TPNGImage 1.4
Gustavo Daud

Help file version 1.13

Introduction to TPNGImage 1.4

You're now reading TPNGImage* documentation, this version intends to replace the previous version, 1.2. Improvements in this new version includes:

- The new unit may or may not use the units SysUtils, Classes and Graphics, which will greatly reduce the size of the final executable. Read more about this feature here.
- CRC checking will now be fully performed.
- Some bugs when reading interlaced images are now fixed.
- Error on broken images are now better handled using new exception classes.
- The images may be saved using interlaced mode also.
- Transparency information won't be discarted after the image is loaded any more.
- Most of the images are decoded much faster now.
- The images will be better encoded using fresh new algorithms.
- IMPORTANT! Now transparency information is used to display images.

Most of the settings may be changed in the pngimage.pas unit by changing define triggers. Read more about define triggers here.

Using the component...

The component by default self integrates to TPicture class when included in the main unit uses clause.
All you have to is to copy all files to directory acessible in the search path and include pngimage to the uses.

** Note:* The name for the product continues to be TPNGImage but the real component now is called TPNGObject to avoid conflicts

This is the main object for the component. By default it's derived from *TGraphic* in order to integrates with Delphi and to be able to be used with TPicture, TImage and many others. To gain access to when using *TImage* or *TPicture*, typecast the TImage.Picture.Graphic or TPicture.Graphic to TPNGObject. The object provides access to some interesting and important features.

Provides pointer to direct access to alpha information

property AlphaScanline[**const** Index: Integer]: pByteArray;

Description
Some kind of *Portable Network Images* also provides transparency information. This allows the images to be draw on the screen with transparent parts by blending the foreground against the background using the value provided. Scanline property provides direct access to image contents and AlphaScanline provides direct access to the image transparency.

It's important to know that only when ColorType is COLOR_RGBALPHA or COLOR_GRAYSCALEALPHA *AlphaScanline* is valid. For other color types, this property will return **nil**.

The data in the pointer received will be always an array of bytes with the same size as the image width meaning that each byte represents the transparency for the correspondent position starting from 0 (totally transparent) to 255 (opaque).

Returns pointer to a object containg the list with all chunks.

property Chunks: TPNGList;

Description
This is the property which allows accessing all individuals chunks inside a *Portable Network Graphics* image.

Compression level when saving the image.

type TCompressionLevel = 0..9;
property CompressionLevel: TCompressionLevel;

Description
To save images, currently *Portable Network Graphics* uses a compression technique called ZLIB. ZLIB allows setting the compression level when compressing images, this allows smaller data but with some speed lost. Set this property to a higher value to compress better and to a low value to compress fastest.

Returns if the image is empty.

property Empty: Boolean;

Description
Returns if the image contains any data or not.

Returns the filters to use when saving an image.

type TFilter = (pfNone, pfSub, pfUp, pfAverage, pfPaeth);
type TFilters = **set of** TFilter;
property Filters: TFilters;

Description
When saving the image, Portable Network Graphics allows to use different filters to reduce the final image size. To compress the best as possible, the component tests all the selected filters for each image line to detect the best. This property allows to set which filters to test, meaning that if you select all the options, the image will get smallest as it can be but it will compress five times slower than if you had choosen only one item.

Comments
It is always recommended to set this property to test all the filters, but for speed purposes you might also choose only one filter and get full speed when saving.

Returns a pointer to the TChunkIHDR chunk.

property Header: TChunkIHDR;

Description
This property will return a pointer to the TChunkIHDR which contains the image information such as bit depth, color type, interlacing method, compression scheme and more.

Returns height for the current image.

property Height: Integer;

Description
This is a read-only property which will return the current image height. This property can not be changed. It's recommended instead to assign from another TBitmap when you want to change the image (or when you don't need to change size accessing Scanline property).

Returns the interlacing method to use when saving the image.

type TInterlaceMethod = (imNone, imAdam7);
property InterlaceMethod: TInterlaceMethod;

Description
This property returns/sets the interlace method to use when saving an image. Currently there are only two: none and Adam 7. Also, after an image is loaded, this property receives the interlace method used by this image. Read bout interlacing.

Maximum size allowed for each IDAT chunk.

property MaxIdatSize: Cardinal;

Description
This property allows the set the maximum size for each IDAT chunk which contains the image data. *Portable Network Graphics* allows multiple idat chunks (when one is followed by the other) to reduce the memory allocated to save images. Currently the minimum size allowed is 65535.

Sets bit transparent color for the png image.

property TransparentColor: TColor;
property TransparentColor: ColorRef;

Description
Use this property to change the bit transparency color for png images.

Note
Setting bit transparent color is not allowed for images already containing alpha information for each bit. Check header property ColorType (COLOR_RGBALPHA and COLOR_GRAYSCALEALPHA are not allowed).

Returns the transparency mode used by this png image.

type TPNGTransparencyMode = (ptmNone, ptmBit, ptmTranslucid);
property TransparencyMode: TPNGTransparencyMode;

Description
There are currently three transparency modes for png: none when all the image is opaque, bit when one color will be fully transparent and translucid when all colors may have transparent values. This property is mainly to mantain compability with Windows Delphi TBitmap since it supports only None and Bit (Translucid data is lost when assigning to a TBitmap).

Provides memory pointer to have direct access to the png contents

property Scanline[**const** Index: Integer]: Pointer;

Description
Like delphi *TBitmap*, TPNGObject now also supports direct access to the image contents (include to alpha information using AlphaScanline). This property is intended to be used by experienced graphics programmers. Also the contents depends on the current color type and bit depth values (See header property) as the table shows bellow:

Image kind	**Number of bits for each pixel**	**Recommended typecast to access the scanline data**
COLOR_GRAYSCALE - *Each pixel is intensity from 0 to 2^BitDepth - 1* COLOR_PALETTE - *Each pixel is a index to the palette table*		
Bitdepth = 1	1 bit	***pByteArray* (windows.pas)** (Except for 8 and 16, bit manangement algorithms should be used since there's more than one pixel per byte)
Bitdepth = 2	4 bits (not 2)	
Bitdepth = 4	4 bits	
Bitdepth = 8	1 byte	
Bitdepth = 16 (Grayscale)	1 byte	
COLOR_RGB - *Each pixel contains values for Red, Green, Blue intensities* COLOR_RGBALPHA - *Same as RGB but followed by an Alpha Value* COLOR_GRAYSCALEALPHA - *Same as GRAYSCALE but with alpha value*		
	1 byte per	

Bitdepth = 8	sample	*TRGBLine = array[word] of TRGBTriple;* ***pRGBLine = ^TRGBLine;***
Bitdepth = 16	1 byte per sample	

Returns width for the current image.

property Width: Integer;

Description
This is a read-only property which will return the current image width. This property can not be changed. It's recommended instead to assign from another TBitmap when you want to change the image (or when you don't need to change size accessing Scanline property).

Returns/sets arbirtuary pixels in the current png image.

property Pixels[const X, Y: Integer]: TColor;

Description
This property automates reading and setting pixels in the current png image by automatically reading/changing scanline property for all the different pixel formats. For more details read the Direct Access to Pixels article.

Add a tEXt chunk to the PNG image.

procedure AddtEXt(**const** Keyword, Text: String);

Description
AddtEXt is an easy way to add a new tEXt chunk (TChunktEXt) containing additional textual information.

Assigns contents from another bitmap or png object.

procedure Assign(Source: TPersistent);

Description
Use assign method to assign contents from a *TBitmap* object or from another *TPNGObject*. When assigning from a TBitmap, the component will set the current Portable Network Graphics image to use the same bit depth and color type as the bitmap.

Assigns contents from a windows bitmap handle.

procedure AssignHandle(Handle: HBitmap);

Description
Use AssignHandle to copy image contents from a windows bitmap handle into the current image.
Note: AssignHandle does not owns the handle in the parameter. The program is responsible to destroy it
using API DeleteObject.

Generates partial transparency information for the current image

procedure CreateAlpha;

Description
Use CreateAlpha to convert the current image into a partial transparency image.

When the current image color type is:

- COLOR_RGB, it is transformed into COLOR_RGBALPHA and AlphaScanline becomes valid.
- COLOR_GRAYSCALE is transformed into COLOR_GRAYSCALEALPHA and also AlphaScanline becomes valid.
- COLOR_PALETTE, the tRNS chunk is created containg alpha information for the current image.

Draws the current image into a windows device context (hdc).

procedure Draw(Canvas: TCanvas; **const** Rect: TRect);
procedure Draw(Canvas: HDC; **const** Rect: TRect);
the above only with objectpascal trigger set off

Description
This methods draws the current Portable Network Graphics image into a windows device context. Rect is the area where the image should be painted and Canvas is the canvas object.
Note
When the rect area is larger than the current image the image will be stretched only when it's not partial transparent.

Raises an error.

procedure LoadError(ExceptionClass: ExceptClass; Text: String);

Description
The component uses this method every time it needs to raise an error. ExceptionClass is the class for the error exception and text is the text to be displayed.

Loads the image from a file.

procedure LoadFromFile(**const** Filename: String);

Description
LoadFromFile method will load the filename into the current png object. It's recommended to use **try except end** operators to handle the different errors that might happen.

Loads a png file from a resource using a resource id.

procedure LoadFromResourceID(Instance: HInst; ResID: Integer);

Description
Use LoadFromResourceName to load a *Portable Network Graphics* image from a file resource. When you have the resource name, use LoadFromResourceName instead.

Loads a png file from a resource using a resource name.

procedure LoadFromResourceName(Instance: HInst; **const** Name: String);

Description
Use LoadFromResourceName to load a *Portable Network Graphics* image from a file resource. When you have the resource ID, use LoadFromResourceID instead.

Loads the current image from a TStream descendent.

procedure LoadFromStream(Stream: TStream);

Description
This method uses the data from the Stream object on the parameter to load a Portable Network Graphics image. This method might be useful to read data from a TStream that handles different sources of data such as registry, resources or even from internet.

Removes bit transparency from the image.

procedure RemoveTransparency();

Description
This method remove any bit transparency from the current image. Note: It doesn't remove partial transparency for RGBA images.

Saves the current image into a file.

procedure SaveToFile(**const** Filename: String);

Description
This method saves the current image into the file specified by filename parameter. The component will use Filters and MaxIdatSize parameters when saving. Also this component keep all the chunks from the loaded image, change the necessary and save everything.

Saves the current image to a TStream descendent.

procedure SaveToStream(Stream: TStream);

Description
This method saves the current loaded image using a TStream descendent. The component saves all the chunks when a image is loaded, change the necessary and use then to save the file again.

This object is returned by chunks property from TPNGObject. TPNGList provides access to all the chunks inside the current image.

Using TPNGList

Count property returns the number of items, Item[Index: Cardinal] property returns a TChunk using the index position (from 0 to Count - 1). Check the name from the returned TChunk using the name property and typecast using the appropriate class. For instance, if name property returns 'IHDR', typecast the returned TChunk as TChunkIHDR.

Returns the number of items in the list.

property Count: Cardinal;

Description
This property returns the number of items inside the chunk list.

Returns an item from the list.

property Item[Index: Cardinal]: TChunk;

Description
Use Item to return any chunk from the current Portable Network Graphics image. Index parameter is a position from 0 to Count - 1. The returned value will be the corresponding TChunk object. This should be typecasted to a descendent from TChunk to get it's full properties.

Returns the first item from the chunk with the same class.

type TChunkClass = **class of** TChunk;
property ItemFromClass[ChunkClass: TChunkClass]: TChunk;

Description
Use ItemFromClass property to search and return the first item in the list using the same class from the parameter. For instance, ChunkClass parameter might be TChunkIDAT.

Add a new item using the class from parameter.

type TChunkClass = **class of** TChunk;
procedure Add(ChunkClass: TChunkClass): TChunk;

Description
Add method will create a new item using the ChunkClass from the parameter and return a pointer to it. The Add method will select the most appropriate position to add depending on the chunk class.
Note
It's not allowed to add a second critical chunk such as TChunkIHDR. Calling Add using these on the parameters will return an error.

This class is responsible to handle all the chunks inside a Portable Network Graphics image. This means that all chunks shares this class as ancestor class to handle the data.

You should also typecast this class to get each chunk type properties when you know the chunk type. To check the chunk type read the name property.

Returns a pointer to the current chunk data.

property Data: Pointer;

Description
Returns a pointer to the chunk data. Use DataSize property to know the data length.

Returns the chunk data size.

property DataSize: Cardinal;

Description
This property should be used with the Data property. The property returns the size for the data returned.

Returns a pointer to the TChunkIHDR.

property Header: TChunkIHDR;

Description
This property returns a pointer to the *TChunkIHDR* chunk which should be always the first chunk.

Returns this chunk position from the list.

property Index: Integer;

Description
This property returns the current TChunk position inside the TPNGList. The position can be between 0 and TPNGList.Count - 1.

Returns the chunk name.

property Name: String;

Description
This property returns the current chunk name. Use the value returned from this property to typecast the TChunk to the right class. For instance, if Name property returns *IHDR*, typecast *TChunkIHDR(TChunk)*.

Returns owner.

property Owner: TPNGObject;

Description
Returns a pointer to the TPNGObject owner.

Assigns from another chunk contents.

procedure Assign(Source: TChunk);

Description
Use this method to assign the contents from another TChunk. When creating new chunk classes it's recommended to override this method to copy extra properties.

Create a new form and insert an edit box, a listbox, a memo and a button. The edit box is supposed to receive the file name, the listbox will contain all the keywords for the textual chunks. The memo will contain the text for the selected keyword in the listbox. And finally the button will load the file and fill the listbox. Use the code bellow:

```
uses
  Forms, pngimage, StdCtrls, Classes, Controls;

TForm1 = class(TForm)
  Button1: TButton;
  ListBox1: TListBox;
  Memo1: TMemo;
  Edit1: TEdit;
  procedure Button1Click(Sender: TObject);
  procedure ListBox1Click(Sender: TObject);
public
  png: TPngObject;
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
end;

{Form being created, create the png object}
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  png := tpngobject.create;
end;

{Form being destroyed, destroy the png object}
desstructor TForm1.Destroy;
begin
  inherited Destroy;
  png.free;
end;

{User clicked on the button, load the file and fill list}
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  try
    {Load the png file into the object}
    png.LoadFromFile(Edit1.Text);
    {Clear the listbox}
    listbox1.items.clear;
    {Searches for all the chunks using the type TChunktEXt}
    {add these to the listbox and a pointer to the chunk}
    {Note that all textual chunks are descendent from TChunktEXt}
    for i := 0 to png.chunks.count - 1 do
      if png.chunks.item[i] is TChunktEXt then
        listbox1.Items.AddObject(TChunktEXt(png.chunks.item[i]).keyword, png.chunks.item[i]);
  except
    {In case the image could not be loaded, show error}
    showmessage('The file could not be loaded.');
  end;
end;
```

Called when the chunk should load data.

function LoadFromStream(Stream: TStream; **const** ChunkName: TChunkName; Size: Integer): Boolean;

Description
This method is called to load the chunk from a stream using the property Stream. ChunkName is the name of the chunk and size is the size of the data. After this methods reads the data it should also read and check the crc (network ordered longint, 4 bytes).

To avoid reading the data and calculating the crc, call inherited method to let the ancestor do the work. Finally use Data property to get information.

The method should return true or false if it sucessfully readed the data.

Resizes the current chunk data.

procedure ResizeData(**const** NewSize: Cardinal);

Description
This method is used to resize the chunk data returned by data property.

Called to save the current chunk data into a stream.

function SaveToStream(Stream: TStream): Boolean;

Description
This method should write the entire chunk into the stream. The first part is the chunk length which is a network ordered cardinal, followed by the chunk name which is a 4 byte string. Then it is followed by the actual data and then a network ordered cardinal with the crc for the chunk name and for the data.

A easy way to handle is changing the data using Data and ResizeData and then calling inherited SaveToStream. This way will write everything and calculate the crc.

This must be the last chunk in a Portable Network Graphics image. This chunk indicates that the image has reached the end.
Handles the IEND chunk type.

This must be the first chunk in a *Portable Network Graphics* image. This is a important chunk indicating the image type, color type, compression method and other information. Internally it's responsible for allocating and preparing the data to hold the image.
Handles the IHDR chunk type.

Returns the current image bit depth.

property BitDepth: Byte;

Description
Bitdepth is the number of bytes for each sample. It's not recommended to change this property since it won't realloc data. The possible bit depths for the different color types (ColorType property) are:

Color type	**Allowed Bit Depths**	**Interpretation**
COLOR_GRAYSCALE	1,2,4,8,16	Each pixel is a grayscale sample.
COLOR_RGB	8,16	Each pixel is an R,G,B triple.
COLOR_PALETTE	1,2,4,8	Each pixel is a palette index; a PLTE chunk must appear.
COLOR_GRAYSCALEALPHA	8,16	Each pixel is a grayscale sample, followed by an alpha sample.
COLOR_RGBALPHA	8,16	Each pixel is an R,G,B triple, followed by an alpha sample.

Returns the current image color type.

property ColorType: Byte;

Description
Color type is the description for each pixel in the image. It's not recommended to change this property since it won't realloc data.

Color types	**Description**
COLOR_PALETTE	Each pixel is a palette index; a PLTE chunk must appear
COLOR_GRAYSCALE	Each pixel is a grayscale sample.
COLOR_RGB	Each pixel is an R,G,B triple.
COLOR_GRAYSCALEALPHA	Each pixel is a grayscale sample, followed by an alpha sample.
COLOR_RGBALPHA	Each pixel is an R,G,B triple, followed by an alpha sample.

Compression method for the data.

property ColorType: Byte;

Description
This property indicates the compression algorithm to compress the data. Currently the only possible value is 0, deflate/inflate. Changing this property is not recommended since it won't change anything else.

Filter set to use with the image.

property FilterMethod: Byte;

Description
This property defines the current filter set used by the current image. Currently only 0 is defined which is the none/sub/up/average/paeth set. Changing this property is not recommended.

Returns the current image height.

property Height: Cardinal;

Description
This property holds the height readed from the current image. Changing this property is not recommended since it won't reallocate data and may cause errors when saving. Changing image should be done by assigning a tbitmap using *TPngObject* assign.

Method to use to encode image.

property InterlaceMethod: Byte;

Description
Interlace method defines the way the image is compressed. Currently two methods are definied: 0 which is none and 1, Adam 7. Read more about interlacing.

Returns the current image width.

property Width: Cardinal;

Description
This property holds the width readed from the current image. Changing this property is not recommended since it won't reallocate data and may cause errors when saving. Changing image should be done by assigning a tbitmap using *TPngObject* assign.

PNG images can specify, via the gAMA chunk, the power function relating the desired display output with the image samples. Display programs are strongly encouraged to use this information, plus information about the display system they are using, to present the image to the viewer in a way that reproduces what the image's original author saw as closely as possible.
The name for this chunk is gAMA.

Returns the current gamma chunk value.

property Gamma: Cardinal;

Description
Contains gamma value. The value is encoded as a 4-byte unsigned integer, representing gamma times 100000. For example, a gamma of 1/2.2 would be stored as 45455. Changing this value won't update the image.

Contains the palette values for the current image. When ColorType from TChunkIHDR is *COLOR_PALETTE*, this chunk is required. When it is *COLOR_RGB* or *COLOR_RGBALPHA* it is optional to provide a sugested palette to which the truecolor image could be quantized.

Returns a palette item value.

```
type TRGBQUAD = packed record
  rgbBlue: Byte;
  rgbGreen: Byte;
  rgbRed: Byte;
  rgbReserved: Byte;
end;
property Item[Index: Byte]: TRGBQuad;
```

Description
This property is read-only and will return a palette item. The index can be from 0 to count - 1.

Returns number of palette items.

property Count: Integer;

Description
This property returns the number of items in the palette.

The tRNS chunk specifies that the image uses simple transparency: either alpha values associated with palette entries (for indexed-color images) or a single transparent color (for grayscale and truecolor images). Although simple transparency is not as elegant as the full alpha channel, it requires less storage space and is sufficient for many common cases.

Bit transparent color for the png image.

property TransparentColor: ColorRef;

Description
This property sets/returns the transparency color for png images when using bit transparency mode (only one color is fully transparent).

Returns transparency information for each palette item.

var PaletteValues: Array[Byte] **of** Byte;

Description
This variable contains the transparency information for each item in the palette. 0 is fully transparent and 255 is opaque. This variable is only valid when color type from TChunkIHDR is *COLOR_PALETTE*.

Contains the actual compressed and unfiltered image data. This chunk will be used to get the data into the image. Use Scanline and AlphaScanline from TPNGObject to have access to the decoded image data.

The tIME chunk gives the time of the last image modification (*not* the time of initial image creation). Universal Time (UTC, also called GMT) should be specified rather than local time.

Day for the last modification date.

property Day: Byte;

Description
Day should contain the day for the image last modification. The value for the first day in the month is 1 and for the last 31.

Hour for the last modification date.

property Hour: Byte;

Description
Hour should contain the hour for the image last modification. First hour is 0 and last is 23.

Minute for the last modification date.

property Minute: Byte;

Description
Minute should contain the minute for the image last modification. First minute is 0 and last is 59.

Month for the last modification date.

property Month: Byte;

Description
Month should contain the month for the image last modification. The value for the first month is 1 and for the last 12.

Second for the last modification date.

property Second: Byte;

Description
Second should contain the second for the image last modification. First second is 0 and last is 60. 60, for leap seconds; not 61, a common error

Year for the last modification date.

property Year: Word;

Description
Year should contain the year for the image last modification. The year should be complete, for instance 2002 not 02.

Textual information that the encoder wishes to record with the image can be stored in tEXt chunks. Each tEXt chunk contains a keyword (see above) and a text string.

Tile for the text information.

property Keyword: String;

Description
Change/read this property to change the keyword for the text property.

Text for the text chunk.

property Text: String;

Description
Change/Read this property to get the text information for this chunk.

Inverts an integer bytes.

function ByteSwap(**const** a: Integer): Integer;

Description
Since all the 4 bytes integers from *Portable Network Graphics* data must encoded using network order, this method is intended to adjust integers from/to delphi form.

Registers this chunk class with TPNGObject.

type TChunkClass = **class of** TChunk;
procedure RegisterChunk(ChunkClass: TChunkClass);

Description
Use RegisterChunk method to register a new class using TChunk as it's ancestor. The method will use the last four letters from the chunk name as the chunk name (Or you may also set fName variable on the constructor). When the component reads a chunk using the same name as a registered chunk it will use LoadFromStream method to load the data and SaveToStream when saving.

If you intend to create new chunk classes you must call this method before loading or saving image using TPngObject. See example 2 to see an example registering a new chunk class.

Method to allow calculating crc from data.

```
type TByteArray = array[word] of Byte;
type pByteArray = ^TByteArray;
procedure update_crc(crc: Cardinal; buf: pByteArray; len: Integer): Cardinal;
```

Description
This method is used to calculte crc values from data. CRC is the value resulting for the previous calls for update_crc. Buf is a pointer to the data to calculate and len is the size for the data in Buf. When you haven't any previous crc to use with CRC parameter, set it as $FFFFFFFF. When you finished calculating the value xor the resulting crc with $FFFFFFFF.

An example:

```
const
  DATA: ARRAY[0..3] of Char = ('C', 'R', 'C', 'C')
var
  crcvalue: Cardinal;
begin
  crcvalue := update_crc($FFFFFFFF, @Data[0], 1);
  crcvalue := update_crc(crcvalue, @Data[1], 1);
  crcvalue := update_crc(crcvalue, @Data[2], 1);
  crcvalue := update_crc(crcvalue, @Data[3], 1) xor $FFFFFFFF;
end;
```

is the same as:

```
const
  DATA: ARRAY[0..3] of Char = ('C', 'R', 'C', 'C')
var
  crcvalue: Cardinal;
begin
  crcvalue := update_crc($FFFFFFFF, @Data[0], 4) xor $FFFFFFFF;
end;
```

PNG is not just a replacement for *compuserve GIF*, it goes further than that. *Portable Network Graphics* is a new generation format, supporting partially transparency images. This means that pixels are blended with the background pixels when drawing, feature supported by this new *TPNGImage* version. Also this means that pre-calculations are made to blend the image pixel and the background pixel in order to partial transparency to happen.
The *TPNGImage 1.4* library also provides access to the transparency data in order to allow image editors to have access to this feature.

How does the transparency works in PNG ?

There are two different methods for storing transparency information, one for 24bits, 48bits and grayscale and the other for palette (8 bits or less). When working with 24bits or more, the transparency information is stored next to each pixel red, green and blue values, and then stored in a different memory space by TPNGImage. In the second way, all the transparency information is stored in a chunk called *tRNS* containing transparency information for each palette entry.

So, in order to have access to transparency data, you should verify the color mode being used. To do so, access the bitdepth property from the IHDR chunk, if the returns *COLOR_PALETTE* constant, you should check for the tRNS chunk, otherwise use the AlphaScanline array (from the main *TPNGObject*).

Transparency data

In booth method, transparency is a single byte (in tpngimage implementation), containing information for the associated pixel. This byte contains values between 0 and 255. 0 value means that the associated pixel is completly transparent (the background pixel is intact), and 255 value means that the background pixel is replaced by the image pixel. For instance, if the transparency contains a value like 128, the image pixel is blended with the background pixel and resulting pixel contains half of the background pixel plus half of the image pixel.

Detecting transparency mode

It is real simple to detect the transparency mode for the current image. Just check the BitDepth property from the IHDR chunk, accessed by bitdepth property from the header (direct access to the IHDR chunk) property:

```
if Header.BitDepth <> COLOR_PALETTE then
  ... First way
else
  ... Second way;
```

Accessing thru AlphaAscanline

The AlphaScanline property contains only a valid value when the color mode for the current image is either COLOR_RGBALPHA or COLOR_GRAYSCALEALPHA. You should check it before accessing it.

It's real simple to use this way. AlphaScanline is an indexed property that returns a pointer to a byte array containing transparency information for needed image line. So, it means that the return is a **pByteArray** which is type *^Array[Word] of Byte* for the line. Also the returned array contains one byte for each line pixel, so the array bounds is *0* to *ImageWidth - 1*

The example shows how to make the current image (don't work for COLOR_PALETTE mode) half transparent:

```
procedure MakeImageHalfTransparent(Obj: TPNGObject);
var i, j: integer;
begin
 //Add alpha channel in case it is RGB or GRAYSCALE
 if Obj.Header.ColorType in [COLOR_RGB, COLOR_GRAYSCALE] then
   Obj.CreateAlpha();

 //Set half transparent transparency, value 128 (256 / 2)
 if Obj.Header.ColorType in [COLOR_RGBALPHA, COLOR_GRAYSCALEALPHA] then
   FOR j := 0 TO Obj.Header.Height - 1 DO
     FOR i := 0 TO Obj.Header.Width - 1 DO
       Obj.AlphaScanline[j]^[i] := 128
end;
```

Accessing thru tRNS chunk

The tRNS chunk class contains transparency information when the color type is COLOR_PALETTE (It exists also for the other color types but only for bit transparency modes). It works different than AlphaScanline property since instead of providing transparency information for each pixel in the image, it contains transparency for each palette entry.

Acessing the tRNS chunk also is really easy using the ItemFromClass method from the Chunks property. Once you have a pointer to the TChunktRNS, the transparency information for each palette entry is acessed thru the PaletteValues property, which is an array of byte. This array has a fixed size of 256 entries (0 to 255), but the valid values are between 0 and (2 POWER Header.BitDepth - 1). 255 means that the palette entry is completly opaque and 0 completly transparent.

The example bellow makes the image half transparent when the color mode is COLOR_PALETTE:

```
procedure MakeHalfTransparent(Obj: TPNGObject);
var
  i: Integer;
  TRNS: TCHUNKtRNS;
begin
  //Creates tRNS chunk in case its not avaliable
  if (Obj.Header.ColorType = COLOR_PALETTE)  and (Obj.Chunks.ItemFromClass(TChunktRNS)
= nil) then
    Obj.CreateAlpha();
  //Gets pointer to the tRNS chunk
  TRNS := Obj.Chunks.ItemFromClass(TChunktRNS) as TChunktRNS;
  //Set transparency information
  if TRNS <> nil then
    with TRNS do
      for i := 0 to DataSize - 1 do
        PaletteValues[i] := 128
end;
```

Converting to partial transparency

The TPNGImage latest release provides a method to generate transparency data for the current image when its not avaliable, this method is CreateAlpha. It generates alpha information:

- When the image color type is COLOR_RGB or COLOR_GRAYSCALE it converts to COLOR_RGBALPHA and COLOR_GRAYSCALEALPHA and makes the AlphaScanline property avaliable.
- For COLOR_PALETTE color type, it generates a tRNS chunk which may be acessed using Chunks.ItemFromClass.

If the image already contains alpha information, the method don't do nothing, so its always safe to call it. Also, it initialize the transparency image so the image is fully opaque for booth COLOR_PALETTE and COLOR_RGB/COLOR_GRAYSCALE modes.

To access the transparency information when the image is COLOR_RGB/COLOR_GRAYSCALE use the first way, and for COLOR_PALETTE use the second way.

About chunks

A *Portable Networks Image* is made of several information packets called chunks. Some of these are necessary and essencial to allow the image to be displayed, others contain additional information as text or historiograms.

Inside a file, a chunk contains: a network ordered 4 bytes value containing the size for the chunk data; a 4 bytes string containing the chunk name; the data with the length specified before; and for the last also a network ordered 4 bytes unsigned integer containing the chunk crc for the chunk name and data.

```
ChunkLength: Cardinal;
ChunkName  : Array[0..3] of Char;
ChunkData  : Array[0..ChunkLength - 1] of byte;
ChunkCRC   : Cardinal;
```

The crc part is important to validate the chunk data, as when the image was saved. The crc is created using the ChunkName and ChunkData only. This storage method allows flexibility to the images, allowing decoders to ignore certain chunks when they don't reconize it.

About the chunk name

The chunk name will indicate the chunk content type so it can be decoded. The name should contain 4 ASCII uppercase or lowercase letters. Also the case is sensitive. There are some rules to tell which letter should be lowercased or uppercased.

Letter	**In case it is:**	
	Uppercase	**Lowercase**
1	The chunk is critical and must be know by the decoder.	It's a secondary chunk and might be ignored.
2	The chunk is part from the official PNG specification.	This is a private chunk with specific purposes.
3	Following PNG specification, this letter must be uppercased.	It's not definied yet when this letter should be lowercased.

4	Depends on the image contents to exist.	May be saved unchanged in case this image is not changed.

Critical chunks order

A valid PNG image must contain a IHDR, one or more sequencial IDAT chunks and in the end a IEND chunk. In case this image requires a palette, the PLTE chunk is also mandatory.

Name	**Multiples ok ?**	**Ordem do chunk**
IHDR	*No*	Must be always the first
PLTE	*No*	Before IDAT
IDAT	*Yes*	Multiple must be sequencial
IEND	*No*	Always the last

Accessing each chunk in the component

All the readed chunks are stored in allocated memory by the component. This component implements objects to handle with all the different chunk types and provides ways to read different properties from each chunk. The ancestor class to handle all the chunks is TChunk. Altough there are several differend classes with TChunk as it's ancestor to read specific information such as TChunkIHDR handling the IHDR chunk.

Use the property Chunks from TPNGObject to access all the stored chunks. TPNGObject.Chunks.Count retorna o total número de chunks e TPNGObject.Chunks.Item[i] returns a TChunk object from the position i (should be between 0 and Count - 1).

As you get the returned TChunk, use it's property Name to get the chunk type. You might also use the **is** operator followed by the class to test (for instance: TPNGObject.Chunks.Item[i] is TChunkIHDR).

Knowing the chunk name, and if there is a class to handle this chunk, you should access this class using typecast, for instance TChunkIHDR(TPNGObject.Chunks.Item[0]). Now you may access this object properties as specified in this help file.

Portable Network Graphics interlacing

As JPEG (Joint photographic experts group), Portable Network Graphics also supports image interlacing. This technique encodes the image in a way to allows the user preview the image faster as it is being transfered.

PNG's two-dimensional interlacing scheme is more complex to implement than GIF's line-wise interlacing. It also costs a little more in file size. However, it yields an initial image *eight times* faster than GIF (the first pass transmits only 1/64th of the pixels, compared to 1/8th for GIF). Although this initial image is coarse, it is useful in many situations. For example, if the image is a World Wide Web imagemap that the user has seen before, PNG's first pass is often enough to determine where to click. The PNG scheme also looks better than GIF's, because horizontal and vertical resolution never differ by more than a factor of two; this avoids the odd "stretched" look seen when interlaced GIFs are filled in by replicating scanlines. Preliminary results show that small text in an interlaced PNG image is typically readable about twice as fast as in an equivalent GIF, i.e., after PNG's fifth pass or 25% of the image data, instead of after GIF's third pass or 50%. This is again due to PNG's more balanced increase in resolution.

About the triggers

This version introduces new set of triggers to enable or disable some features.

Trigger name	When the trigger is	
	Being used	**Disabled**
ObjectPascal	SysUtils, Graphics and Classes are used by the unit.	Heavy units are not used but the TPNGObject is not registered as a TGraphic class
ErrorOnUnknownCritical	In case it decoder finds an unknown critical chunk, the application stops.	Unknown critical chunks are ignored by the decoder.
CheckCRC	CRC is checked for all the data readed.	The CRC is not checked (less security for more speed)
RegisterGraphic	Registers TPNGObject as a TGraphic	TPNGObject is not registered as TGraphic
SemiTransparentDraw	Images with partial transparency are draw agains the background	Semi transparency parts are ignored

Enabling/disabling triggers

All this triggers are set in the beginning of pngimage.pas unit. To set them, add (to use) or remove the lines {$DEFINE NameOfTheTrigger}.

Disable use of heavy object pascal units

One of the new features for this version is to disable the use of heavy object pascal units. This features only becomes interesting when the programmer is writting a pure windows api application. This features also makes easier to use the unit with other pascal compilers (not tested).

To enable/disable using heavy object pascal units add/remove (to disable) the line in the beginning of the unit pngimage.pas:
{$DEFINE ObjectPascal}

Translating the library

As the previous version, the new TPNGImage is translate ready to adapt to any language. To translate, edit the file **pnglang.pas** located in the same directory as *pngimage.pas* and feel free to change anything.

There are some triggers to make the translation easier, just comment the triggers for the languages you **don't want** and leave the trigger for the language you want. For instance, if you want English instead of Portuguese, change {$DEFINE Portuguese} to {.$DEFINE Portuguese} and then {.$DEFINE English} to {$DEFINE English}.

Network Order

All the values containing more than one byte, as written in Portable Network Graphics specification must use Network Order format, it means that the most significant byte must come first followed by the less significant. ([Byte more significant] [Byte less significant] for 2 bytes values or B3 B2 B1 B0 to 4 bytes values). The more important bit (value 127) from a byte is bit 7 and the less important (value 1) is bit number 0.

Cyclic redundancy check

Abbreviation of cyclic redundancy check, a common technique for detecting data transmission errors. A decoder reads the received data and compare it with the 4 byte 32 bits CRC the encoder calculated, which comes together with the data transmited. In case the values are different, the image has suffered data modification during the transmission.

TPNGImage 1.4
Gustavo Daud

Errors handling

The new TPNGImage introduces a few new classes to handle different errors. All this classes are descendent from **Exception** meaning that the error might be handled using the **try** operator from delphi. It's recommended to handle errors when using the methods LoadFromFile, SaveToFile, LoadFromStream and SaveToStream from TPNGObject.

The new classes introduced are:

Error class	**Description**
EPngError	Can be several errors, but it's never called when the image is being loaded or saved.
EPngUnexpectedEnd	The decoder found an invalid end of the file.
EPngInvalidCRC	One or more chunks contain an invalid crc identifier which is provided within each chunk
EPngInvalidIHDR	The size from the IHDR chunk is invalid
EPNGMissingMultipleIDAT	The IDAT chunk data has ended and it still misses image parts
EPNGZLIBError	ZLIB returns an error, common causes are low memory or invalid compressed data
EPNGInvalidPalette	The PLTE chunks contains an invalid number of entries
EPNGInvalidFileHeader	The Portable Network Graphics image contains an invalid file header.
EPNGIHDRNotFirst	The IHDR chunk is not the first chunk on the image but it must
EPNGNotExists	The file does not exists.
EPNGSizeExceeds	Either width or height is more than 65535 pixels.
EPNGMissingPalette	The image uses a color table but PLTE chunk was not found
EPNGUnknownCriticalChunk	The image contains an unknown critical chunk
EPNGUnknownCompression	The image uses an unknown compression method found on the IHDR chunk
	The image uses an unknown

EPNGUnknownInterlace	interlacing method found on the IHDR chunk
EPNGNoImageData	There is no IDAT chunk.

About filters

PNG allows the image data to be *filtered* before it is compressed. Filtering can improve the compressibility of the data. The filter step itself does not reduce the size of the data, it prepares the data to be much better compressed. All PNG filters are strictly lossless.

PNG defines several different filter algorithms, including "None" which indicates no filtering. The filter algorithm is specified for each scanline by a filter-type byte that precedes the filtered scanline in the precompression datastream. An intelligent encoder can switch filters from one scanline to the next. The method for choosing which filter to employ is up to the encoder.

For images of color type 3 (indexed color, COLOR_PALETTE), filter type None is usually the most effective. Note that color images with 256 or fewer colors should almost always be stored in indexed color format; truecolor format is likely to be much larger.
Filter type 0 is also recommended for images of bit depths less than 8. For low-bit-depth grayscale images, it may be a net win to expand the image to 8-bit representation and apply filtering, but this is rare.

This component allows to select the filter set to using the property Filters from TPNGObject.

Bit and partial transparency

In order to have more compability between Windows, TBitmap and TPNGObject features, TPNGObject introduces three different transparency modes:

- None - When there are no transparent areas in the image
- Bit - Each pixel may be full transparent or opaque against the background
- Partial - Pixels may be translucid (semi transparent)

TPNGObject provides a read-only property to obtain the transparency mode for the image, TransparentMode property. Also changing TransparentColor property changes the image to Bit transparency mode always, loose partial transparency information.

Direct access to pixels data

Accessing the pixels data directly is very useful when you need speed while manipulating the png data. For this purpose, the new version provides two properties to do so: Scanline and AlphaScanline.

Checking for the pixel format...

Before directly accessing the data (using typecasts) you must first verify which pixel format the current png is using. This information is stored in the TChunkIHDR chunk. This is returned using header property from the main TPNGObject.

TChunkIHDR provides the properties BitDepth and ColorType. Using the table typecasting table (described in this page) you may directly access the data.

There are two typecasts you should use, **pByteArray** (windows.pas) when pixels occupy 8 or less bits and **pRGBLine** otherwise.

```
type
  TRGBLine = array[word] of TRGBTriple;
  pRGBLine = ^TRGBLine;
```

The method bellow returns whenever you should use pRGBLine or pByteArray.

```
type
  TScanlineTypeReturn = (stByteArray, stRGBLine)
function GetScanlineType(const png: TPngObject):
  TScanlineTypeReturn
begin
  with png.Header do
  begin
    if ColorType in [COLOR_GRAYSCALE,
      COLOR_PALETTE] then
      Result := stByteArray
    else
      Result := stRGBLine
  end
end;
```

Using pRGBLine

The method bellow returns an arbituary pixel from the png object. **Note:** It does not performs any bounds checking; Valid values for X are (0, png.Width - 1) and for y (0, png.Height - 1)

```
function GetRGBLinePixel(const png: TPngObject;
  const X, Y: Integer): TColor;
```

```
begin
  with pRGBLine(png.Scanline[Y])^[X] do
    Result := RGB(rgbtRed, rgbtGreen, rgbtBlue)
end;
```

And the following sets a pixel value:

```
procedure SetRGBLinePixel(const png: TPngObject;
 const X, Y: Integer; Value: TColor);
begin
  with pRGBLine(png.Scanline[Y])^[X] do
  begin
    rgbtRed := GetRValue(Value);
    rgbtGreen := GetGValue(Value);
    rgbtBlue := GetBValue(Value)
  end
end;
```

Adapt the code above so it may fit to your needs.

Using pByteArray

This method is much more complicated than pRGBLine because, except for bit depths 8, allows more than one pixel for each byte (pixels never cross byte boundaries) requering bit manipulation to access the data. Also, for COLOR_PALETTE it must map the value to a palette entry because it is actually an index rather than the actual value (different from pRGBLine).

```
{Returns pixel for png using palette and grayscale}
function GetByteArrayPixel(const png: TPngObject;
  const X, Y: Integer): TColor;
var
  ByteData: Byte;
  DataDepth: Byte;
begin
  with png, Header do
  begin
    {Make sure the bitdepth is not greater than 8}
    DataDepth := BitDepth;
    if DataDepth > 8 then DataDepth := 8;
    {Obtains the byte containing this pixel}
    ByteData := pByteArray(png.Scanline[Y])^[X div
      (8 div DataDepth)];
    {Moves the bits we need to the right}
    ByteData := (ByteData shr ((8 - DataDepth) -
      (X mod (8 div DataDepth)) * DataDepth));
    {Discard the unwanted pixels}
    ByteData:= ByteData and ($FF shr (8 - DataDepth));

    {For palette mode map the palette entry and for
      grayscale convert and returns the intensity}
    case ColorType of
      COLOR_PALETTE:
```

```
    with TChunkPLTE(png.Chunks.ItemFromClass(TChunkPLTE)).Item[ByteData] do
      Result := rgb(GammaTable[rgbRed], GammaTable[rgbGreen],
        GammaTable[rgbBlue]);
   COLOR_GRAYSCALE:
   begin
     ByteData := GammaTable[ByteData * ((1 shl DataDepth) + 1)];
     Result := rgb(ByteData, ByteData, ByteData);
   end;
  end {case};
 end {with}
end;
```

And the following sets a pixel value:

```
{Sets a pixel for grayscale and palette pngs}
procedure SetByteArrayPixel(const png: TPngObject;
 const X, Y: Integer; const Value: TColor);
const
 ClearFlag: Array[1..8] of Integer = (1, 3, 0, 15, 0, 0, 0, $FF);
var
 ByteData: pByte;
 DataDepth: Byte;
 ValEntry: Byte;
begin
 with png.Header do
 begin
  {Map into a palette entry}
  ValEntry := GetNearestPaletteIndex(Png.Palette,
   ColorToRGB(Value));

  {16 bits grayscale extra bits are discarted}
  DataDepth := BitDepth;
  if DataDepth > 8 then DataDepth := 8;
  {Gets a pointer to the byte we intend to change}
  ByteData := @pByteArray(png.Scanline[Y])^[X div
   (8 div DataDepth)];
  {Clears the old pixel data}
  ByteData^ := ByteData^ and not (ClearFlag[DataDepth] shl
   ((8 - DataDepth) - (X mod (8 div DataDepth)) * DataDepth));

  {Setting the new pixel}
  ByteData^ := ByteData^ or (ValEntry shl ((8 - DataDepth) -
   (X mod (8 div DataDepth)) * DataDepth));
 end {with png.Header}
end;
```

Pixels property

The methods above are extracted from the Pixels[] property which might be used to set and get pixels. Although this property do all the dirty work, it is probably fast for time critical algorithms. It is recomended to change and adapt the code in order to fast change and access the data.

Converting from TBitmap to TPNGObject

TPNGObject provides flexibility by allowing to convert from a Bitmap file format to *Portable Network Graphics* format.
This is easily done like in the same way as any graphic class in delphi.
IMPORTANT: Always remember to add pngimage to the unit uses.

Converting from Windows bitmap file to PNG file
This method loads a bitmap and saves it using png format

```
procedure BitmapFileToPNG(const Source, Dest: String);
var
  Bitmap: TBitmap;
  PNG: TPNGObject;
begin
  Bitmap := TBitmap.Create;
  PNG := TPNGObject.Create;
  {In case something goes wrong, free booth Bitmap and PNG}
  try
    Bitmap.LoadFromFile(Source);
    PNG.Assign(Bitmap);    //Convert data into png
    PNG.SaveToFile(Dest);
  finally
    Bitmap.Free;
    PNG.Free;
  end
end;
```

Converting from PNG file to Windows bitmap file
The above inverse. Loads a png and saves into a bitmap

```
procedure PNGFileToBitmap(const Source, Dest: String);
var
  Bitmap: TBitmap;
  PNG: TPNGObject;
begin
  PNG := TPNGObject.Create;
  Bitmap := TBitmap.Create;
  {In case something goes wrong, free booth PNG and Bitmap}
  try
    PNG.LoadFromFile(Source);
    Bitmap.Assign(PNG);    //Convert data into bitmap
    Bitmap.SaveToFile(Dest);
  finally
    PNG.Free;
    Bitmap.Free;
  end
```

```
end;
```

Converting from TImage to PNG file
This method converts from TImage to PNG. It has full exception handling and allows converting from file formats other than TBitmap (since they allow assigning to a TBitmap)

```
procedure TImageToPNG(Source: TImage; const Dest: String);
var
  PNG: TPNGObject;
  BMP: TBitmap;
begin
  PNG := TPNGObject.Create;

  {In case something goes wrong, free PNG}
  try
    //If the TImage contains a TBitmap, just assign from it
    if Source.Picture.Graphic is TBitmap then
      PNG.Assign(TBitmap(Source.Picture.Graphic))    //Convert bitmap data into png
    else begin

      //Otherwise try to assign first to a TBimap
      BMP := TBitmap.Create;
      try
        BMP.Assign(Source.Picture.Graphic);
        PNG.Assign(BMP);
      finally
        BMP.Free;
      end;

    end;

    //Save to PNG format
    PNG.SaveToFile(Dest);
  finally
    PNG.Free;
  end
end;
```

Create a new form and insert an edit box, a listbox, a memo and a button. The edit box is supposed to receive the file name, the listbox will contain all the keywords for the textual chunks. The memo will contain the text for the selected keyword in the listbox. And finally the button will load the file and fill the listbox. Use the code bellow:

```
uses
  Forms, pngimage, StdCtrls, Classes, Controls,
    Dialogs;

type
  TForm1 = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    Memo1: TMemo;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure ListBox1Click(Sender: TObject);
  public
    png: TPngObject;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

{$R *.DFM}

var
  Form1: TForm1;

implementation

{Form being created, create the png object}
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  png := tpngobject.create;
end;

{Form being destroyed, destroy the png object}
destructor TForm1.Destroy;
begin
  inherited Destroy;
  png.free;
end;

{User clicked on the button, load the file and fill list}
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  try
    {Load the png file into the object}
    png.LoadFromFile(Edit1.Text);
    {Clear the listbox}
    listbox1.items.clear;
    {Searches for all the chunks using the type TChunktEXt}
    {add these to the listbox and a pointer to the chunk}
    {Note that all textual chunks are descendent from
    {TChunktEXt}
    for i := 0 to png.chunks.count - 1 do
      if png.chunks.item[i] is TChunktEXt then
        listbox1.Items.AddObject(TChunktEXt(
          png.chunks.item[i]).keyword, png.chunks.item[i]);
  except
    {In case the image could not be loaded, show error}
    showmessage('The file could not be loaded.');
  end;
end;

{User selected an item, show text on the memo}
procedure TForm1.Listbox1Click(Sender: TObject);
begin
  if listbox1.itemindex <> -1 then
    memo1.text := TChunktEXt(
      Listbox1.Items.Objects[Listbox1.itemindex]).Text;
end;
```

One of the powerful features from the component is the support to additional TChunk descendents. To do so, you have to create a new class descendent from TChunk. To read the data, there are two ways: if the chunk contains large amounts of data, override LoadFromStream and read data, crc and check if the crc is valid. If the data is not too large you might use the content from Data and DataSize property.

To save it, either override SaveToStream and write data manually, override SaveToStream, modify data property and call inherited SaveToStream. Also there other ways as modifying the data property directly when the user reads/writes a property.
Other important method to override is the Assign method to copy the chunk custom properties.

The essencial part is to register the chunk using RegisterChunk(ChunkClass: TChunkClass) from pngimage.pas.

The chunk bellow reads a text and shows a message box using the text.

```
type
  TChunkcUSt = class(TChunk)
  private
    fText: String;
  public
    function SaveToStream(Stream: TStream): Boolean; override;
    function LoadFromStream(Stream: TStream; const
      ChunkName: TChunkName; Size: Integer): Boolean; override;
    procedure Assign(Source: TChunk); override;
    property Text: String read fText write fText;
  end;

implementation

{Saving chunk to a stream}
function TChunkcUSt.SaveToStream(Stream: TStream): Boolean;
var
  ChunkLength, ChunkCRC: Cardinal;
begin
  {ChunkLength must be in network order}
  ChunkLength := ByteSwap(Length(fText));
  Stream.Write(ChunkLength, 4);
  {Writes chunk name}
  Stream.Write(fName[0], 4);
  ChunkCRC := update_crc($ffffffff, @fName[0], 4);
  {Writes data and finishes calculating crc}
  Stream.Write(fText[1], Length(fText));
  ChunkCRC := Byteswap(update_crc(ChunkCRC, @fText[1],
    Length(fText)) xor $ffffffff);
  {Writes crc}
  Stream.Write(ChunkCRC, 4);
  Result := TRUE;
end;

{Loading chunk from a stream}
function TChunkcUSt.LoadFromStream(Stream: TStream;
  const ChunkName: TChunkName; Size: Integer): Boolean;
var
  ReadCRC, ChunkCRC: Cardinal;
begin
  {Prepares text to hold}
  SetLength(fText, Size);
  {Reads data}
  Stream.Read(fText[1], Size);
  {Calculates crc for data readed}
  ChunkCRC := update_crc($ffffffff, @ChunkName[0], 4);
  ChunkCRC := Byteswap(update_crc(ChunkCRC, @fText[1],
    Size) xor $ffffffff);
  {Reads crc and verify}
  Stream.Read(ReadCRC, 4);

  {Check if crc is valid}
  Result := (ReadCRC = ChunkCRC);
  if not Result then
    Owner.LoadError(EPngInvalidCRC,
      EPngInvalidCRCText)
  {If it's valid, show text using a message box}
  else ShowMessage(fText);
end;

{Assigns contents from another chunk}
procedure TChunkcUSt.Assign(Source: TChunk);
begin
  fText := TChunkcUSt(Source.fText);
end;

initialization
  RegisterChunk(TChunkcUSt);
finalization
```

As a TGraphic descendent, TPngObject supports assigning from and to TBitmap. As an intermediary class, TBitmap talks to most (if not all) other formats such as metafiles, icons, jpgs among others.

The example bellow loads a jpg, a png and draws the png over the jpg. The png image may use alpha transparency normally.

```
{Draws a PNG over a JPG and saves again}
procedure PNGOverJPG(InJPG, InPNG: String; OutJPG: String);
var
  JPG: TJPEGImage;
  BMP: TBitmap;
  PNG: TPNGObject;
begin
  {Creates and loads the input images}
  JPG := TJPEGImage.Create;
  JPG.LoadFromFile(InJPG);
  BMP := TBitmap.Create;
  BMP.Assign(JPG);
  PNG := TPNGObject.Create;
  PNG.LoadFromFile(InPNG);
  {Draws over the bitmap (containing the JPG)}
  BMP.Canvas.Draw(0, 0, PNG);
  {Assigns back to the JPG}
  JPG.Assign(BMP);
  {Saves the JPG}
  JPG.SaveToFile(OutJPG);
  {Free the images}
  JPG.Free;
  BMP.Free;
  PNG.Free;
end;
```

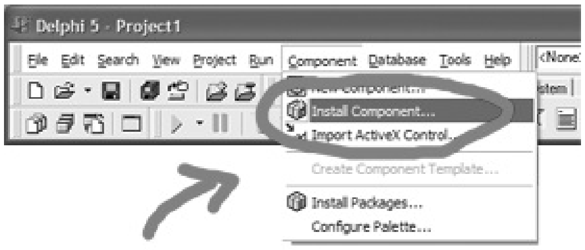
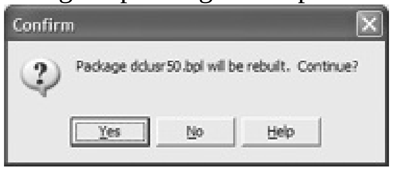
TPNGImage 1.4
Gustavo Daud

Complete list of features

- Chunks are organized in a clever way with an ancestor class to handle all the chunks and derived classes to handle the different kinds
- The chunks engine allows upgrades by registering new chunk types to include new features
- Full cyclic redundancy check for all the chunks
- Full access to all the data from the chunks
- Several derived classes from TChunk to obtain different information such as text keyword and more
- Gamma chunk is used to change the image gamma to be displayed as when it was encoded.
- No need for any external library, everything is compiled with your exe project
- Full comented source code and also well written.
- Can read all the images from the official PNG test suite (see showcase 2). new
- Can load interlaced and non interlaced images
- Ability to assign data from a tbitmap or bitmap handle
- The PNG images may also be saved
- Saving with interlace is now fully supported. new
- The encoder is optimized to create really small final images when saving.
- No chunk is lost when loading an image
- Portable Network Graphics partial transparent images are now supported new
- Trigger to disable use of Delphi VCL
- Full speed when loading and saving
- Allows to select the filters to test when saving images to create the smallest image as possible
- Improved error handling engine to allow user to detect easily all the errors. new
- Better chunk access system, now the engine selects the best place to put chunks when adding manually. new
- Provides now accessed to the image data and for the transparency alpha data. new
- Allows to select the compression level for ZLIB.
- Allows the set the maximum size for IDAT chunks (and then dividing the rest of the data in new chunks). new
- Complete help file using HTML help

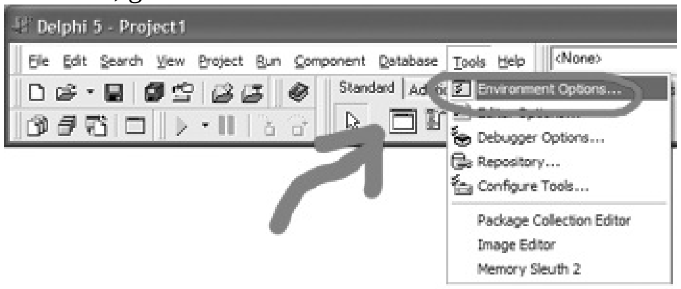
Installing the component

The component is installed as any other common delphi component.

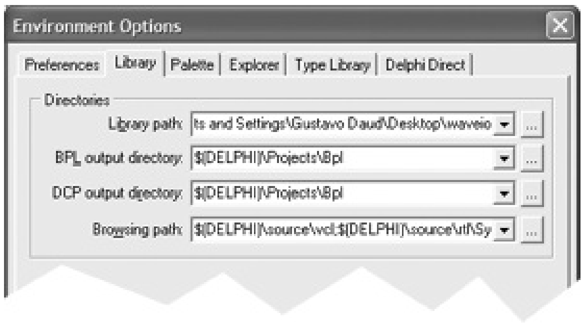
1. First of all, copy (unzip) all the files to any directory. The **obj** sub-directory is necessary to install the component.
2. Run a Borland Delphi instance
3. Go to the menu **Component** and select **Install Component** item as shown in the picture bellow:

Delphi 5 - Project1
File Edit Search View Project Run Component Database Tools Help
Install Component...
Import ActiveX Control...
Create Component Template...
Install Packages...
Configure Palette...
4. In the "Unit file name:" field, click on "Browse..." and point to the *pngimage.pas* file.
By default Delphi will install in the "Borland User Components" package, it might be changed using "Package file name" field or "Into new package" page.
Click on Ok
5. A confirmation dialog will be shown asking if you want to build the package (name might change depending on Delphi version). Click on "Yes".

Confirm
Package dclusr50.bpl will be rebuilt. Continue?

6. You have just installed PngImage, now close the package window (Don't forget to put "Yes" when it ask if you want to save the package).

You may also want to add the unit to the search path.

1. This time, go to the "Tools" menu and select the item "Enviroment Options".


2. Click on the "Library" page.



Library path: ts and Settings\Gustavo Daud\Desktop\waveio
BPL output directory: $(DELPHI)\Projects\Bpl
DCP output directory: $(DELPHI)\Projects\Bpl
Browsing path: $(DELPHI)\source\vcl;$(DELPHI)\source\rtl\Sy

3. On the field "Library path" **add** a ";" followed by the unit directory. For instance ";c:\png".
4. Click on the ok button.

TPNGImage 1.4
Gustavo Daud

Component license

The previous versions from this component were unclear about the license to use this component. Here it is:

1. This component should be distributed freely over the internet only when containing the exact same files from the original packaging.
2. Modified files may not be distributed. If you want to contribute with TPNGImage, send the enhancements to the author and if he implements your changes, you will be given the proper credit.
3. The component may be used in commercial projects but may NEVER be sold as source code.
4. Commercial graphics libraries are not allowed to use this component WITHOUT AUTHOR PRIOR AGREEMENT.
5. Credit for the author is required somewhere in the product documentation/or about box/etc for applications distributed over the internet.
6. Source code may be changed since it's not redistributed.

If are about to use the component in a major project which is going to be distributed over the internet, I'd love to know, so please send me an email telling me about.

TPNGImage 1.4
Gustavo Daud

Component author and website

Gustavo Huffenbacher Daud
Currently, contact me at: gubadaud@terra.com.br

http://pngdelphi.sourceforge.net

This is the best place to get the latest version. *TPNGImage* took me lots of hours programming, and it's free even for commercial projects (read license). above.