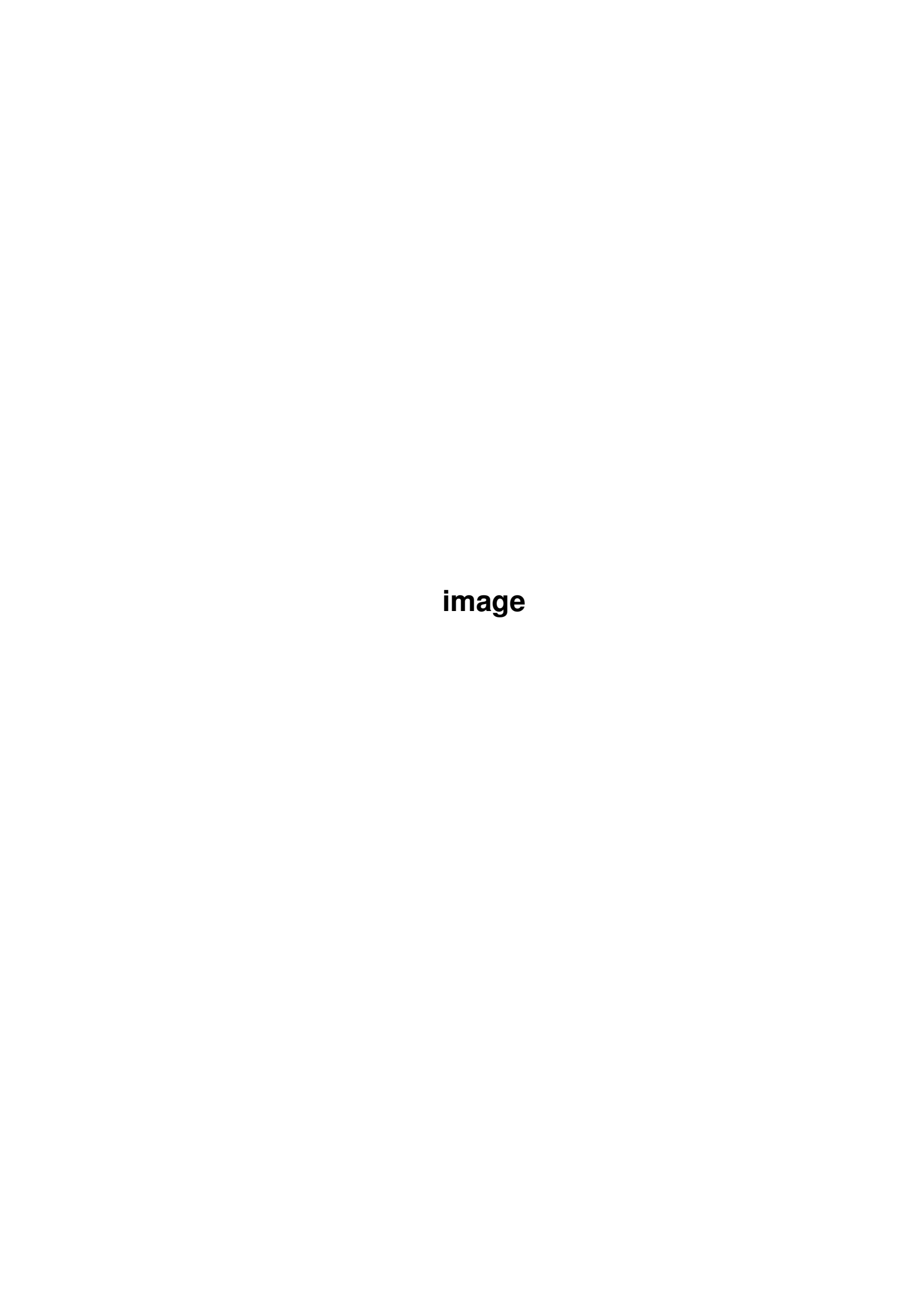
image

COLLABORATORS			
	TITLE : image		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 29, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

image

1.1 S0

```
LightWave Images -- Stuart Ferguson 1/13/95

1  Introduction
   (1)  Public declarations
   (2)  Public forward definitions

2  Image I/O Server Interface
    2.1  Image Loaders
       (3)  Public types
    2.2  Image Savers
       (4)  Public types
    2.3  Result Value
       (5)  Public declarations
    2.4  Image Transfer Protocols
  2.4.1  Color Protocol
     (6)  Public types
  2.4.2  Index Protocol
     (7)  Public types
  2.4.3  Generic Protocol
     (8)  Public types
     (9)  Public forward definitions
  2.4.4  Error Handling
  2.4.5  Misc Types
     (10)  Public declarations
     (11)  Public declarations

3  Test Server
    3.1  Targa Reader
       (12)  Targa types
       (13)  Targa functions
       (14)  Targa utilities
       (15)  Read targa data
       (16)  Read targa lines
       (17)  Read uncompressed targa line
       (18)  Read compressed targa line
       (19)  Read a targa pixel element into 'bgra'
       (20)  Store 'bgra' pixel to line buffers
    3.2  Targa Saver
```

1.2 s1

This module provides interfaces for dealing with image types commonly employed by LightWave users. This allows the loading and saving of large, deep images in an expandable set of formats, and for accessing the data in a uniform manner regardless of the underlying data format. This interface is designed with plug-in image loaders and savers in mind and it provides some built-in IFF format support.

Image types are given by the following values. RGB24 is an image with eight bits each of red, green and blue data for each pixel. GREY8 is an image with eight bits of greyscale value at each pixel. INDEX8 is an image with up to eight bits of color index at each pixel, mapped through a 24 bit color table.

(1) Public declarations

```
#define IMG_RGB24       0
#define IMG_GREY8       1
#define IMG_INDEX8      2
. . .
```

Image color component, grey or index values are all unsigned chars scaled from 0 to 255.

(2) Public forward definitions

```
typedef unsigned char            ImageValue;
. . .
```

1.3 s2

The image input and output interfaces are designed to be extended with plug-in loaders and savers. As result, each interface really only defines the local data structure for the activation function.

1.4 s2.1

Image loaders are servers that are called sequentially until one is able to load the image file. An application will normally have a standard format in which images are saved, so that will normally be tried first after which other loaders may be tried in any order the host can determine. If loaders are just scanned in the host plug-in database they will be called in something like alphabetical order.

The activation call for a loader gets passed a pointer to a filename as well as callbacks for image data transfer. If the loader cannot open the file it sets the 'result' field to IPSTAT_BADFILE and returns. If it does not recognize the file format, it sets the result to IPSTAT_NOREC. If it can load the image, it calls the 'begin' callback with type of image protocol it would like. The loader then sends the data from the file to the host through the protocol and calls the 'done' callback when complete to allow the source to dispose of the protocol. These callbacks are called with the 'priv_data' pointer as the first field.

(3) Public types

```
typedef struct st_ImLoaderLocal {
  void             *priv_data;
  int               result;
  const char       *filename;
  Monitor          *monitor;
  ImageProtocolID (*begin) (void *, int type);
  void            (*done) (void *, ImageProtocolID);
} ImLoaderLocal;
. . .
```

1.5 s2.2

Image savers are servers of "ImageSaver" class that write an image out to a file in a single specific format. The save format is typically chosen directly by the user with an interface showing the user names for the servers, so no scanning or ordering is required.

The activation call for savers gets a filename, a requested protocol type, and a callback for the host to output its image data to the saver protocol. The flag in the 'sendData' callback can contain the IMGF_ALPHA bit if the saver can store alpha data and IMGF_REVERSE bit if the saver wants the data sent bottom to top rather than top to bottom. The saver should create a protocol and set flags most appropriate for the destination file format. The 'sendData' callback will return a non-zero error code if anything failed on the sending end or if the destination reports an error.

(4) Public types

```
. . .
typedef struct st_ImSaverLocal {
  void            *priv_data;
  int              result;
  int              type;
  const char      *filename;
```

```
    Monitor         *monitor;
    int            (*sendData) (void *, ImageProtocolID, int);
  } ImSaverLocal;
  . . .
```

1.6 s2.3

The result value indicates the status of the loader or saver upon completion. If the load or save was sucessful, the value should be IPSTAT_OK. If a loader fails to recognize a file as something it can load it should set the result to IPSTAT_NOREC. If the server could not open the file it should return IPSTAT_BADFILE. Any other error is just a generic failure of the loader or saver and so should set the result to IPSTAT_FAILED. Other failure modes might be possible if required in the future.

```
    (5)  Public declarations
  . . .
  #define IPSTAT_OK               0
  #define IPSTAT_NOREC            1
  #define IPSTAT_BADFILE          2
  #define IPSTAT_ABORT            3
  #define IPSTAT_FAILED          99
  . . .
```

1.7 s2.4

Images are passed from source to destination using an image protocol. Typically, the source will select the protocol type and the destination will create a protocol of that type. The source will then send the image data to the source by calling callbacks in the protocol. Both ends are then given an opportunity to clean up. This is called a pusher protocol since the source "pushes" the data at the destination rather than the destination pulling it.

There are two protocols for the three types of images: color and index protocols. The protocol ‘type’ can have any of the same values as image type and determines the callbacks in the protocol and what they do. Protocols contain a private data pointer which should be passed as the first argument to all the callbacks.

1.8 s2.4.1

The color protocol is used for the RGB and grey valued images (RGB24 and GREY8 types). The source starts the output by calling the 'setSize' function with the width and height of the image and flags. The flags can contain the IMGF_ALPHA bit to indicate that the source data contains an alpha channel. The source then sends the data by calling the 'sendLine' function with each image row number and a pointer to a line of image data and a line of alpha data, if any was indicated. For greyscale images, the image line consists of one image value per column in the image (G1 G2 ... Gw). For RGB images, this line data consists of three image values per column of the image in RGB order (R1 G1 B1 R2 G2 B2 ... Rw Gw Bw). The alpha data is in greyscale format.

```
    (6)  Public types
  . . .
  typedef struct st_ColorProtocol {
    int              type;
    void            *priv_data;
    void           (*setSize) (void *, int, int, int);
    int            (*sendLine) (void *, int, const ImageValue *,
              const ImageValue *);
    int            (*done) (void *, int);
  } ColorProtocol;
  . . .
```

1.9 s2.4.2

Colormap index images use the index protocol. The source must first call 'setSize' and 'numColors' with image size, flags and number of entries in the colormap. The source must then set the colormap by calling the 'setMap' callback for each entry in the colormap. Any entry which is not set is left undefined. The data in the image is then filled in using the 'sendLine' function just like the greyscale case except that the image values are not grey values but colormap indices. Alpha values are in greyscale data format.

```
    (7)  Public types
  . . .
  typedef struct st_IndexProtocol {
    int              type;
    void            *priv_data;
    void           (*setSize) (void *, int, int, int);
    void           (*numColors) (void *, int);
    void           (*setMap) (void *, int, const ImageValue[3]);
    int            (*sendLine) (void *, int, const ImageValue *,
              const ImageValue *);
    int            (*done) (void *, int);
  } IndexProtocol;
  . . .
```

1.10 s2.4.3

The generic protocol is either of these possibilities plus the type field for easy type identifcation.

```
    (8)  Public types
  . . .
  typedef union un_ImageProtocol {
    int              type;
    ColorProtocol    color;
    IndexProtocol    index;
  } ImageProtocol;

    (9)  Public forward definitions
  . . .
  typedef union un_ImageProtocol  *ImageProtocolID;
```

1.11 s2.4.4

There are two specific mechanisms for dealing with errors that occur while using image protocols. The destination can return error codes from the 'sendLine' and 'done' callbacks, and the source can pass an error code to the destination's 'done' callback.

If an error occurs in the source of a protocol, such as a failure partway though reading a file, the source can stop calling 'sendLine' prematurely. This will often trigger an error in the destination since it will have been keeping track of the amount of data sent. The source should then also pass a non-zero error code to the 'done' callback which will signal an error to the destination.

If an error occurs in the destination of a protocol, such as a failure partway through saving an image, the destination should start to return a non-zero error code from 'sendLine.' A well-written source will stop sending data when this happens, but the destination should be prepared to continue to get lines of data and to continue to return an error code. A failed destination should also return a non-zero error code from the 'done' callback.

1.12 s2.4.5

Flags to be passed to 'setSize' and 'sendData' callbacks.

```
    (10)  Public declarations
  . . .
  #define IMGF_ALPHA               1
  #define IMGF_REVERSE             2
  . . .
```

There are also some protocol macros defined to get the whole calling interface right.

```
    (11)  Public declarations
```

```
. . .
#define IP_SETSIZE(p,w,h,f)     (*(p)->setSize) ((p)->priv_data,w,h,f)
#define IP_NUMCOLORS(p,n)       (*(p)->numColors) ((p)->priv_data,n)
#define IP_SETMAP(p,i,val)      (*(p)->setMap) ((p)->priv_data,i,val)
#define IP_SENDLINE(p,ln,d,a)   (*(p)->sendLine)
   ((p)->priv_data,ln,d,a)
#define IP_DONE(p,err)          (*(p)->done) ((p)->priv_data,err)
```

1.13 s3

This is a very simple server designed to test an alternate image format. The single plug-in will load and save Targa 32 and 24 bit formats.

1.14 s3.1

The targa loader will recognize a targa file by reading the header into this data struct. The 'type' gives the compression format and interpretation of image data, 'bits' gives the pixel size and 'reverse' indicates if the lines will come bottom to top.

(12) Targa types

```
typedef struct st_TargaInfo {
  unsigned char    type, bits;
  short            width, height;
  int              reverse;
} TargaInfo;

#define CKPT_TGA_BADFILE        991
#define CKPT_TGA_NOREC          992
. . .
```

The main reader just reads the header, and if this can be matched as a targa image it reads the body. Errors will be captured by the exception context and will set the result code.

(13) Targa functions

```
  int
TargaLoader (
  long                    version,
  GlobalFunc             *global,
  ImLoaderLocal          *local,
  void                   *servData)
{
  ReadStrmID              strm;
  TargaInfo               tga;
```

```
    int                     fail;

    if (version != 1)
      return AFUNC_BADVERSION;

    if (!CkptCapture (fail)) {
      if (fail == CKPT_ABORT)
        local->result = IPSTAT_ABORT;
      else if (fail == CKPT_TGA_BADFILE)
        local->result = IPSTAT_BADFILE;
      else if (fail == CKPT_TGA_NOREC)
        local->result = IPSTAT_NOREC;
      else
        local->result = IPSTAT_FAILED;

      return AFUNC_OK;
    }

    strm = StrmReadOpen (local->filename, NULL);
    if (!strm)
      CkptRecover (CKPT_TGA_BADFILE);
    ARM1 (StrmReadClose, strm);

    if (!ReadTargaHeader (strm, &tga))
      CkptRecover (CKPT_TGA_NOREC);

    <Read targa data>

    StrmReadClose (strm);
    CkptEnd ();

    local->result = IPSTAT_OK;
    return AFUNC_OK;
  }
  . . .
```

The reader will check just a very few things in the header before it decides it can load the file. This might be a problem since targa files are much less self-identifying than others. Any values that are out of range cause this to return 0, indicating failure to recognize. If it returns 1, the info has been read.

```
    (14)  Targa utilities

    static int
  ReadTargaHeader (
    ReadStrmID               strm,
    TargaInfo               *tga)
  {
    unsigned char            byte, idLen;

    (*strm->readBytes) (strm, &idLen, 1);
    (*strm->readBytes) (strm, &byte, 1);
    if (byte)
      return 0;

    (*strm->readBytes) (strm, &tga->type, 1);
```

```
    if (tga->type != 2 && tga->type != 10)
      return 0;

    (*strm->skipBytes) (strm, 9);
    (*strm->readIWords) (strm, &tga->width, 1);
    (*strm->readIWords) (strm, &tga->height, 1);

    (*strm->readBytes) (strm, &tga->bits, 1);
    if (tga->bits != 24 && tga->bits != 32)
      return 0;

    (*strm->readBytes) (strm, &byte, 1);
    byte &= 0xF0;
    if (byte == 0)
      tga->reverse = 1;
    else if (byte == 0x20)
      tga->reverse = 0;
    else
      return 0;

    (*strm->skipBytes) (strm, idLen);
    return 1;
  }
  . . .
```

We will always send the data in RGB24 format, since we currently only recognize targa 32 and 24 bit formats. Buffers are allocated to transfer the rgb and alpha data in the right byte-packing order. The protocol is started and recovery actions are armed in case we fail partway through.

(15) Read targa data

```
  {
    ImageProtocolID          ip;
    ColorProtocol           *cp;
    ImageValue              *buf, *abuf;
    int                      bufSize, i, alpha;

    ip = (*local->begin) (local->priv_data, IMG_RGB24);
    if (!ip)
      CkptRecover (CKPT_IO_ERROR);

    alpha = (tga.bits == 32);

    bufSize = tga.width * 4;
    buf = NEW_Z (bufSize);
    ARM_Z (buf, bufSize);
    abuf = (alpha ? buf + 3 * tga.width : NULL);

    cp = &ip->color;
    IP_SETSIZE (cp, tga.width, tga.height, (alpha ? IMGF_ALPHA :
       0));

    ARM2 (local->done, local->priv_data, cp);

    MON_INIT (local->monitor, tga.height);
```

```
    if (local->monitor)
      ARM1 (local->monitor->done, local->monitor->data);

    if (CkptBegin ()) {
      ARM2 ((void*)cp->done, cp->priv_data, -1);
      <Read targa lines>
      CkptEnd ();
    }
    if (IP_DONE (cp, 0))
      CkptRecover (CKPT_IO_ERROR);

    MON_DONE (local->monitor);
    (*local->done) (local->priv_data, ip);
    FREE_Z (buf, bufSize);
  }
```

Basically we just read all the lines in forward or reverse order.
They may be compressed or not.

```
    (16)  Read targa lines

  for (i = 0; i < tga.height; i++) {
    int                       ln, x;
    unsigned char             bgra[4];
    ImageValue               *rgbBuf, *alphaBuf;

    rgbBuf = buf;
    alphaBuf = abuf;
    if (tga.type == 2) {
      <Read uncompressed targa line>
    } else {
      <Read compressed targa line>
    }

    ln = (tga.reverse ? tga.height - i - 1: i);
    if (IP_SENDLINE (cp, ln, buf, abuf))
      break;

    if (MON_STEP (local->monitor))
      CkptRecover (CKPT_ABORT);
  }
```

Uncompressed lines of data are just ‘width’ pixels which we read
sequentially.

```
    (17)  Read uncompressed targa line

  for (x = 0; x < tga.width; x++) {
    <Read a targa pixel element into ‘bgra’>
    <Store ‘bgra’ pixel to line buffers>
  }
```

A compressed line is enough pixels in literals and runs to fill a
scanline. If the scanline is not exactly filled, this is an error.

```
    (18)  Read compressed targa line
```

```
  x = 0;
  while (x < tga.width) {
    unsigned char            test;
    int                      count, k;

    (*strm->readBytes) (strm, &test, 1);
    count = (test & 0x7F) + 1;
    if (test & 0x80) {
      <Read a targa pixel element into 'bgra'>
      for (k = 0; k < count; k++) {
        <Store 'bgra' pixel to line buffers>
      }
    } else {
      for (k = 0; k < count; k++) {
        <Read a targa pixel element into 'bgra'>
        <Store 'bgra' pixel to line buffers>
      }
    }
    x += count;
  }
  if (x != tga.width)
    CkptRecover (CKPT_IO_ERROR);
```

24 and 32 bit targa pixels are just 3 or 4 bytes in BGR(A) order. We read that into an array that will be unpacked into the format we want.

```
    (19)  Read a targa pixel element into 'bgra'

  (*strm->readBytes) (strm, bgra, (alpha ? 4 : 3));
```

Once we have read a pixel we can store it to the accumulating output row by sticking the rgb and optional alpha into their buffers.

```
    (20)  Store 'bgra' pixel to line buffers

  *rgbBuf++ = bgra[2];
  *rgbBuf++ = bgra[1];
  *rgbBuf++ = bgra[0];
  if (alpha)
    *alphaBuf++ = bgra[3];
```

1.15 s3.2

```
    (21)  Targa types
  . . .
  typedef struct st_TargaSave {
    WriteStrmID      strm;
    Monitor         *mon;
    int              width, height;
    int              alpha, result;
  } TargaSave;
```

The targa saver sets up a protocol of the RGB24 type and requests a send from the source. Since the protocol callbacks have to return result codes, the ckpt mechanism is more of a hinderance here.

```
    (22)  Targa functions
  . . .
    int
  TargaSaver (
    long                     version,
    GlobalFunc              *global,
    ImSaverLocal            *local,
    void                    *servData)
  {
    ImageProtocol            prot;
    TargaSave                tga;

    if (version != 1)
      return AFUNC_BADVERSION;

    if (local->type != IMG_RGB24) {
      local->result = IPSTAT_FAILED;
      return AFUNC_OK;
    }

    tga.strm = StrmWriteOpen (local->filename);
    if (!tga.strm) {
      local->result = IPSTAT_BADFILE;
      return AFUNC_OK;
    }

    tga.result = IPSTAT_OK;
    tga.mon = local->monitor;

    prot.type = IMG_RGB24;
    prot.color.priv_data = &tga;
    prot.color.setSize = Targa_SetSize;
    prot.color.sendLine = Targa_SendLine;
    prot.color.done = Targa_Done;

    (*local->sendData) (local->priv_data, &prot, IMGF_ALPHA);

    StrmWriteClose (tga.strm);
    local->result = tga.result;
    return AFUNC_OK;
  }
```

The set size callback will just record the size and alpha status in the save info and write the header. The header is mostly zero expcpt for a few bytes with special values and the size as reversed byte order words.

```
    (23)  Targa utilities
  . . .
    XCALL_(static void)
  Targa_SetSize (
    TargaSave               *tga,
    int                      w,
    int                      h,
    int                      flags)
  {
```

```
    unsigned char            hdr[12];
    short                    size[2];
    int                      fail;

    if (!CkptCapture (fail)) {
      tga->result = IPSTAT_FAILED;
      return;
    }

    tga->width = w;
    tga->height = h;
    tga->alpha = ((flags & IMGF_ALPHA) != 0);

    memset (hdr, 0, 12);
    hdr[2] = 2;
    (*tga->strm->writeBytes) (tga->strm, hdr, 12);

    size[0] = w;
    size[1] = h;
    (*tga->strm->writeIWords) (tga->strm, size, 2);

    hdr[0] = (tga->alpha ? 32 : 24);
    hdr[1] = 0x20;
    (*tga->strm->writeBytes) (tga->strm, hdr, 2);

    if (tga->mon)
      MON_INIT (tga->mon, tga->height);

    CkptEnd ();
  }
  . . .
```

Writing a line is really easy. The pixel loop just unwraps the rgb and optional alpha data into targa pixel format and writes it. Write errors will return an error code, but nothing else.

```
    (24)  Targa utilities
  . . .
    static int
  Targa_SendLine (
    TargaSave               *tga,
    int                      line,
    const ImageValue        *data,
    const ImageValue        *adata)
  {
    unsigned char            bgra[4];
    int                      i, plen;
    int                      fail;

    if (tga->result != IPSTAT_OK)
      return -1;

    if (!CkptCapture (fail)) {
      if (fail == CKPT_ABORT)
        tga->result = IPSTAT_ABORT;
      else
        tga->result = IPSTAT_FAILED;
```

```
      return -1;
    }

    plen = (tga->alpha ? 4 : 3);

    for (i = 0; i < tga->width; i++) {
      bgra[2] = *data++;
      bgra[1] = *data++;
      bgra[0] = *data++;
      if (tga->alpha)
        bgra[3] = *adata++;

      (*tga->strm->writeBytes) (tga->strm, bgra, plen);
    }

    if (tga->mon && MON_STEP (tga->mon))
      CkptRecover (CKPT_ABORT);

    CkptEnd ();
    return 0;
  }
  . . .
```

The 'done' callback completes the monitor transaction and returns the aggregate error status.

```
    (25)  Targa utilities
  . . .
    static int
  Targa_Done (
    TargaSave               *tga,
    int                      error)
  {
    if (error)
      tga->result = IPSTAT_FAILED;

    if (tga->mon)
      MON_DONE (tga->mon);

    return (tga->result != IPSTAT_OK);
  }
```

1.16 s3.3

```
    (26)  Targa Image server

  #include <image.h>
  #include <strmu.h>
  #include <splug.h>
  #include <std.h>

  <Targa types>
  <Targa utilities>
  <Targa functions>
```

```
ServerRecord            ServerDesc[] = {
  { "ImageSaver",   "Targa",      TargaSaver },
  { "ImageLoader",  "Targa",      TargaLoader },
  { NULL }
};
```