NINTENDO NITRO-SampleTools

TXLib Manual

NITRO Texture Library

Version 1.2.0

The contents in this document are highly confidential and should be handled accordingly.

NTR-06-0088-001-A3
Released: August 10, 2005

Contents

Released: August 10, 2005

Tables

Figures

Revision History

Version	Revision Date	Description
1.2.0	7/01/2004	• Updated TXLib source code. • Changed data byte width in `m_colorPaletteAddress` from 2 to 4 bytes.
1.1.0	4/23/2004	• Corrected errors. • Changed `NITRO` in header files to `NITRO-SampleTools`.
1.0.0	3/5/2004	• Revised the manual format. • Changed the specifications for the color processing class. • Added the a3i5 texture format.
0.5.0	1/30/2004	• Initial release.

1 Overview

1.1 What Is TXLib?

TXLib is the library used for converting BMP (`.bmp` extension), TGA (`.tga` extension), and other image files to binary format NTF (`.ntft`, `.ntfp`, and `.ntfi` extensions) or text format NTF (`.c` extension). NTF stands for NITRO Texture File. These files are optimized for embedding into NITRO programs. However, TXLib is only used for inputting from memory or outputting to memory. Therefore, another library is required for outputting to files. Separately from TXLib, we have prepared the TXLibUtility library as a sample of this file output.

1.2 Objective

This library was created to unify the various image files that exist into a proprietary image format, and to unify the programs used to create NTF. This eliminates the need to create a number of programs in order to create NTF from image files, saving time and trouble.

1.3 Recommended Environment

TXLib was developed in the following environment. Program in an environment no earlier than the environment below:

Table 1-1 Environment

OS	Compiler
Windows 2000 with Service Pack 4	Visual C++ .net 2003

2 NITRO Texture File (NTF)

2.1 NTF Texture Format Types

NTF texture format types are texture format types that can be used with NITRO. For details on NITRO texture data structure, see “NITRO Programming Manual.”

For details on the “tex4x4” texture format, refer to “A Description of the NITRO 4x4 Texel Compressed Texture.” The following table shows the NTF texture formats.

Table 2-1 List of NTF Texture Format Types

Texture Format	Content
palette4	4-color palette texture
palette16	16-color palette texture
palette256	256-color palette texture
tex4x4	4x4 texel compressed texture
a3i5	a3I5 translucent texture (32-color palette + 3 bit alpha)
a5i3	a5I3 translucent texture (8-color palette + 5 bit alpha)
direct	Direct color texture

This manual uses the names shown above in the description.

Palette format is used when referring to texture formats other than tex4x4 that have color indices (palette4, palette16, palette256, a3i5, and a5i3).

2.2 NTF File Formats

NTF supports two types of format: binary and text.

2.2.1 Binary Format

Binary format files consist of raw data with no header. There are three files used for handling texel data, palette data, and palette index data, respectively. Their names are as follows.

Texel data: *filename*`.ntft`

Palette data: *filename*`.ntfp`

Palette index data: *filename*`.ntfi`

Since they do not contain texture-related information such as texture format and image size, you cannot load them in TextureViewer. Therefore, in order to preview on a NITRO LCD, it is necessary to create individual programs for displaying the texture.

2.2.2 Text Format

Text format files can collectively handle the three elements — texel data, palette data, and palette index data — in one file. Although they are larger than binary format, they have the following advantages:

1. Because it is text data, you can confirm data content with an editor.
2. Input image file name and the time created are recorded. Therefore, you do not have to remember this type of information.
3. Image size and the converted texture format are recorded. Therefore, you can quickly see what type of texture it is.
4. They can be built into TextureViewer. Therefore, you can easily preview on a NITRO LCD, even without programming knowledge.

TextureViewer is a sample program for displaying NTF data on a NITRO LCD. It was created using the NITRO-SDK and is included in TXLib. For details, see `TextureViewer_manual.pdf`, found in `TextureViewer\doc`.

2.2.3 Text Format Contents

The content of NTF text format is as follows.

```
// NITRO Texture File
// format:    GX_TEXFMT_PLTT256
// width:       128
// height:      128
// original_width:   128
// original_height:  128
// date:      Wed Feb 25 10:36:02 2004
// source:    ../common/inputFiles/mario_128x128_direct.tga

#include <nitro.h>
#include "textureDataTypedef.h"

const u16 palette256_Palette[ 256 ] = {
      0x42DF, 0x4A56, 0x7FFF, 0x0010, 0x6962, 0x7FFF, 0x7FFF, 0x0843,
      0x0014, 0x0C75, 0x3218, 0x61E4, 0x7FFF, 0x7FFF, 0x7FFF, 0x5D21,
 (omitted)
};

const u8 palette256_Texel[ 128 * 128 / 1 ] = {
      0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,  //   0 line
      0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,  //   0 line
 (omitted)
};

const tTextureParameter              palette256_Parameter = {
      16384,                512,    0,
      GX_TEXFMT_PLTT256,    128,    128,    128,    128};

//    texel size,    palette size,  palette index size
//    Format,        Width,         Height, Actual Width,  Actual Height
```

Code 2-1 NTF Text Format Contents

This is the content of an NTF when a texture format was converted from direct to palette256.

A comment about the NTF contents is included at the beginning of the file, so you can identify the type of texture data. The following list explains each item.

- `format` the texture format
- `width` and `height` the size of the output NTF image
- `original_width` and `original_height` the size of the source input image
- `date` the creation date
- `source` the path name of the source input image file

Each element (texel data, palette data, palette index data) is combined into a single file and output. In this example, because the data was converted to palette256, there is palette data (`palette256_Palette`) and texel data (`palette256_Texel`), but no palette index data.

If you select output for TextureViewer when you output the NTF, TextureViewer information (`tTextureParameter`) is also appended. TextureViewer references this information and it is required for previewing on a NITRO LCD.

`tTextureParameter` is defined in the `textureDataTypedef.h` header file. This header file is in the TextureViewer's include folder. It is actually used when compiling TextureViewer.

3 TXLib

3.1 TXLib Overview

TXLib uses the CTexture class for handling textures. CTexture holds color index, palette, alpha, and other elements that are required to construct a texture. By providing data and using the functions that are provided, you can convert texture formats (such as palette4 and direct), reduce palette colors, and more. In order to use TXLib, first provide BMP, TGA, or other image data as input to a CTexture. After image processing, the CTexture in TXLib is output in the image format you want. For example, you can output to memory by passing a CTexture that has been image-processed to the Write function in the CNtf class that handles NTF.

3.2 TXLib Structure Diagram

The diagram on the following page shows the structure of TXLib. The CTexture class handles texture. It is made up of six elements from CData to CAlphaImage. Each element class is switched to enabled or disabled according to CTexture's texture format. When converting CTexture's texture format, it uses CColorProcessor to perform color processing. In TXLib's initial state, this class has been derived, and CDefaultColorProcessor is actually performing color processing. Finally, it uses the CNtf function when outputting an NTF.

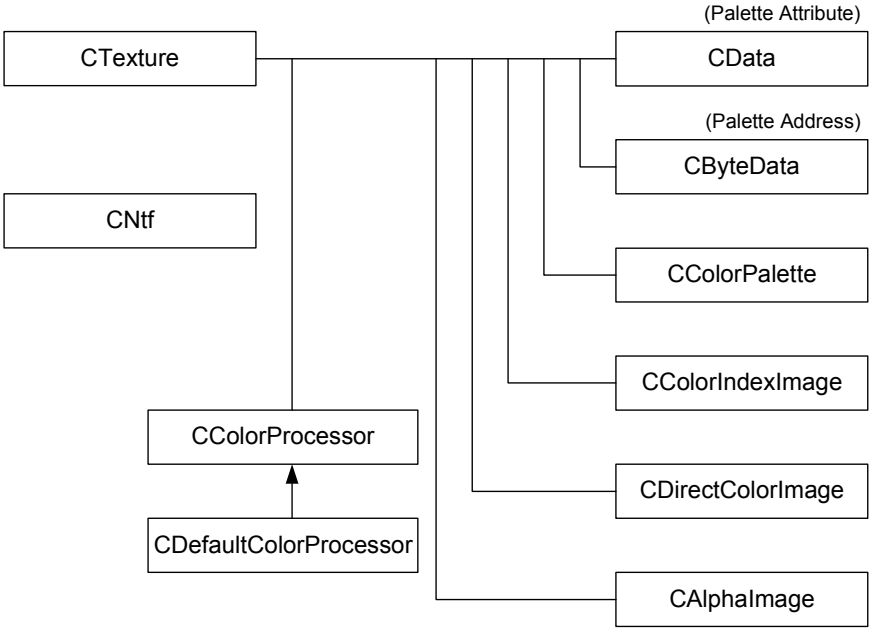


Figure 3-1 TXLib Structure Diagram

3.3 TXLib Class List

This section describes each of the TXLib classes. In the class table, functions for acquiring member variables are omitted.

3.3.1 CData

This class holds and manipulates blocks of 8-bit data. This is the most basic element of TXLib. It is used for color data, color index data, and palette attributes that constitute CTexture.

Table 3-1 CData

Variables		
bool	`m_available`	Whether data can be used
u32	`m_byteSize`	Byte size of data
u8	`m_pData`	Data
Functions		
`void Clear`		Clears memory.
`bool IsAvailable`		Determines whether data is available for use.
`void Create`		Specifies number of data and creates class.
`void GetValue`		Gets data.
`void SetValue`		Sets data.
`void ChangeNumber`		Resizes the block of data.
`void QuickSortTopDown`		Sorts data in ascending or descending order.
`operator=`		Copies another CData object.

3.3.2 CByteData

This class holds and manipulates data columns that have byte and multi-byte widths (8-, 16-, 32-bit), which CData cannot handle. You can specify the number of bytes to use when you create it. You can specify 1, 2, or 4 bytes. Because it has little functionality, use CData for 1-byte wide data. It is used for the palette address that constitutes a CTexture.

Table 3-2 CByteData

Variables		
bool	`m_available`	Whether data can be used
u32	`m_byteSize`	Byte size
u8	`m_byteCount`	Data byte width
u32	`m_number`	Data number
u8*	`m_pData`	Data
Functions		
`void Clear`		Clears memory.
`bool IsAvailable`		Determines whether data is available for use.
`void Create`		Specifies byte width and data number, and creates the class.
`void GetValue`		Gets data.
`void SetValue`		Sets data.
`operator=`		Copies the class.

3.3.3 CColorPalette

This handles the color palette. It contains 8 bits of data (CData) for each of red, green, and blue. It is used as the palette that constitutes a CTexture.

Table 3-3 CColorPalette

Variables		
bool	m_available	Determines whether data is available for use.
CData	m_redChannel	Red data
CData	m_greenChannel	Green data
CData	m_blueChannel	Blue data
Functions		
void Clear		Clears memory.
bool IsAvailable		Determines whether data is available for use.
void Create		Specifies byte width and data number, and creates the class.
void GetValue		Gets color values.
void SetValue		Sets color values.
void CutCommonColor		Cuts common colors in the palette to reduce color number.
void ChangeNumber		Changes the data number.
int FindColor		Acquires from the palette the index of the color that is the same as a specified color value.
void QuickSortTopDown		Sorts brightness in ascending or descending order.
void CommonLowerBit		Converts color values to a bit number that has been pseudo-specified, and increases common colors.
operator=		Copies the class.

3.3.4 CColorIndexImage

This handles 8-bit color index data. It is used by the color index image that constitutes a CTexture.

Table 3-4 CColorIndexImage

Variables		
bool	m_available	Determines whether data is available for use
u16	m_width	Image size: width
u16	m_height	Image size: height
CData	m_colorIndexData	Color index data
Functions		
void Clear		Clears memory.
bool IsAvailable		Determines whether data is available for use.
void Create		Specifies width and height, and creates the class.
void GetValue		Gets index value.
void SetValue		Sets index value.
operator=		Copies the class.

3.3.5 CDirectColorImage

This handles direct color images that have 8 bits each of RGB. It is used as direct color image that constitutes a CTexture.

Table 3-5 CDirectColorImage

Variables		
bool	m_available	Determines whether data is available for use.
u16	m_width	Image size: width
u16	m_height	Image size: height
Cdata	m_redChannel	Red data
Cdata	m_greenChannel	Green data
CData	m_blueChannel	Blue data
Functions		
void Clear		Clears memory.
bool IsAvailable		Determines whether data is available for use.
void Create		Specifies width and height, and creates the class.
void GetValue		Gets color value.
void SetValue		Sets color value.
operator=		Copies the class.

3.3.6 CAlphaImage

This handles 8-bit alpha images. It is used as an alpha image that constitutes a CTexture.

Table 3-6 CAlphaImage

Variables		
bool	m_available	Determines whether data is available for use.
u16	m_width	Image size: width
u16	m_height	Image size: height
CData	m_alphaChannel	Alpha data
Functions		
void Clear		Clears memory.
bool IsAvailable		Determines whether data is available for use.
void Create		Specifies width and height, and creates the class.
void GetValue		Gets alpha value.
void SetValue		Sets alpha value.
operator=		Copies the class.

3.3.7 CTexture

This class handles textures. It holds the elements (from the color index to the palette address) necessary for textures handled by NITRO. In addition, the current texture format is maintained in m_type. The list of texture formats is shown in Table 3-8.

Only required elements are enabled from this format; unnecessary ones are disabled. Table 3-9 shows the relation between each element and the texture formats that enable them.

This texture format can be converted using the Convert function. For details, see Chapter 4 – Texture Format Conversion.

The color processor class is used in the Convert function. For details, see Chapter 5 – Color Processor Class.

Table 3-7 CTexture

Variables		
CColorIndexImage	m_colorIndexImage	Color index
CColorPalette	m_colorPalette	Color palette
CDirectColorImage	m_directColorImage	Direct color image
CAlphaImage	m_alphaImage	Alpha image
CData	m_colorPaletteAttribute	Color palette attribute
CByteData	m_colorPaletteAddress	Color palette address
tTLBTextureType	m_type	Texture format
CColorProcessor*	m_pColorProcessor	Pointer to color processor class

Functions	
void Clear	Clears memory.
void SetType	Sets texture format.
void SetColorProcessor	Sets color processor class.
bool Convert	Converts texture format.
bool IsPaletteType	Determines whether the texture format is a palette format.
bool AlignmentImageSize	Adjusts the image size of the texture to the set size.
operator =	Copies the class.

Table 3-8 List of Texture Formats

Texture Format Name	Texture Format Contents
kType_palette4	4-Color Palette Texture
kType_palette16	16-Color Palette Texture
kType_palette256	256-Color Palette Texture
kType_tex4x4	4x4 Texel Compression Texture
kType_a3i5	a3i5 Translucent Texture (32-color palette + 3-bit alpha)
kType_a5i3	a5i3 Translucent Texture (8-color palette + 5-bit alpha)
kType_direct	Direct Color Texture

Table 3-9 Relationship Between CTexture Elements and the Enabled Texture Formats

Elements in CTexture	Enabled Texture Formats
Color Index Image	All except direct.
Palette	All except direct.
Direct Color Image	direct.
Alpha Image	direct, a5i3, and a3i5.
Palette Attribute	tex4x4.
Palette Address	tex4x4.

CTexture specifications are shown below. Data for the color index and direct color image is arranged from top to bottom (lines) and left to right (pixel rows). Color values consist of eight bits each for RGB, for each texel or each palette color. Alpha values consist of eight bits for each texel. The color index consist of eight bits for each texel. The palette address is the address conforming to NITRO (one address for two colors in a palette).

3.3.8 CNtf

This class handles NTF. It is used when you want to output NTF.

Table 3-10 CNtf

Functions	
`bool Write`	Converts CTexture to NTF data and outputs to memory.
`bool GetDataByteSize`	Acquires the memory size required for NTF data output.

3.3.9 CColorProcessor

This class performs color processing. This is an abstract base class. Actual color processing is performed by a derivative class. For details, see Chapter 5 – Color Processor Class.

3.3.10 CDefaultColorProcessor

This class performs color processing. It is derived from CColorProcessor. In the TXLib initial state, this class is set to CTexture, and is used to perform color processing. If you want to perform color processing by some other method, prepare a class that derives from CColorProcessor, and set that class in CTexture. For details, see Chapter 5 – Color Processor Class.

4 Texture Format Conversion

The `CTexture::Convert` function converts texture formats. This function compares the current texture format and the conversion texture format, and then calls the corresponding conversion function (a protected function) and performs the conversion.

4.1 Converting from Palette Format to Direct

This conversion is performed by the `CTexture::ChangePaletteTypeToDirect` function. The direct color image is created from the color index and the palette colors that it indicates.

The alpha image is created as follows. When converting from a5i3 or a3i5, it uses the value as it is. When converting from the palette4, palette16, or palette256 formats, it uses opaque color (0xff). For direct, be sure to have alpha images. Because the palettes are unnecessary, they will be cleared.

4.2 Converting Between Palette Formats

This conversion is performed by the `CTexture::ChangePaletteType` function.

Palette and color index conversion processing depends on the palette, as described below.

- When converting from a low palette number to a high palette number (for example, from palette16 to palette256), only the texture format changes. Note that it does not change the palette number. For example, even if you convert from palette16 to palette256, the palette number remains 16 (when the palette number of palette16 is 16).
- Color reduction is necessary when converting from a high palette number to a low palette number (for example, from palette256 to palette16), and occurs via the following steps. First, the palette format is converted to direct color data (`CTexture::ChangePaletteTypeToDirect` function). Next, the direct color data is converted to palette format (`CTexture::ChangeDirectToPaletteType` function).

Alpha images are processed as follows.

- When converting from palette4, palette16, or palette256 to a5i3 or a3i5, create using the opaque color (0xff).
- When converting from a5i3 or a3i5 to palette4, palette16, or palette256, alpha is not required and is cleared.

4.3 Converting from Direct to Palette Format

This conversion is performed by the `CTexture::ChangeDirectToPaletteType` function.

The `CColorProcessor::ConvertToPaletteType` function that is set in CTexture creates the palette and color index from the direct color image. See Chapter 5 – “Color Processor Class” for details on the color processor class.

The direct color image is not needed and is cleared. When converting to palette4, palette16, or palette256, the alpha image is not needed and is cleared.

4.4 Converting from Direct to tex4x4 Format

This conversion is performed by the `CTexture::ChangeDirectToTex4x4` function.

The actual conversion is performed by `CColorProcessor::ConvertToTex4x4` function that is set in CTexture. For details, see Chapter 5 – Color Processor Class. Alpha images and direct color images are not needed and are cleared.

4.5 Converting from Palette Format to tex4x4 Format

First, palette format is converted to direct color data (`CTexture::ChangePaletteTypeToDirect` function). Next, direct color data is converted to tex4x4 (`CTexture::ChangeDirectToTex4x4` function).

4.6 Converting from tex4x4 Format to Direct

This conversion is performed by the `CTexture::ChangeTex4x4ToDirect` function.

First, 3-color (transparent color mode) or 4-color (4-color mode) data is extracted from the palette address and the palette attributes (`CTexture::GetTex4x4Palette` function). Next, a direct color image is created from the 3 colors or 4 colors, and the color index. Transparent texels become black because the color index value is 3 and the palette color value is 0.

Finally, the palette address, palette attributes, color index, and color palette are cleared because they are not needed.

4.7 Converting from tex4x4 Format to Palette Format

First, tex4x4 format is converted to direct color data (`CTexture::ChangeTex4x4ToDirect` function). Next, direct color data is converted to palette format (`CTexture::ChangeDirectToPaletteType` function).

5 Color Processor Class

This section describes the color processor class used for texture format conversions.

5.1 CColorProcessor Function List

Below is a list of functions in CColorProcessor, which processes color.

Table 5-1 CColorProcessor

Functions	
`int FindNearColor`	Searches the palette for a color near the specified color value and returns that index. When no color is found, returns -1.
`bool ConvertToPaletteType`	Converts from direct to palette format.
`bool ConvertToTex4x4`	Converts from direct to text4x4.

These functions are pure virtual functions and are not actual functions. To actually perform color processing, you must prepare a CColorProcessor-derived class and define code for how to color process in each function. This allows the user to customize color processing.

In TXLib, a color processing class called CDefaultColorProcessor is provided. In the initial state, color processing can be done with this class.

Below is an explanation of color processing with CDefaultColorProcessor.

5.1.1 int FindNearColor

This procedure is performed with CDefaultColorProcessor.

It calculates the difference for each RGB element between each palette color and the specified color, and applies a weight to each element (R: 30, G: 59, B: 11).

Using the value obtained in step (1), it searches the palette for a similar color using the least squares method, and returns its index.

Because FindNearColor always returns an index, an error (-1) is never returned.

In addition, you can specify in an argument the indices that you do not want to select in order not to return those specific indices (such as transparent color). When no transparent color is used and all indices are targeted, set the parameter to a value below “-1.”

5.1.2 bool ConvertToPaletteType

In the CDefaultColorProcessor, you can select one of the following conversion methods.

1. Create the palette and color index from the colors used in the texture image without using transparent color.
2. Create the color index using the parameter pUsePalette as the palette without using transparent color.
3. Create the palette and color index, mapping the texel of the same color as the parameter transparentColor to the transparent color. Color 0 is set to transparentColor and is used as the transparent color.
4. Create the palette and color index using a texel with an alpha value of less than 0x80 as the transparent color. Color 0 of the palette is set to black (RGB 0,0,0) and is used as the transparent color.
5. Use the texel with the same color as Color 0 of the parameter pUsePalette as the transparent color and create the color index using pUsePalette as the palette.

The value of the conversion parameters determines which of these methods you select.

The list of conversion parameters used by ConvertToPaletteType, and the conversion type selected by that conversion value, are described below.

Table 5-2 List of Parameters used by the CDefaultColorProcessor::ConvertToPaletteType Function

bool	`transparencyFlg`	Transparent color enable flag (`false`: do not use transparent color; `true`: use transparent color)
bool	`alphaColorCutFlg`	Alpha value enable flag (`false`: texel that is same color as transparentColor is the transparent color; `true`: texel with an alpha value of less than 0x80 is the transparent color).
tTLBRGB	`transparentColor`	Color to consider the transparent color.
TXLib::CColorPalette*	`pUsePalette`	Palette to use for conversion.

When `transparencyFlg` is `false` and `pUsePalette` is NULL, then Method (1) is selected.

The direct texture image undergoes color reduction until it fits into the texture format conversion palette. For example, when converting to palette256, color reduction is performed until the palette number can fit within the 256 colors.

If the number of colors in the texture image after removing common colors is smaller than the palette number of the created texture, all the colors with the common colors removed are copied directly to the palette. For example, when converting to palette256, if the number of colors in the texture image after removing the common colors is 250, those 250 colors are copied directly to the palette.

If the number of colors is greater than the palette number, then the palette is created using color reduction with the median cut method.

This color reduction process is not limited to Method (1) above, but is used whenever a palette is created (when `pUsePalette` is NULL).

The palette created last is allocated to the direct color image, and a color index is created.

When `transparencyFlg` is `false` and a palette has been specified in `pUsePalette`, then Method (2) is selected.

Because the palette is set, only the color index is created. If the palette number of the specified palette is greater than the palette number held by the created texture, then the extra palettes are cut. For example, when converting to palette16, if the palette number of the specified palette is 30 colors, then 14 extra colors will be cut and the palette number will be adjusted to 16 colors.

When `transparencyFlg` is `true` and `alphaColorCutFlg` is `false`, then Method (3) is selected.

`transparentColor` is put in Color 0 of the palette, that color and the direct color image are compared, and an alpha image is created. If the color and the direct color image are the same, the alpha value is set to 0x00; if they are different, the alpha value is set to 0xff. This alpha image is necessary when not including the color considered transparent in the color reduction algorithm, or when it is necessary to distinguish between color values and transparent colors when creating a color index. Specifically, alpha values of less than 0x80 are processed as transparent; values greater than this are opaque.

The method of creating the palette and color index is the same as Method (1), excluding the consideration of the alpha value.

When `transparencyFlg` is `true` and `alphaColorCutFlg` is `true`, then Method (4) is selected.

The palette and color index are created the same as in Method (3). The difference with Method (3) is setting a color considered transparent (black) in Color 0 of the palette. If there is not enough space in the palette to insert this color (for example, a 256-color palette is already created when converting to palette256), then the palette is shifted over one and the color is inserted. For this reason, the end of the original palette is overwritten.

When `transparencyFlg` is `true` and a palette has been specified in `pUsePalette`, then Method (5) is selected.

Because the palette is set, the same process as in Method (3) occurs with Color 0 of the palette considered transparent.

5.1.3 bool ConvertToTex4x4

In the `CDefaultColorProcessor`, the conversion to tex4x4 is the same as the conversion to palette format (`ConvertToPaletteType` function), a processing method can be selected based on the value of the conversion parameters.

The list of parameters used by `ConvertToTex4x4` is shown in Table 5-3.

Table 5-3 List of Parameters used by the CDefaultColorProcessor::ConvertToTex4x4 Function

bool	`transparencyFlg`	Transparent color enable flag (`false`: do not use transparent color; `true`: use transparent color).
tAlgorithm	`algorithm`	Algorithm.
bool	`alphaColorCutFlg`	Alpha value enable flag (`false`: texel that is same color as `transparentColor` is the transparent color; `true`: texel with an alpha value of less than 0x80 is the transparent color).
tTLBRGB	`transparentColor`	Color that is considered transparent.
tTLBRGB	`commonColorRange`	Range value of standardized colors for the tex4x4 palette when compressing palette number.
bool	`compressFlg`	Palette number compression enable flag (`false`: no compression; `true`: compression).

tex4x4 palette is the palette used with tex4x4. When linear interpolation is used, only two colors are indicated. When linear interpolation is not used, four colors are indicated.

Table 5-4 tAlgorithm Algorithm

Function	Description
`kAlgorithm_fastNotLinear`	Fast; no linear interpolation.
`kAlgorithm_fastLinear`	Fast; with linear interpolation.
`kAlgorithm_roundRobinNotLinear`	Round-robin; no linear interpolation.
`kAlgorithm_roundRobinLinear`	Round-robin; with linear interpolation

The following is a step-by-step explanation of the internal process of the `ConvertToTex4x4` function.

(1) **Create a 1 block direct color image.**

One block of direct color image is extracted from all the direct color images. “One block” refers to a block of 4x4 texels, and 16 colors will be extracted.

(2) **When transparent color is enabled, create a one block binary alpha image (0x00 or 0xff).**

When `transparencyFlg` is `true`, a one block binary alpha image is created. This image is created to find out which texel in the direct color image is transparent.

There are two methods to create a one-block alpha image: from the color designated transparent or from the source alpha image.

When `alphaColorCutFlg` is `false`, an alpha image is created from the color that has been designated as the transparent color. The color value of the direct color image texel is compared to the color value of the specified `transparentColor`. If the color values are the same, then transparency (0x00) is set as the alpha value and then the alpha image created. If the values are different, then opacity is set as the alpha value and then the alpha image created.

When `alphaColorCutFlg` is `true`, then the source alpha image becomes binary and a new alpha image is created. If the alpha value of the base alpha image is less than 0x80, a transparent color (0x00) is used when creating the image; if the value is more than 0x80, then an opaque color (0xff) is used when creating the image.

When the alpha values for the alpha image created here are opaque (0xff) for all texels, it is determined that no transparent texels are in that block.

(3) **Create a tex4x4 palette from one block.**

A less than 4-color tex4x4 palette is created from one block. Although 1 block uses 16 colors, the common colors are removed from these 16 colors. If the remaining color number is two or less, then they are copied to the tex4x4 palette. If three or more colors are remaining, a tex4x4 palette is created using the algorithm specified by “algorithm.”

(4) **Create palette attributes.**

The characteristics of the block are set in palette attributes according to the color and mode (transparent color mode or 4-color mode) of the created tex4x4 palette. If there are two or fewer colors in the tex4x4 palette, then there is linear interpolation. If there are 3 or 4 colors in the palette, there is no linear interpolation.

(5) **Create the palette and palette address.**

In the processing of the first block, the created tex4x4 palette is added without modification to the group of tex4x4 palettes. Because the tex4x4 group must have an even number of colors, if the added tex4x4 palette has one or three colors, a dummy color (black) is added to the tex4x4 palette group. Furthermore, the palette address is set at this time. The palette address of the first block is “0.”

For the second and subsequent blocks, a search is made for a palette in the tex4x4 palette group being created that is the same as the tex4x4 palette of the block. If a matching tex4x4 palette is found, the address of that palette is set as the palette address. If a match is not found, that tex4x4 palette is added to the tex4x4 palette group, and a new palette address is set. When palette color values are compared, the range within which colors are judged to be the same is set in `commonColorRange`.

`commonColorRange` has RGB color elements. When determining whether blocks have the same tex4x4 palettes, if the tex4x4 palettes are within the `commonColorRange`, they are designated as the same tex4x4 palette, and the tex4x4 palette is shared. The values of the `commonColorRange` elements can be specified in a range between 0 – 31 because they are used internally by the `ConvertToTex4x4` function after multiplying by 8 (because the color value is pseudo-processed as 5 bit in the function).

`compressFlg` must be `true` for this the tex4x4 palette to be a shared process. If `compressFlg` is `false`, tex4x4 palettes are always added resulting in no compression of palette number. When the tex4x4 palette has a color number of “0” (when the entire texel for the block is transparent), “0” is set as the palette address and the tex4x4 palette is not created.

(6) **Create a color index.**

The created tex4x4 palette is allocated to the direct color image and a color index is created. When the tex4x4 palette has two colors, linearly-interpolated colors are allocated, virtually creating four colors. These four colors are used to create the color index.

When the texel is transparent (alpha value of less than 0x80), the color index value is set to 3, which indicates transparency. For all other colors, a similar color is found in the palette using the `CDefaultColorProcessor::FindNearColor` function and the appropriate color index is allocated.

(7) **Change the palette number to the actual palette number being used.**

The initial tex4x4 conversion initial status secures the maximum palette number (4 colors per block) of necessary memory in order to create the tex4x4 palette group. For this reason, when a tex4x4 palette that uses linear interpolation for a block is used, that block has two extra colors in its palette. To eliminate this excess, the palette number is changed to the actual palette number being used.

(8) **Compress the palette number for the tex4x4 group.**

Finally, compress the number of colors in the tex4x4 palette group to under 32,768 colors in order to prevent the palette number from exceeding the NITRO palette memory.

6 TXLibUtility (TXLib High Level Library)

6.1 Overview

The TXLib package contains the ntexconv conversion tool to convert image files to NTF. This tool, created using TXLib, uses the high-level TXLibUtility library to access TXLib.

TXLibUtility is a library used to pass image file data to TXLib and to output data received from TXLib as image data or NTF. The following describes how to use TXLibUtility.

To learn how to use the ntexconv tool, see `\doc\ntexconv_manual.pdf`.

6.2 TXLibUtility Structure Diagram

The following diagram shows the structure of TXLibUtility.

CBmp and CTga are classes for handling BMP and TGA , respectively. They are used to input BMP or TGA image data to TXLib or to output BMP or TGA image data from TXLib. The Utility function is used to input and output BMP and TGA image data using files. TXLib's CNtf also uses the Utility function to output files.

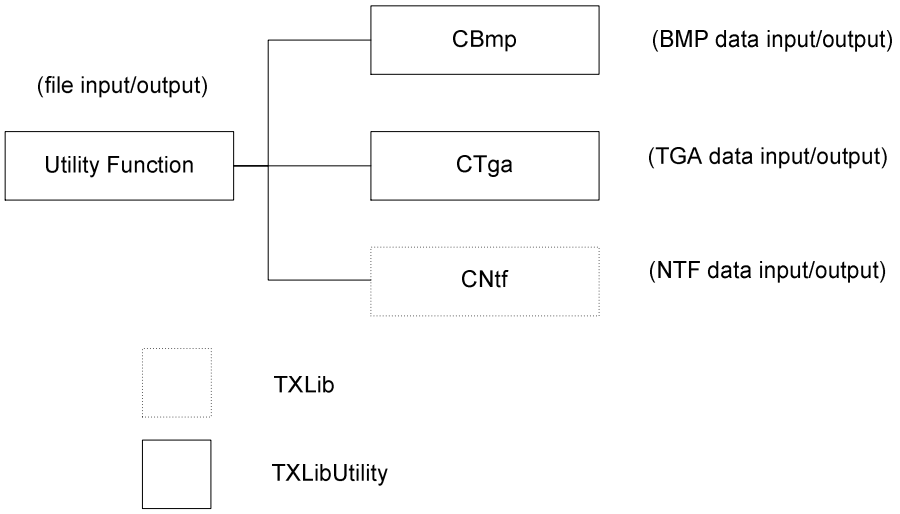


Figure 6-1 TXLibUtility Block Diagram

6.3 TXLibUtility List

The following table lists the class functions and the TXLibUtility functions.

Table 6-1 CBmp

Function	
`bool Read`	Converts BMP data that is in memory to CTexture.
`bool Write`	Converts CTexture to BMP data and outputs to memory.
`bool GetDataByteSizeMax`	Acquires the memory size required to output BMP data.
`TXLib::CTexture::tType GetConvertType`	Acquires the conversion texture format.

Table 6-2 CTga

Function	
`bool Read`	Converts TGA data that is in memory to CTexture.
`bool Write`	Converts CTexture to TGA data and outputs to memory.
`bool GetDataByteSizeMax`	Acquires the memory size required to output TGA data.
`TXLib::CTexture::tType GetConvertType`	Acquires the conversion texture format.

Table 6-3 TXLibUtility Function

Function	
`bool Load`	Loads image data from BMP and TGA files and creates CTexture.
`bool SaveNtf`	Converts CTexture to NTF data and stores to file.
`bool Save`	Converts CTexture to BMP or TGA data and stores to file.
`bool ChangeExtName`	Changes the file path name extension to the specified extension.

6.4 CBmp and CTga Function Details

This section explains the processes performed internally by the `CBmp` and `CTga` functions. Although `CBmp` and `CTga` are different image formats, the processes are the same. This section uses `CBmp` to illustrate these processes.

6.4.1 Read Function

The `Read` function performs the following processes.

1. The BMP header portion is read into memory, and the BMP image format, image size, and palette number are loaded from the header portion.
2. CTexture is created using the palette format if the image has a color index, and using direct color data if the image is 24-bit direct. For example, if palette256, CTexture is created at a size necessary for the palette and color index, and then data is entered. The texture format is then set to palette256. If direct, the direct color index is created and data entered. Because an alpha image does not exist in the source image, it is created as opaque (0xff).

6.4.2 GetConvertType Function

This function obtains the texture format of the BMP output. For example, when the texture format is tex4x4, there is no corresponding BMP image format, so the format is converted to direct before being output. This function obtains the texture format, in this case “direct.”

6.4.3 GetDataByteSizeMax Function

This function obtains the memory size necessary to output data to memory. When data is output, memory of the size obtained by this function is allocated. Because palette numbers are not calculated at this point, memory size for the maximum palette number is returned, in other words 16 for palette16 or 256 for palette256. The actual memory size is obtained after memory is written to by the following Write Function.

6.4.4 Write Function

This function converts CTexture data to BMP data and then outputs that data. The following operations occur internally.

1. When the texture format is not compatible with the BMP image format, the texture format is converted. For example, tex4x4 is converted to direct because BMP does not have tex4x4 formats.
2. The BMP data header portion is created from the texture format.
3. The created header portion is written to memory.
4. The palette, color index, and direct color image are rearranged as BMP data and output to memory.
5. Finally, the memory size output to memory is returned.

6.4.5 Relation of CBmp Image Formats to CTexture's Texture Formats

The following table shows the BMP image format types handled by `CBmp`.

Table 6-4 BMP Image Format Types Handled by TXLibUtility::CBmp

Image Format	Description
palette2	2-color Color Index
palette16	16-color Color Index
palette256	256-color Color Index
24bit direct	8-bit per pixel RGB Direct Color Image

Note: `CBmp` cannot handle RLE-compressed image data.

Input of BMP formats not supported by CTexture is handled as follows:

- palette2 is converted to palette4
- 24-bit direct is converted to direct

BMP output of texture formats not supported by BMP format is handled as follows:

- palette4 and a5i3 are converted to palette16
- a3i5 is converted to palette256
- tex4x4 is converted to 24-bit direct

6.4.6 Relationship of CTga Image Formats to CTexture's Texture Formats

The following shows the TGA image format types that are handled.

Table 6-5 TGA Image Format Types Handled by TXLibUtility::CTga

Image Format	Description
gray	Grayscale (8-bit Brightness Index)
palette256	256-color Color Index
16bit direct	5-bit per RGB Direct Color Image, 1-bit alpha
24bit direct	8-bit per RGB Direct Color Image
32bit direct	8-bit per RGB Direct Color Image, 8-bit alpha

Input to CTexture of TGA formats not supported by CTexture is handled as follows:

- Gray is converted to direct
- 16-bit direct is converted to direct
- 24-bit direct is converted to direct

TGA output of texture formats not supported by TGA is handled as follows:

- palette4 and palette16 are converted to palette256
- a5i3, a3i5, and tex4x4 are converted to 32bit direct

6.5 TXLibUtility Function Details

The processes performed internally by TXLibUtility functions are described in the flow charts below.

6.5.1 TXLibUtility::Load

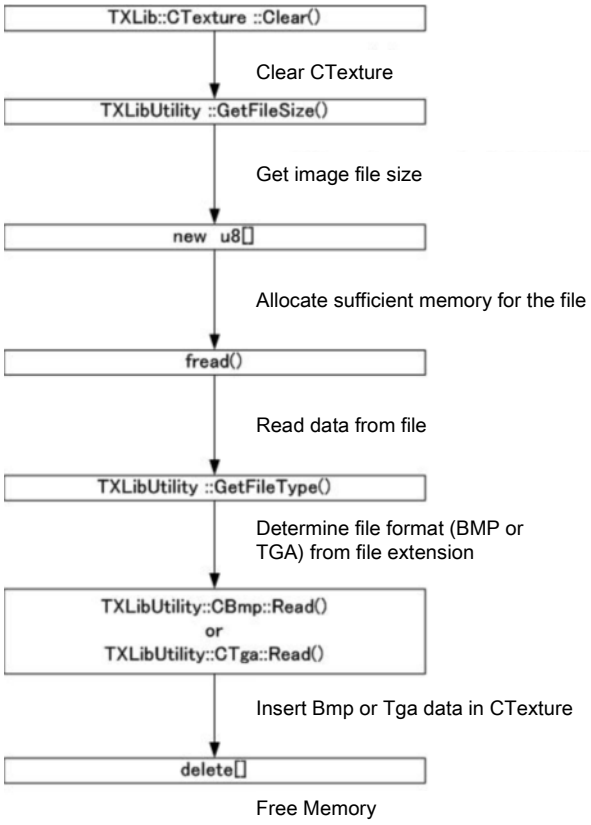


Figure 6-2 Block Diagram of the TXLibUtility::Load Function

1. Clears the current CTexture.
2. Secures memory to load the file data. Uses the `TXLibUtility::GetFileSize` function to obtain the memory size necessary to load the data.
3. Data from the file is loaded to the secured memory. When BMP data is loaded, the `CBmp Read` function is used, when TGA data is loaded, the `CTga Read` function is used to enter image data to CTexture.
4. Finally, the secured memory is released using `delete[]`. Do not forget to release memory as failure to do so may lead to memory leaks.

6.5.2 TXLibUtility::SaveNtf

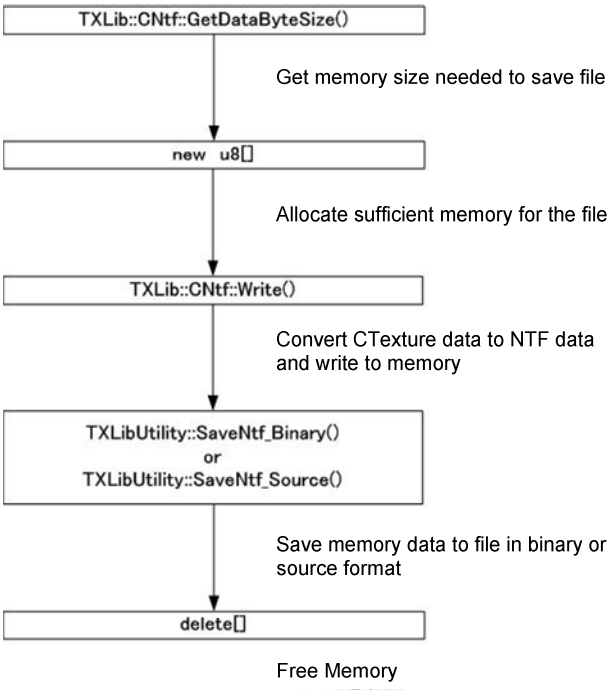


Figure 6-3 Block Diagram of the TXLibUtility::SaveNtf Function

1. Secures memory necessary for saving the file. The `TXLib::CNtf::GetDataByteSize` function is used to obtain the memory size necessary to save the file.
2. NTF data is written to the secured memory. Use the `CNtf Write` function to realign the data order to NTF data order and output the data to memory. You can select using an argument whether or not to adjust the NTF image size. See Chapter 8 – NTF Image Size for details on adjusting NTF image sizes.
3. The data in memory is output to a file in binary or text format using either the `TXLibUtility::SaveNtf_Binary` or `SaveNtf_Source` function. The `SaveNtf` function argument selects which format to output.
4. Finally, the secured memory is released using `delete[]`. Do not forget to release memory as failure to do so may lead to memory leaks.

6.5.3 TXLibUtility::Save

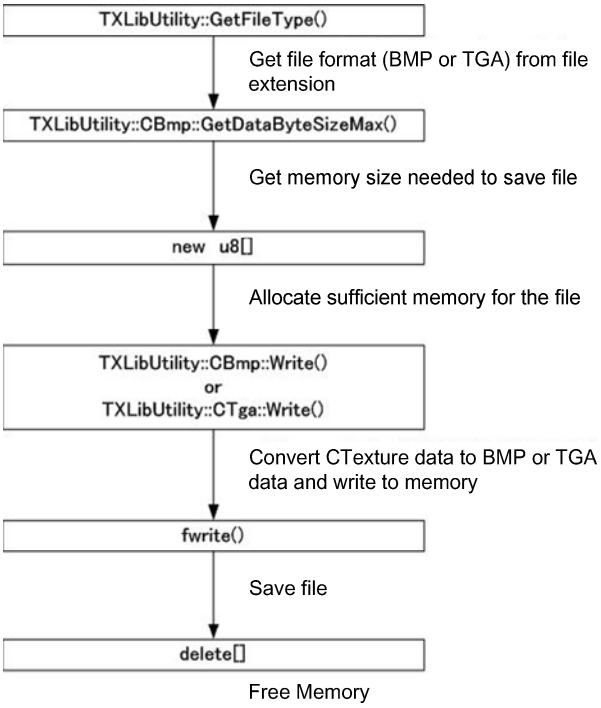


Figure 6-4 Block Diagram of the TXLibUtility::Save Function

This function has the same flow as the `SaveNtf` function.

When outputting as BMP data, use the `CBmp Write` function. When outputting as TGA data, use the `CTga Write` function.

7 Sample Code

To make the samples more readable, error processing is not performed here. We have also prepared sample programs in `TXLib\build\samples`. For practical source samples, please refer to that location.

7.1 Code Example 1

This sample reads one image file and outputs it in NTF without changing the texture format.

```
TXLib::CTexture texture;

TXLibUtility::Load (
              &texture,
              "../common/inputFiles/mario_128x128_palette256.bmp") ;

TXLibUtility::SaveNtf (
              "outputFiles/after.c", // output NTF path name
              &texture,              // CTexture
              false,                 // output text format
              NULL,                  // input image file path name
              false,                 // does not output TextureViewer file
              true                   // alignment NTF image size
              ) ;
```

Code 7-1 Code Example 1

This sample uses the Load function to load an image file. It determines whether it is BMP or TGA from the file name extension, reads the data, and stores it in CTexture. It is converted to NTF and output by the `SaveNtf` function. If the third argument is `false`, output is in text format. If it is `true`, output is in binary format. If you input the path name of the input image file in the fourth argument, a record of what image file was converted will be left in the NTF. You can use NULL if you do not need this information. Select `true` for the fifth argument only if you want information for TextureViewer to be output in the file. Set the sixth argument to `true` when the NTF image size will be adjusted. See Chapter 8 – NTF Image Size for details on adjusting NTF Image sizes.

7.2 Code Example 2

This sample reads one image file, converts the texture format from palette256 to palette16, and outputs NTF. It also outputs a BMP file for image confirmation on a PC.

```
TXLib::CTexture texture;

TXLibUtility::Load (
      &texture,
      "../common/inputFiles/mario_128x128_palette256.bmp"
      ) ;

texture.Convert (TXLib::CTexture::kType_palette16) ;

TXLibUtility::SaveNtf (
      "outputFiles/after.c",         // output NTF path name
      &texture,                      // CTexture
      false,                         // output text format
      NULL,                          // input image file path name
      false,                         // does not output TextureViewer file
      true                           // alignment NTF image size
      ) ;

TXLibUtility::Save (
      "outputFiles/palette256ToPalette16.bmp",
      &texture
      ) ;
```

Code 7-2 Code Example 2

This `Convert` function is used to convert the CTexture created with the Load function into any texture format. This example converts to palette16.

This code outputs NTF with the `SaveNtf` function, and outputs a BMP file for image confirmation with the `Save` function.

7.3 Code Example 3

This sample reads one TGA file, converts the texture format from palette256 to tex4x4, and outputs NTF.

```
TXLib::CTexture texture;
TXLib::CDefaultColorProcessor::tConvertFactor       factor;

std::memset (  (void*) &factor, 0,
      sizeof (TXLib::CDefaultColorProcessor::tConvertFactor) ) ;

factor.algorithm =
      TXLib::CDefaultColorProcessor::kAlgorithm_roundRobinNotLinear;

factor.transparencyFlg       = true;
factor.alphaColorCutFlg      = false;
factor.transparentColor.red  = 0xff;
factor.transparentColor.green= 0xff;
factor.transparentColor.blue = 0xff;

factor.commonColorRange.red  = 0;
factor.commonColorRange.green= 0;
factor.commonColorRange.blue = 0;
factor.compressFlg           = true;

TXLibUtility::Load (
      &texture,
      "../common/inputFiles/mario_128x128_palette256.bmp"
      ) ;

texture.GetColorProcessor () ->SetConvertFactor (&factor) ;
texture.Convert (TXLib::CTexture::kType_tex4x4) ;

TXLibUtility::SaveNtf (
      "outputFiles/palette256ToTex4x4.c",    // output NTF path name
      &texture,                     // CTexture
      false,                        // output text format
      NULL,                         // input image file path name
      false,                        // does not output TextureViewer file
      true                          // alignment NTF image size
      ) ;

TXLibUtility::Save (
      "outputFiles/palette256ToTex4x4.bmp",
      &texture
      ) ;
```

Code 7-3 Code Example 3

When converting to tex4x4, the conversion parameters structure specifies how the conversion will be performed. Values are input into each variable of the structure, and set in TXLib using the `SetConvertFactor` function. By doing so, the conversion parameter values are used for converting to tex4x4. Therefore, you must call the `SetConvertFactor` function before the Convert function. You can also customize the conversion parameters along with the color processor class. For details, see Chapter 5 – Color Processor Class.

7.4 Code Example 4

This is a sample of customizing the color processor class function, ConvertToPaletteType, that converts from direct to palette formats.

```
(in main.cpp)
TXLib::CTexture texture;

TXLibUtility::Load (
      &texture,
      "../common/inputFiles/mario_128x128_direct.tga"
      );

CMyColorProcessor colorProcessor;
texture.SetColorProcessor ( &colorProcessor ) ;

texture.Convert ( TXLib::CTexture::kType_palette256 ) ;

TXLibUtility::SaveNtf (
              "outputFiles/after.c", // output NTF path name
              &texture,              // CTexture
              false,                 // output text format
              NULL,                  // input image file path name
              false,                 // does not output TextureViewer file
              true                   // alignment NTF image size
) ;

TXLibUtility::Save (
              "outputFiles/after.bmp",
              &texture
) ;


(in MyColorProcessor.cpp)
bool CMyColorProcessor::ConvertToPaletteType (
      TXLib::CTexture*       pTexture
)
{
      // Color Processing
      // Create function to convert from direct to palette format
}
```

Code 7-4 Code Example 4

When you want to customize a function of a color processing class, this prepares an original, CDefaultColorProcessor-derived color processor class, and overrides the function you want to customize. In this example, the `ConvertToPaletteType` function that converts from direct to palette format is overridden, and the function will perform customized color processing.

In order to use the created function in TXLib, set the color processor class with the `SetColorProcessor` function. By doing this, the customized function `CMyColorProcessor::ConvertToPaletteType` will be called as the function that converts from direct to palette format.

The `SetColorProcessor` function must be called before the Convert function. For details on the color processor class, see Chapter 5 – Color Processor Class.

8 NTF Image Size

Texture sizes that can be used in NITRO are limited to 8, 16, 32, 64, 128, 256, 512 or 1024 in the height or width. When NTF data is output (`CNtf::Write` function), NTF image sizes can be adjusted to conform to these restrictions or can be output without size restrictions. This selection is made with an argument in the `CNtf::Write` function.

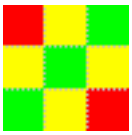
The following table describes how NTF is output for input image data sizes when restrictions are added to the NTF image size.

Table 8-1 Relationship between Size of Input Image Data and NTF Output

• Image size of input image data	• NTF Output
• Either height or width is “0.”	• Error.
• Either height or width exceeds “1024.”	• Error.
• Both height or width are between “1” and “1024” and follow NITRO size limitations.	• Normal output.
• Both height or width are between “1” and “1024” but do not follow NITRO size restrictions.	• Expands the image to the closest NITRO image size and outputs.

The following is an example of input image data when the image size does not follow the NITRO size restrictions.

Example: When the Input Image Data Size is 47 x 47



• Input Image Data	• Output NTF
• 47x47	• 64x64

The closest NITRO size for the image size of this input image data is 64 x 64. The NTF image is expanded to 64 x 64, and the expanded areas are written with data from the edges of the source image.