is a letter, a word, or group of letters that is preceded by a dash, or minus sign. The flag may or may not have arguments following it on the command line. Multiple arguments are separated by spaces. Arguments that follow can be negative, i.e. be preceded by a minus sign.

There are a few command line flags and arguments that are common with almost all of the modules. These arguments, or parameters, have to do with specifications of input and output files and of windows or regions of interest.

-*i filename*

If this flag is present, it is the name of the file to use as input. If it is not present, then the input image is expected on the standard input pipe.

-*o filename*

If this flag is present, it is the name of the file to use as output. If it is not present, then the output image is written to the standard output pipe.

-*[f]win x y w h*

The -*win* flag specifies a window, consisting of four parameters, that indicates the region of the image to process. This rectangular region is *w* pixels wide and *h* scanlines high and is located *x* pixels in from the left edge and *y* scanlines down from the top of the image.

If the -*fwin* flag is present, then the four parameters following are floating point numbers between 0.0 and 1.0 and they represent the fraction of the width or height of the image to skip or process.

The square brackets around the *[f]* indicate that the *f* is optional. The brackets are not typed into the command line. In other words, in the above example, there are two options available, either -*win* or -*fwin*.

4.2 Module Descriptions

append -- append one image to another

usage: append [-a] filename [-i input] [-o output] -[r | l | t | b]

One image is appended to another. The image being appended is designated with the *-a* flag, followed by the file name. If this is the first argument, then the *-a* flag is not required.

The input image is the image to which the other image is appended. The input image is specified by the *-i* flag or is on standard in.

The way in which the image is appended is specified with one of four flags, *-r, -l, -t* or *-b*. These refer to appending the image on the right, left, top or bottom, respectively. One of these flags must be present on the command line.

The sides of the two images that join together must have the same length. In other words, if the image is appended to the right or left, then the two images must have the same number of scanlines. If the image is appended at the top or bottom, then the two images must have the same number of pixels.

Both images must have the same number of bits/sample and samples/pixel.

crop -- limit the image to specified rectangular region

usage: crop [-i input] [-o output] -[f]win x y w h

A window flag, either integer *-win*, or floating point *-fwin*, is required. The input image is read from the input file specified by *-i* or from standard in. Only the image data in rectangular region specified by the window is transferred to the output image.

header -- print out the header information from the image file

usage: header [[-i] input]

The header information from the image file is read and printed on standard out. The image data is then read from the input and a check is made to see if the stated amount of image data is present. If any of the known TIFF tags are present, their values are also printed to standard out.

histogram -- create an image containing a plot of the histogram of the image

usage: histogram [-i input] [-o output] -[f]win x y w h [-red] [-green] [-blue] [-key] [-solid]

An image is read on the input, a histogram is made for each sample and a plot is drawn. The output image contains a 512 x 512 plot with an 8 pixel border on the left and an 8 scanline border on the bottom.

If any of the four flags, *-red*, *-green*, *-blue*, or *-key* (black) are present, then only those present are drawn on the plot. Otherwise all four are drawn. The key plot contains the histogram of either the luminance of a color image or the histogram of a monochrome image.

The default is to draw lines between the points in the histogram. If the *-solid* flag is present, then only the points are drawn.

metadata -- add metadata to the image file header

usage: metadata [[-i] input] [-o output] [-replace tag string] [-append tag string] [-xres num denom] [-yres num denom]

This module will change the metadata in the TIFF tags in the header. Unlike the other modules, the actual command executed by this module is not stored in the header.

If the *-replace* flag is present, then the the TIFF tag in the header is replaced with the accompanying string. The recognized tags are: "DocumentName", "ImageDescription", "Make", "Model", "PageName", "Software", "DateTime", "Artist", "HostComputer", "Copyright".

If the *-append* flag is present, then the string is appended to any existing strings associated with the tag in the header. In accordance with the TIFF 6.0 standard, each string ends with a zero character, i.e. a byte of value 0. If two strings are appended together, then each still ends with a zero byte.

If the *-xres* or *-yres* flags are present, then the resolution parameters are set in the TIFF header. The resolution value set is given by the ratio num/denom.

normalize -- locally adjust contrast and brightness

usage: normalize [-i input] [-o output] [-w width] [-strobe] [-tungsten] [-ultraviolet]

The contrast and brightness of each color channel are adjusted locally over a square region specified by the *-w* flag. The variance and mean are measured within the window, centered around the pixel being adjusted, and the pixel is scaled so that the mean value goes to 128 and black to white covers 6 sigma.

If the -*ultraviolet* flag is present, then the green channel is substituted for the red channel and the blue channel is made brighter by the value of one standard deviation. The -*strobe* and -*tungsten* flags are unused at this time.

The output image is always 8 bits/sample and it will have the same samples/pixel as the input image.

packimage -- convert image to 8 bits/sample unsigned integer

usage: packimage [-i input] [-o output] [-minmax min max] [-sigma] [-width w] [-luminance] [-verbose]

This module converts an image to 8 bits/sample from 16 or 32 by stretching the contrast based on the range of input pixel values. The purpose is to move the image information into a visible range.

If the -*minmax* flag is present, the *min* and *max* values are used to linearly stretch the image. Any value at or below *min* is made black and any value at or above *max* is made white.

If the -*sigma* flag is present, the standard deviation and mean of the whole image are calculated and the image values are stretched so that the mean is at 128 and black and white are 6 standard deviations apart. If the -*width* flag is present, then black and white are *2*w* standard deviations apart.

If neither the -*minmax* nor the -*sigma* flag is specified, then the image is linearly stretched from the actually minimum and maximum of the image. The minimum and maximum are the endpoints of all the image samples.

If the -*luminance* flag is present, then the stretch parameters (*min* and *max*, or variance and mean) are calculated from the luminance (.3R+.6G+.1B) and applied equally to all channels.

If the -*verbose* flag is present, then the statistics of the image are written to stderr.

pseudocolor -- combine two images together into one pseudocolor image

usage: pseudocolor -u UV -v [tungsten | strobe] [-o output] [-w width] [-cv channel] [-cu channel]

There are two input images, an ultraviolet image specified by the -*u* flag and a visible image specified by the -*v* flag. A single color channel from each file is normalized then combined together into a pseudocolor file. The red channel of the pseudocolor image is taken from the visible file. The green and blue channels of the pseudocolor image are the same and are taken from the ultraviolet image.

A spatially local normalization of contrast and brightness is performed on each color channel. This is the same normalization performed in the normalize module. The *-w* flag specifies the size of the square region over which the normalization is performed around each pixel.

By default, the red channel of the visible image and the blue channel of the ultraviolet channel are used to make the pseudocolor image. If the *-cu* flag is present, a different channel for the ultraviolet image can be specified, where 0 is red, 1 is green and 2 is blue. If the *-cv* flag is present, the channel of the visible image can be specified.

pushbutton -- display the difference between two images

usage: pushbutton -u UV -v [tungsten | strobe] -o output [-w width] [-statistics redmin redmax bluemin bluemax] [-pseudocolor] [-normalized]

The same two input images for the pseudocolor module are used in the pushbutton module. The same two channels are used also. The difference is that the pushbutton module produces a monochrome image that shows the difference between the two images.

The *-statistics* flag specifies the min and max points in the red and blue channels to be used to linearly stretch the two channels before subtracting. This flag must be present to produce the normal monochrome pushbutton image.

If the *-pseudocolor* flag is present, then the two channels are not subtracted, but are put into the channels of a pseudocolor image on the output. The only difference between this pseudocolor output and the image produced by the pseudocolor module is that this output has a global adjustment of the contrast, while the pseudocolor module does a local adjustment over a sliding window. The *-statistics* flag must be specified for this to complete.

If the *-normalize* flag is present, then a local adjustment of the contrast and brightness is made over the two channels, rather than the global adjustment. The output image is then the difference between the two channels. This mode does not use any information that may be specified with the *-statistics* flag.

readtif -- read a TIFF file and convert to the form used in Archie

usage: readtif [-i] input [-o output] [-[f]win x y w h] [-d directory] [-verbose]

The input filename argument is required. The input file cannot be read from standard in. If the first argument is the input filename, the *-i* flag is not required. Readtif can read TIFF files in which the scanlines and metadata are not separated or contiguous, so it must read from the file directly.

If the *-d* flag is present, an alternate image directory can be specified. The first (default) directory is 0.

The *-verbose* flag causes the TIFF tag information to be printed to stderr.

A rectangular window of the input image can be specified to limit the content of the output image.

rect -- blur the input image with a rect function

usage: rect [-side] width [height]

The input image is blurred with a rectangularly shaped filter that has a constant value and a unit area. The calculation is done in a fast averaging method in which the computation is a constant and is not a function of the size of the blur window.

The size of the rectangular window is specified with the *-side* flag. This specification is required, however, if it is the first flag, the *-side* flag is not required. If there is only one argument to the *-side* flag, then it is the *width* and the *height* of the window. If two arguments are present, then the second argument is the *height.*

rotate -- rotate the image in 90 degree increments

usage: rotate [[-i] input] [-o output] -angle theta

The *-angle* flag is a required argument. The parameter theta must be a multiple of 90 degrees and may be positive or negative. A positive angle of rotation is in the counter-clockwise direction. This module may be extended to arbitrary angles in the future.

scale - magnify or reduce the size of an image

usage: scale [-i input] [-o output] -[a]f x [y]

The image is increased or decreased in size by a nearest neighbor scaling, i.e. the closest pixel to scaled position is used.

The scale factor is specified with the *-f* flag. It may be either integer or floating point. Only one parameter is expected to follow the flag and both directions are scaled by the same factor.

The *-af* flag specifies anamorphic scaling, where the scale factors in *x* and *y* are different. This flag has two parameters following it.

statistics -- compute the variance and mean of the input image

usage: statistics

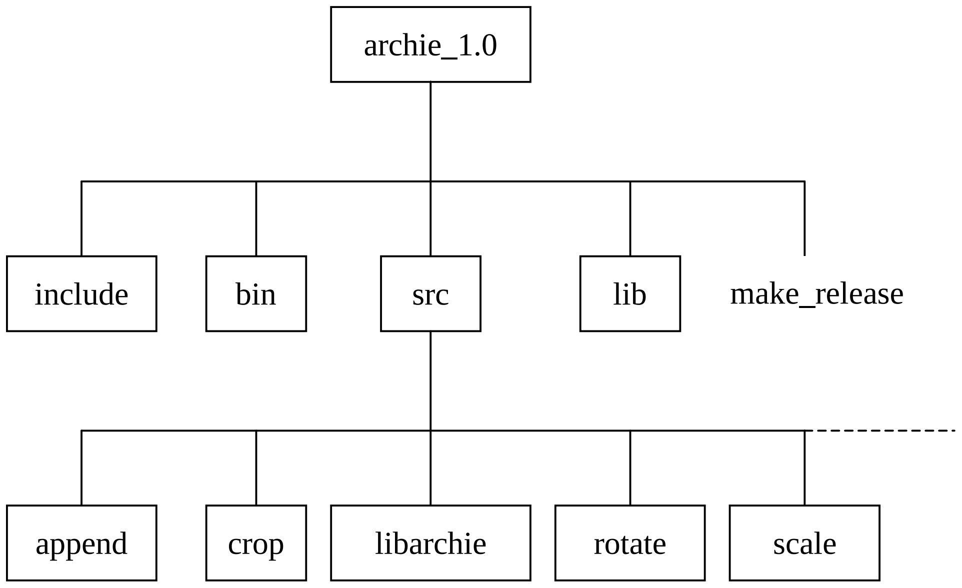
The complete input image is read and the standard deviation, mean, max and min values of the image are computed and written to standard out.

5.0 Compiling the Software

The software source code is packaged in a tar file, archie_1.0.tar. The source code can be extracted with the following command:

```
tar xvf archie_1.0.tar
```

The extracted files will create the Archie release directory, which has the following structure:



The cshell command file, *make_release*, will compile the Archie 1.0 software and place the compiled results in the *bin* directory. Each individual module has it own directory under *src* that contains the source code and a *Makefile* to compile it.

5.1 Make_release

The code is compiled by executing the command file *make_release*. This will clean up old versions and compile the complete package in the proper order.

The first operation is to change into each individual *src* directory and clean any previously compiled code. This is accomplished by executing the command, *make clean*, to remove all previous *.o file and the compiled executable code.

The second operation is to compile the library routines. The library routines are compiled and the library, libarchie.a, is constructed and copied to the lib directory where it can be referenced by the individual modules.

The third operation is to change to each individual directory and compile the individual module. The Makefile refers to the released library two levels higher in the directory structure. The compiled executable code is copied to the bin directory two levels higher.

For later changes to individual modules, the code can be recompiled by simply executing the command, *make*, while in the individual source directory. If a change is made to the library, then all of the source code needs to be recompiled.

6.0 Writing a Software Module

The requirements of a software module are to:

- read command line arguments
- read any input image headers
- resolve any conflicts and halt execution, if necessary
- write the new header reflecting changes to be made to the image
- stream the image data through the module
- make the changes to the image data as it travels through

These requirements are met with the following software structure:

The *main()* procedure defines the basic order of processing the command line, the headers and finally the data. This procedure should be the same for all modules.

The *getargs()* procedure defines the command line arguments that this module will recognize and calls any errors in the command line arguments.

The *processhandle()* procedure checks the characteristics of the input images against the user requests in the command line and raises any conflicts or final errors that it finds.

Finally, now that everything checks out, the *processdata()* procedure actually executes the processing to be done on the input images.

A record of changes made to this module is kept in comments at the end of the file.

```
/* sample.c
 *
 */

main(argc, argv)
  int argc;
  char *argv[];
  {
  getargs(argc, argv);
  input = gethandle(fdin);
  processhandle();
  puthandle(fdout, output);
  processdata();
  exit(0);
  }

void getargs(argc, argv)
  int argc;
  char *argv[];
  {
  }

void processhandle()
  {
  }

void processdata()
  {
  }

/* Change Log
 * K. Knox, 28-May-02, ...
 */
```

6.1 Command Line Arguments

The command line arguments are read in the procedure, *getargs().* An example of this procedure, taken from the rotate module, is shown on the next page. This section explains that example procedure in detail and shows how it is constructed.

The module, *rotate*, has three flags that it recognizes, *-in* , *-out* and *-angle.* At the beginning of the procedure, it loops over all of the arguments, starting with argument 1. (Argument 0 is just the name of the program that was called.)

The loop checks each argument to see if it begins with *-i, -o, or -a.* It only checks the first two letters, because there is no ambiguity in these flags after two letters.

If it finds one of these flags, it sets the parameter, *isflag*, to a defined constant value that identifies the flag and the number of expected arguments. For example, if the *-angle* flag is encountered, then isflag is set to the defined constant *ANGLE*, which is equal to 3*256+1. The number multiplying 256 is an arbitrary value to identify the flag. The trailing +1 is the number of arguments that this flag expects to be following it.

If this is not a flag, and this is the first argument, then it is assumed to be the input image file name and the local variables are set *as if* the rest of the loop had been completed.

Once it has checked for flags, the variable *isflag* is set to zero or to a non-zero value. The loop branches into two paths depending on whether this argument was or was not a flag.

If this argument is a flag, then first the variable *outstanding* is checked to see if there are any outstanding arguments still being expected by a previous flag. If so, an error is called. If not, then the variable *outstanding* is set to the number of expected arguments for this flag, that was stored in the lower byte of defined constant, in this example, +1.

We now loop to check the second argument. First, a test to see if it is a flag. If the test shows that it is not, it last checks if this is the first argument. Since it is not, it goes on to the test of the variable, *isflag.* It is not a flag, so it branches into the section that deals with following arguments.

Which flag is expecting this argument is stored in the variable, *flag.* A check is made and the appropriate action is taken. For the case of an input or output image file name, the argument is stored in the appropriate global variable. For the case of the *angle* argument, the argument is scanned for a floating point value and stored in the *angle* global variable.

Once all the arguments have been checked, it drops out of the loop and looks to see if all the arguments are present. If no angle has been specified or if an angle that this module cannot handle has been specified, an error is called.

The input and output file names are checked. If they are set, then the appropriate files are opened. If they are not set, then either *standard in* or *standard out* is specified for I/O.

Last, a library routine is called to automatically record the command line in the header.

```
#define INPUT    1*256+1
#define OUTPUT   2*256+1
#define ANGLE    3*256+1

void getargs(argc, argv)
  int argc;
  char *argv[];
  {
  int n;
  int flag = 0;
  int isflag = 0;
  int outstanding = 0;
  for (n=1; n < argc; n++)
    {
    /* Check this argument for flags. */
    if (strncmp(argv[n], "-in", 2) == 0) isflag = INPUT;
    else if (strncmp(argv[n], "-out", 2) == 0) isflag = OUTPUT;
    else if (strncmp(argv[n], "-angle", 2) == 0) isflag = ANGLE;
    else { isflag = 0; if (n == 1) { outstanding = 1; flag = INPUT; } }

    /* If this is a flag, see if it is interrupting the previous flag. */
    if (isflag)
      {
      if (outstanding > 0) error(err0);
      outstanding = isflag & 255;
      flag = isflag;
      }
    else
      {
      if (flag == INPUT) inputname = argv[n];
      else if (flag == OUTPUT) outputname = argv[n];
      else if (flag == ANGLE) sscanf(argv[n], "%lf", &angle);
      else outstanding++;
      outstanding--;
      }
    }

  /* Check angle parameter. */
  if (angle != 180.0 && angle != -180 && angle != 90 && angle != -90)
error(err1);

  /* Check for the input file name and use stdin, if not there. */
  if (inputname != (char *) 0) fdin = open(inputname, O_RDONLY);
  if (fdin < 0) error(err2, inputname);

  /* Check for the output file name and use stdout, if not there. */
  if (outputname != (char *) 0) fdout = open(outputname, O_WRONLY |
O_CREAT, (unsigned int) 0666);
  if (fdout <= 0) error(err2, outputname);

  /* Create the command line as a comment. */
  commandcomment(argc, argv);
  }
```

6.2 Input and Output Handles

The header information is stored in a structure called a *handle*. The calls to read and write a handle are made in the *main()* procedure. In between reading the input handle and writing the output handle, the output handle needs to be constructed and any conflicts need to be resolved.

```
/* Archie.h
 *
 */

struct HANDLE
  {
  int ImageWidth;
  int ImageLength;
  int BitsPerSample;
  int SamplesPerPixel;
  int SoftMatte;
  int SampleFormat;
  int PhotometricInterpretation;
  int ResolutionUnit;
  double XResolution;
  double YResolution;
  };
```

On the right, is the handle structure that is made available to the programmer. The procedure *gethandle()* that is executed in *main()* returns this structure. In this structure is stored the information about the width and height of the image, the interpretation of the pixel data, whether or not there is a soft matte present, and the resolution of the image.

If any values in the input structure are changed, it has no effect on the image or the processing.

On the other hand, an output structure can be created by duplicating an existing structure or by creating a new structure. These values can be changed, up until the time the structure is written on the output using the procedure, *puthandle().*

The output structure is created in the procedure, *processhandle().* An example of this procedure from the *crop* module is shown on the next page.

The *processhandle()* procedure has two tasks, to resolve any conflicts between the command line requests and the characteristics of the input image, and to create the output handle.

First, the procedure get the information about the image from the input handle that was read with *gethandle()* in *main().* The only concern that *crop* has is to make sure that the number of bits/sample is either 1 or a multiple of 8. An error is called if that is not the case.

Next, it checks the window that was specified by the user in the command line arguments and makes sure that the window does not lie outside the image, calling an error if it does.

Lastly, it creates an output handle, by duplicating the input handle and setting the width and height parameters of the output image handle structure to the width and height of the window that was specified.

The new values in the output structure do not take effect until the output handle is written to the output using the procedure, *puthandle().*

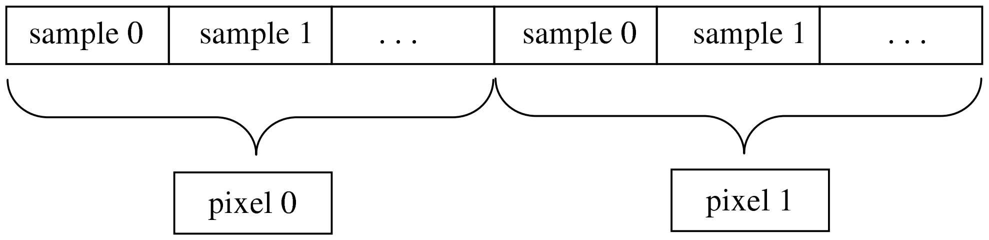
```
/* crop.c
 *
 */

void processhandle()
  {
  /* Read file sizes. */
  pixels = input->ImageWidth;
  scanlines = input->ImageLength;
  BitsPerSample = input->BitsPerSample;
  SamplesPerPixel = input->SamplesPerPixel;

  /* Check the image parameters. */
  if (BitsPerSample != 1 && (BitsPerSample & 7) != 0) error(err4,
BitsPerSample);

  /* Check the window parameters. */
  if (count)
    {
    x = window[0];
    y = window[1];
    w = window[2];
    h = window[3];
    }
  else
    {
    x = pixels*fwindow[0];
    y = scanlines*fwindow[1];
    w = pixels*fwindow[2];
    h = scanlines*fwindow[3];
    }
  if (x < 0 || y < 0 || (x+w) > pixels || (y+h) > scanlines) error(err5, x,
y, w, h, pixels, scanlines);

  /* Make output handle. */
  output = duphandle(input);
  output->ImageWidth = w;
  output->ImageLength = h;
  }
```

6.3 Reading and Writing Scanlines

An example of how to stream image data, taken from the *crop* module, is shown on the next page. This module does very simple processing, since it does not need to understand the data, it only needs to limit its size.

At the beginning of the *processdata()* procedure, input and output buffers are defined. These buffers are specially created using the Archie library routine, *createscans().* This routine only needs to know the image data I/O handle and how many scanlines are to be stored in the buffer. The necessary image information that this procedure needs is determined from the handle structure.

The data is stored in the buffer as packed pixels. If a pixel has multiple samples, then the samples for each pixel are stored continuously. Since the only allowed bits/sample values are 1, 8, 16, 32, and 64, for all but the first, each pixel is an even multiple of bytes. For the case of 1 bit/sample, the pixels are packed into the scanline ignoring the byte boundaries. Padding to fill out to a byte boundary is done only at the end of a scanline. No padding is required for the other cases. The number of bytes in a scanline is therefore, *nbytes = (BitsPerSample*SamplesPerPixel*ImageWidth+7)/8.* The first scanline begins at byte 0 and any subsequent scanlines begin at intervals of *nbytes*.



Back to the *crop* code fragment, the main processing loop cycles over all of the input image scanlines, reading one scanline into the input buffer for each iteration of the loop.

Since the task of *crop* is to pick out a rectangular region of the input image, a check is made to see if the current scanline is within the window. If it is, then the portion of that scanline that is horizontally within the window is copied to the output scanline buffer.

At the end of the loop, an output scanline is written for each line that lies within the window.

A switch statement within the loop is used to treat the scanline data differently depending on the number of bits within each sample. A special case is necessary for 1 bit/sample. A temporary buffer is used to *unpack* the input scanline from packed bits into bytes. The unpacked bytes can be easily cropped and re-packed into the output scanline.

Since the input and output scanline buffers contain information about the length of the scanline, they cannot be freed using the standard *free()* system call, but they require an Archie library routine, *freescans().*

```
/* crop.c
 *
 */

void processdata()
  {
  int m, n;
  unsigned char *utmpIn, *utmpOut, *temp;
  unsigned short *stmpIn, *stmpOut;
  unsigned long *ltmpIn, *ltmpOut;
  double *dtmpIn, *dtmpOut;
  temp = (unsigned char *) malloc(SamplesPerPixel*pixels+7);
  utmpIn = (unsigned char *) createscans(input, 1);
  utmpOut = (unsigned char *) createscans(output, 1);
  for (n=0; n < scanlines; n++)
    {
    readscans(fdin, input, utmpIn);
    if (n >= y && n < (y+h))
      {
      switch (BitsPerSample)
        {
        case 1:
          unpack(utmpIn, temp, SamplesPerPixel*pixels);
          pack(utmpOut, temp+SamplesPerPixel*x, SamplesPerPixel*w);
          break;
        case 8:
          for (m=0; m < SamplesPerPixel*w; m++) utmpOut[m] =
utmpIn[m+SamplesPerPixel*x];
          break;
        case 16:
          stmpIn = (unsigned short *) utmpIn;
          stmpOut = (unsigned short *) utmpOut;
          for (m=0; m < SamplesPerPixel*w; m++) stmpOut[m] =
stmpIn[m+SamplesPerPixel*x];
          break;
        case 32:
          ltmpIn = (unsigned long *) utmpIn;
          ltmpOut = (unsigned long *) utmpOut;
          for (m=0; m < SamplesPerPixel*w; m++) ltmpOut[m] =
ltmpIn[m+SamplesPerPixel*x];
          break;
        case 64:
          dtmpIn = (double *) utmpIn;
          dtmpOut = (double *) utmpOut;
          for (m=0; m < SamplesPerPixel*w; m++) dtmpOut[m] =
dtmpIn[m+SamplesPerPixel*x];
          break;
        }
      writescans(fdout, output, utmpOut);
      }
    }
  free(temp);
  freescans(utmpIn);
  freescans(utmpOut);
  }
```

6.4 Comments

Comments that contain information about the history of the image file can be read, modified and/or written to the header. These comments are stored as ASCII strings in specific TIFF tags that are part of the definition of the TIFF 6.0 format.

On the right, is the list of comments that are defined for the Archie 1.0 software package. For the Archimedes Palimpsest project, these tags are used to store specific information related to the manuscript.

```
/* Archie.h
 *
 */

struct COMMENT
  {
  struct COMMENT *next;
  unsigned long tag;
  unsigned long length;
  char *strings;
  };

/* Comment names. */
#define COMMENT_DocumentName       269
#define COMMENT_ImageDescription   270
#define COMMENT_Make               271
#define COMMENT_Model              272
#define COMMENT_PageName           285
#define COMMENT_Software           305
#define COMMENT_DateTime           306
#define COMMENT_Artist             315
#define COMMENT_HostComputer       316
#define COMMENT_Copyright        33432
```

DocumentName is set to “Archimedes Palimpsest”.

ImageDescription records the type of illumination, either tungsten, strobe or ultraviolet.

Make and Model describe the camera.

PageName is the folio number of the page imaged in this file.

Software is automatically set by the library routines to “Archie 1.0”.

DateTime is automatically set to the date and time the processing was done.

Artist is set to the names of the people who created the image.

HostComputer is where the command line used to process this image is stored. It is automatically appended to whatever history is already in this tag.

Copyright describes the date and owner of the copyright of the image.

Only the three tags, Software, DateTime and HostComputer are automatically filled in by the Archie 1.0 software package library routines. All the other comment fields are modifiable by an individual module.

The descriptions of the specific library calls to modify comments are given in section 7.3. A brief code example is given on the next page.

6.5 Example of Writing Comments

Comments are written using the Archie library routines that specifically deal with comments. An example code fragment from the *append* module is shown below.

The *append* module is interesting in that it has two different input files and it needs to combine the comment fields of the HostComputer tag from both images to record the complete processing history of all three images in the output image.

When the input handle is duplicated to make the output handle, the comments of the input file are copied with it. If nothing is done to modify the comment fields, they will be carried along and recorded in the output image file.

The procedure *getcomment()* returns a pointer to the COMMENT structure shown on the previous page and is used below to get any existing comments stored in the HostComputer tag from the two input files.

The two HostComputer comments from the two input files are combined together using the procedure, *appendcomments().* The procedure, *putcomment(),* then is used to replace any comment of this type in the output handle with the newly combined comment. When the output handle is subsequently written to the output using *puthandle(),* these comments will be recorded in the output file.

```
/* append.c
 *
 */

void processhandle()
  {
  struct COMMENT *commentOut, *commentApp;

  .. . .

  /* Make output handle. */
  output = duphandle(input);
  if (direction == LEFT || direction == RIGHT)
    output->ImageWidth = input->ImageWidth+append->ImageWidth;
  else output->ImageLength = input->ImageLength+append->ImageLength;

  /* Combine the processing comments of the two images. */
  commentOut = getcomment(output, COMMENT_HostComputer);
  commentApp = getcomment(append, COMMENT_HostComputer);
  commentOut = appendcomments(commentOut, commentApp);
  if (commentOut != (struct COMMENT *) 0) putcomment(output, commentOut);
  }
```

Only the HostComputer tag is modified in this example. All other comment fields in the output image will be the same as those in the input image.

7.0 Library Routines

This section contains the detailed descriptions of the available Archie 1.0 library routines, their arguments, how to call them and what they do.

7.1 Handles

```
struct HANDLE *gethandle(fd)
  int fd;
```

Gethandle() takes as an input, a input file descriptor. This parameter is an INT and is returned by the *open()* system call, or it can be 0 for the standard input pipe.

It returns a pointer to a HANDLE structure, which is defined on page 16. This structure contains the size of the image and information about the interpretation of the pixel data. Changes made to this structure will have no effect on the file described by *fd*.

Gethandle() checks the stated byte order of the input file along with the byte order used by the computer executing the software and interprets the input data accordingly.

Non-public information is also stored within the handle returned by this routine. This information includes the comments within the header, the byte order being used, and the number of bytes in a scanline.

As a consequence of the private information within the handle, only handles created by the library routines will work with the other routines in the package.

```
void puthandle(fd, handle)
  int fd;
  struct HANDLE *handle;
```

Puthandle() takes two input arguments. The first is a file descriptor of an output image file, or it can be 1 for the standard output pipe.

The second argument is a *handle* of the output image. This handle should be created using one of the following library routines, such as *duphandle()* or *createhandle().* Because handles contain private information, they will not work if they are constructed by any other means.

Any changes made to the parameters within this output file handle, to reflect the characteristics of the output image, as shown in the code on page 17, must be made before *puthandle()* is called. They will have no effect afterwards.

Puthandle() automatically changes some comment fields in the header. The Software tag is set to be "Archie 1.0". The DateTime tag is set to the current date and time. Lastly, a check is made for the command line to the module and it is appended to any existing statements in the HostComputer tag.

```
struct HANDLE *duphandle(handle)
  struct HANDLE *handle;
```

Duphandle() will duplicate the input HANDLE structure and return a pointer to the copy. The routine is used create a handle for a new image file that is very similar to a previously existing handle.

The routine copies both the public and the private structures associated with the handle. A separate copy of any comments associated with the input handle will be made and associated with the copy that is returned.

No changes made to the duplicate handle that is returned will be reflected in anything associated with the input handle or file. Any changes that are made will affect only the output file that is being executed together with the handle by *puthandle()*.

```
struct HANDLE *createhandle(ImageWidth, ImageLength, BitsPerSample,
        SamplesPerPixel, PhotometricInterpretation, SoftMatte, SampleFormat,
        XResolution, YResolution, ResolutionUnit, ByteOrder, Comments)
  long ImageWidth, ImageLength, BitsPerSample, SamplesPerPixel;
  long PhotometricInterpretation, SoftMatte, SampleFormat;
  double XResolution, Yresolution;
  int ResolutionUnit;
  char ByteOrder;
  struct COMMENT *comments;
```

Createhandle() will create an image file handle when there are no existing handles to copy. All of the arguments listed above are required and must be of the appropriate type.

PhotometricInterpretation has one of two values from Archie.h, either PhotometricGray or PhotometricRGB.

SoftMatte has the value 0 or 1, indicating if a soft matte exists in the image.

SampleFormat has one of four values from Archie.h, either Unsigned Integer, SignedInteger, FloatingPoint, or Undefined.

ResolutionUnit is 2 for inches.

ByteOrder is ‘I’ for Intel (little endian) or ‘M’ for Motorola (big endian).

Comments is a pointer to a queue of comment structures. The first element of each comment structure points to the next. The last structure on the list has a zero “next” pointer.

7.2 Scanlines

```
unsigned char *createscans(handle, nscans)
  struct HANDLE *handle;
  int nscans;
```

Createscans() returns a pointer to a buffer that can hold *nscans* scanlines of an image described by *handle*. This buffer is passed as an argument to *readscans()* and *writescans().*

The pointer returned by *createscans()* is the beginning of the pixel data for the first scanline in the buffer. If there are additional scanlines in the buffer, then they are located at intervals of *nbytes = (BitsPerSample*SamplesPerPixel*ImageWidth+7)/8.*

The buffer created by this routine cannot be freed using the *free()* system call, see below.

```
void readscans(fd, handle, buffer)
  int fd;
  struct HANDLE *handle;
  unsigned char *buffer;
```

Readscans() reads scanlines from *fd* in *buffer*. The number of scanlines the buffer can hold was specified when it was created by *createscans().* Every time *readscans()* is called, a new set of scanlines is stored in *buffer*, overwriting whatever was in the buffer previously.

```
void writescans(fd, handle, buffer)
  int fd;
  struct HANDLE *handle;
  unsigned char *buffer;
```

Readscans() writes the scanlines to *fd* that are contained in *buffer*. The number of scanlines written from the buffer was specified when it was created by *createscans().* Calling *writescans()* does not change the image data contained with *buffer*.

```
void freescans(buffer)
  unsigned char *buffer;
```

Freescans() frees the space used by the buffer that was allocated by *createscans().*

7.3 Comments

```
struct COMMENT *getcomment(handle, tag)
  struct HANDLE *handle;
  unsigned long tag;
```

Getcomment() has two arguments, an input handle and the TIFF tag identified with the desired comment. The available TIFF tag identifiers are defined in Archie.h and are shown on page 20. The pointer that is returned, points to a COMMENT structure, also defined in Archie.h and shown on page 20.

This return value is a pointer to the actual comment within the specified handle. It is not a copy of the comment. If anything is changed within this COMMENT structure, it will be reflected in a change in the comments associated with the input handle.

A null pointer is returned if no comment identified with this *tag* is present in the *handle*.

```
struct COMMENT *createcomment(tag, length, string)
  unsigned long tag, length;
  char *string;
```

Createcomment() has three arguments, a TIFF tag identifier, the length of the string, and the desired string to be associated with this comment. The routine allocates the space needed to create the COMMENT structure and returns a pointer to that structure. The "next" parameter will be set to zero.

The *length* is the number of bytes that it takes to hold the string, including a terminating zero byte.

The *string* may be a single string, terminated with a zero byte, or it may be a set of concatenated strings. The TIFF 6.0 format allows a string to consist of several zero-byte terminated strings placed one after the other. The length parameter includes all of the bytes to be included in the complete set of strings.

```
struct COMMENT *appendstring(comment, string)
  struct COMMENT *comment;
  char *string;
```

Appendstring() will append the zero-byte terminated *string* to the end of the string associated with the input *comment*. The length parameter within the *comment* is adjusted accordingly. New space is allocated to hold the new string and the old space is freed. If the *comment* or the *string* is zero, then the routine does nothing.

struct COMMENT ***appendscomments**(comment1, comment2)
  struct COMMENT *comment1, comment2;

Appendcomments() will append comments together. The two input comments are left alone, and a new comment is created that contains the string of *comment2* appended to the string of *comment1*. If either comment is zero, then a copy of the other comment is returned.

void **putcomment**(handle, comment)
  struct HANDLE *handle;
  struct COMMENT *comment;

Putcomment() searches the structure in *handle* looking for a comment of the same type as *comment*. If it finds one, then it replaces the old comment with the new *comment*.

void **commandcomment**(argc, argv)
  int argc;
  char **argv;

Commandcomment() extracts the command line arguments from the argument strings that were passed to *main()*. The extracted string containing the command line is placed in a global variable that is checked by *puthandle()*. If *puthandle()* finds a command line, it will append it to the HostComputer tag in the handle.

This routine should be called before writing out the output handle to ensure that the command line is added to the image processing history that is stored in the header of the image file.

7.4 Errors

void **error**(errormsg, arg0, arg1, …, arg9)
  char *errormsg;
  int arg0, arg1, …, arg9;

Error() will print the *errormsg* using *fprintf* to *stderr*, along with up to 10 arguments containing additional values, then it stops execution by calling *exit(1)*.

7.5 Pack and Unpack

```
void pack(bits, bytes, length)
 unsigned  char *bits, *bytes;
  int length;
```

pack() will convert a scanline of unsigned chars that are 1 bit/sample, in *bits*, to a scanline of unsigned chars that are 8 bits/sample, in the array *bytes*. The third argument, *length*, is the number of samples, whether packed or unpacked.

```
void unpack(bits, bytes, length)
 unsigned  char *bits, *bytes;
  int length;
```

unpack() will convert a scanline of unsigned chars that are 8 bits/sample, in the array *bytes*, to a scanline of unsigned chars that are 1 bit/sample, in the array *bits*. The third argument, length, is the number of samples, whether packed or unpacked.