NintendoWare for CTR

Character and Text Rendering

CharWriter / TextWriter / TagProcessor

2010/09/27

Ver. 1.1.0

PROVISIONAL TRANSLATION

The content of this document is highly confidential and should be handled accordingly.

Table of Contents

Code

Figures

1 Introduction

This manual describes the following NintendoWare for CTR classes:

- `nw::font::CharWriter`
- `nw::font::TextWriter`
- `nw::font::TagProcessor`

These three classes are derived from the `Font` class to render characters and text.

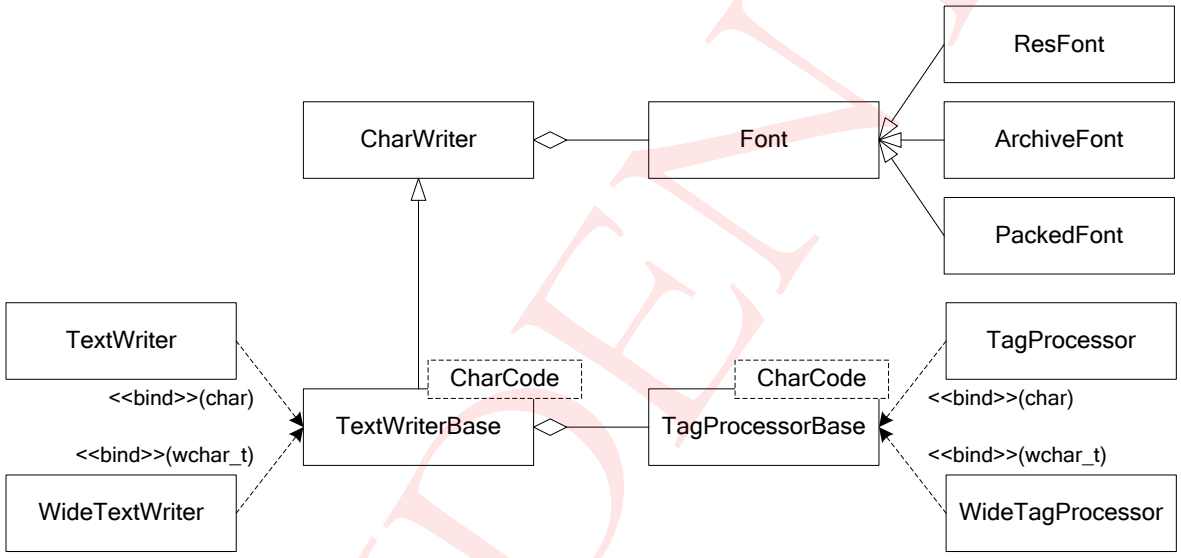
All matters common to overall text rendering are covered in *Text Rendering Fundamentals* (`DrawText_First.pdf`). We recommend that you read *Text Rendering Fundamentals* before reading this document.

2 Library Structure

2.1 Class Structure

The Character Rendering library (a class library) includes the Text Rendering library as one of the classes. Figure 2-1 shows the relationships among the main classes of the Character Rendering library.

Figure 2-1 Class Interrelationships



2.1.1 Font Class and Derived Classes

`Font` is a class that represents character fonts. Since the `Font` class is merely an interface, the actual implementation of character fonts is contained in the classes `ResFont`, `ArchiveFont`, and `PackedFont`. You will need to create an instance of one of these derived classes for use as a Font class.

For details about the Font class and its derived classes, see the Font manual (`Font.pdf`). A description of Font and its derived classes is not provided in this manual.

2.1.2 CharWriter

`CharWriter` is the class that renders characters using the font supplied by the appropriate Font class. The actual rendering is a task shared with the `RectDrawer` class, which is described below. Since `CharWriter` can render only single characters, this class is used for special applications, such as when you independently implement features of `TextWriter`.

2.1.3 TextWriter

`TextWriter` is a derived class that renders text (and in the same way shares the rendering task with the `RectDrawer` class). Because `TextWriter` is a derived class of `CharWriter`, `TextWriter` can do everything that `CharWriter` can do. For this reason, `TextWriter` will normally be used to render both characters and text.

Because `TextWriter` is the name of the class that gets created as an instantiation of the template class `TextWriterBase`, there are no definitions unique to the `TextWriter` class. The same is true of the `WideTextWriter` class, which is also an instantiation of the `TextWriterBase` template class, but with different parameters.

This manual sometimes uses the term `TextWriter` to refer to all three classes, namely, `TextWriterBase`, `TextWriter`, and `WideTextWriter`.

2.1.4 RectDrawer

This is the class for creating the rendering commands for the graphics hardware based on the data created with the `CharWriter` and `TextWriter` classes. Chaarcters are rendered in real time by passing the rendering commands to the graphics hardware.

2.1.5 TagProcessor

`TagProcessor` is a class used for processing tags embedded in text that `TextWriter` is rendering. The `TagProcessor` class can process only the tab character and the carriage return character. To process tags that are embedded in text, you need to create a class derived from `TagProcessor` in which you override the virtual member functions `Process` and `CalcRect`.

Because `TagProcessor` is the name of the class that gets created as an instantiation of the template class `TagProcessorBase`, there are no definitions unique to the `TagProcessor` class. The `WideTagProcessor` class is also an instantiation of the `TagProcessorBase` template class, but with different parameters.

This manual sometimes uses the term `TagProcessor` to refer to all three classes, namely, `TagProcessorBase`, `TagProcessor`, and `WideTagProcessor`.

2.1.6 WideTextWriter and WideTagProcessor

The only difference between `TextWriter` and `WideTextWriter` is the type of character encoding they handle. Similarly, the only difference between `TagProcessor` and `WideTagProcessor` is the type of character encoding they handle. `WideTextWriter` and `WideTagProcessor` process UTF16 text, whereas `TextWriter` and `TagProcessor` process all other types of encoding, namely, CP1252, ShiftJIS, and UTF8.

2.1.7 TextWriterBase and TagProcessorBase

`TextWriterBase` is the class template used to create the `TextWriter` and `WideTextWriter` classes, which differ only in the type of character encodings that they handle. `TextWriterBase` cannot be

used to create any classes other than `TextWriter` and `WideTextWriter`, so use this class template only for those two classes.

The same relationship holds for `TagProcessorBase` with respect to the classes `TagProcessor` and `WideTagProcessor`.

2.2 Supported Operations

The Character Rendering library classes let you do the following to characters or strings:

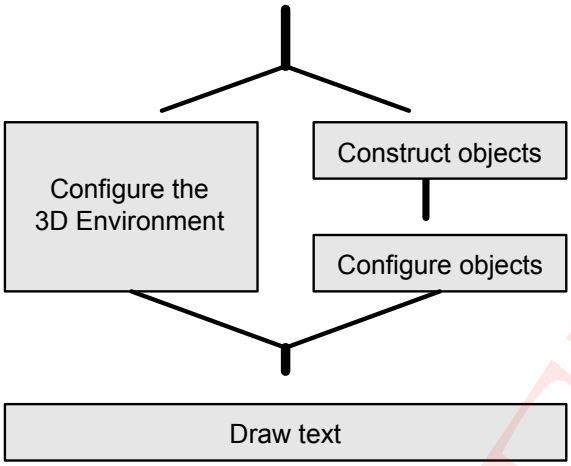
- Change the color of a character
- Apply gradation horizontally or vertically to a character
- Scale a character
- Force a character's font to be treated as a monospaced font
- Process kerning, leading, and tab width
- Align a character left or right
- Wrap strings automatically
- Render a character as formatted text (printf)
- Process tagged text

For details, see Chapter 4 Character Rendering Features.

3 Text Rendering Flow

Figure 3-1 broadly depicts the process flow for text rendering. Items shown in parallel in the figure can be executed in any order. This chapter describes the individual items shown in Figure 3-1.

Figure 3-1 Text Rendering Flow



3.1 Constructing Objects

In order to render text, you need at the very least an instance of a class derived from the Font class, an instance of either the `CharWriter` class or the `TextWriter` class, and an instance of the `RectDrawer` class. To render tagged text you also need an instance of a class derived from the `TagProcessor` class.

The rest of this section explains how to construct these objects.

3.1.1 Derived Font Classes

After you have created an instance of the class derived from the Font class, you need to build the font. The details for this are in the *Font* manual (`Font.pdf`).

3.1.2 CharWriter and TextWriter

Memory for the rendering data is configured for the `CharWriter` class and the `TextWriter` class, set to the size needed for the text being rendered.

Use the `CharWriter::GetDispStringBufferSize` function to get the memory size for the corresponding number of characters, use the `CharWriter::InitDispStringBuffer` function to initialize the allocated memory, and then set this memory to the `CharWriter` class and the `TextWriter` class.

3.1.3 RectDrawer

For the `RectDrawer` class you need to construct the object based on the shader binary for text rendering and the memory for the render settings.

The filename of the shader binary used by the font is defined by the `NW_FONT_RECTDRAWER_SHADERBINARY` macro. The memory size for the render settings is obtained using the `RectDrawer::GetVertexBufferCommandBufferSize` funciton.

3.1.4 Classes Derived From TagProcessor

Because the `TagProcessor` class itself does not need any special initialization, the derived classes do not need a constructor, unless some special process is being implemented that requires initialization.

3.2 Configuring Objects

`CharWriter` and `TextWriter` both have a wide variety of properties. By reconfiguring these properties, you can alter the way characters are rendered.

During rendering, characters are rendered using the properties of the rendering functions that exist at the time the functions are called.

For details about the various properties, see Chapter 4 Character Rendering Features.

3.3 Configuring the 3D Environment

The Character Rendering library can use the CTR system's 3D display feature to display text. The items listed below are not configured by the Character Rendering library, so you will need to set them appropriately before rendering text.

- Culling
- Scissoring
- Polygon Offset
- Early Depth Test
- Depth Test
- Stencil Test
- Masking
- Framebuffer Object

Note that if Culling and Depth Test have been configured to display 3D models, you will need to reconfigure them before text rendering or else the prior settings will affect how the text is rendered.

3.3.1 Coordinate System

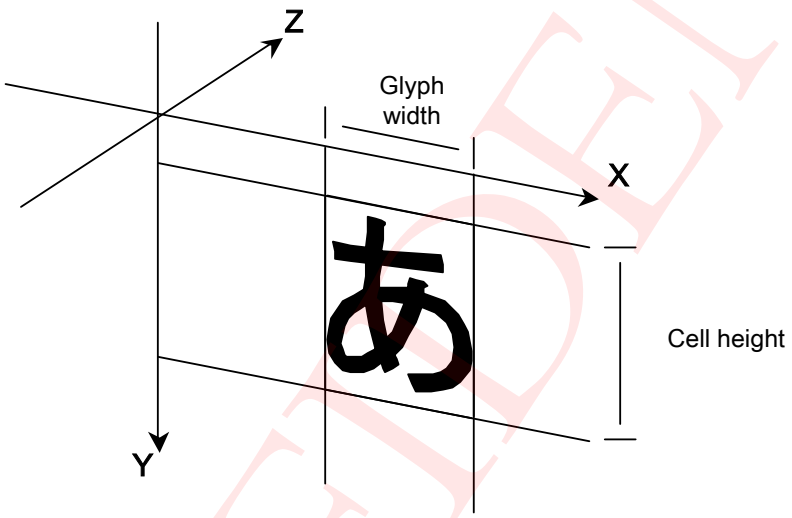
The Character Rendering library takes the coordinate system shown in Figure 3-2 and assumes an orthogonal projection space with a viewpoint orthogonal to the Z axis.

The Y axis and Z axis are reversed from the coordinate system normally used with CTR. When a character is rendered, a texture is mapped to the XY plane of this space on a polygon that is the width of the glyph and the height of the cell in the XY plane of this space.

Normally, the projection matrix is set for the orthogonal projection so that the viewing surface size and the screen size are the same. Note that (0, 0) is defined as the upper-left coordinate of the screen and an identity matrix is set for the position coordinate matrix.

Use the `RectDrawer::SetProjectionMtx` function to set the projection matrix, and use the `RectDrawer::SetViewMtxForText` function to set the position coordinate matrix.

Figure 3-2 Coordinate System for Character Rendering



3.3.2 RectDrawer::DrawBegin, RectDrawer::DrawEnd

The `RectDrawer::DrawBegin` function configures the drawing environment necessary for text rendering. The `RectDrawer::DrawEnd` function performs post-processing of the render settings.

3.3.3 RectDrawer::BuildTextCommand

The `RectDrawer::BuildTextCommand` function creates rendering commands for the graphics hardware in the memory for rendering data configured by the `CharWriter` and `TextWriter` classes. It is called between the `RectDrawer::DrawBegin` function and the `RectDrawer::DrawEnd` function.

3.3.4 CharWriter::UseCommandBuffer

The `CharWriter::UseCommandBuffer` function sends to the graphics hardware the rendering commands which were created by the `RectDrawer::BuildTextCommand` function. It is called between the `RectDrawer::DrawBegin` function and the `RectDrawer::DrawEnd` function.

3.4 Rendering Characters (Text)

3.4.1 Starting and Ending Rendering

The `CharWriter` and `TextWriter` class rendering functions which are described below must be called between the `CharWriter::StartPrint` function and the `CharWriter::EndPrint` function.

3.4.2 Differences Among the Rendering Functions

`CharWriter` and `TextWriter` both offer a number of functions for the rendering of characters and text.

The functions and their main features are:

- `CharWriter::DrawGlyph`

Renders glyphs. This function renders polygons mapped with glyph textures at the position of the cursor. The character's left-space and right-space parameters are ignored, as are the `CharWriter` monospace render settings.

- `CharWriter::Print(CharCode)`

Renders one character of the specified character code.

- `TextWriter::Printf, WideTextWriter::Printf`

Uses the Format Expansion buffer to expand the format and then render the text.

- `TextWriter::VPrintf, WideTextWriter::VPrintf`

This is the `va_list` version of `Printf`.

- `TextWriter::Print(const char*, int)`

Renders only the specified number of bytes of the specified text.

- `WideTextWriter::Print(const wchar_t*, int)`

Renders only the specified number of characters of the specified text.

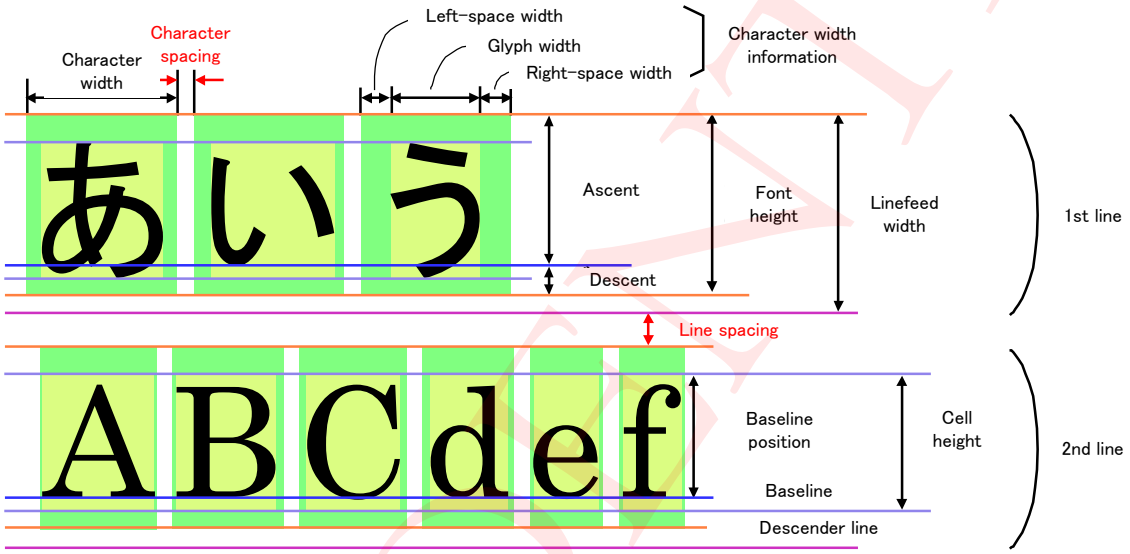
- `TextWriter::Print(const char*), WideTextWriter::Print(const wchar_t*)`

Renders the specified NULL-terminated text.

3.4.3 Overview of Parameters Used When Rendering

Figure 3-3 shows the various parameters that are used during rendering. The green rectangle surrounding each character represents the polygon that actually gets rendered. Items written in black letters are parameters that are specified by the font resource and cannot be changed at runtime. Parameters written in red letters are properties of `TextWriter` and can be set at runtime.

Figure 3-3 Parameters Used for Rendering



4 Character Rendering Features

4.1 Properties of CharWriter

4.1.1 Font

Set and get the font to be used to render the character. If no font is set, no character is rendered even if a character-rendering function is called. Therefore, a font must be set before rendering characters.

4.1.2 Linear Transformation of Glyph Color

Use TEV to perform a linear transformation on each color component of the glyph image.

The linear transformation of the glyph color is not reflected in the display unless TEV is set every time a change is made. For simple changes to character color, use the Character color property described in section 4.1.3.

By default, linear transformation is not performed.

4.1.3 Character Color and Gradation

You can either specify the color of the character being rendered or you can apply a gradation horizontally or vertically to the character.

Character color and gradation effects are implemented through the vertex color of the polygons on which the glyph images are pasted. With intensity fonts, character color is displayed as specified. With RGB fonts, character color is superimposed on the glyph image's original color.

If a linear transformation is applied to the glyph color, the character color is superimposed on the glyph color for intensity fonts as well as for RGB fonts.

Since the character color and the gradation start color share the same variables, you cannot change one without changing the other also.

4.1.4 Character Size

To scale the size of the character being rendered, change the size of the polygons on which texture is pasted.

There are two functions for changing the character size: one specifies the scaling factor and the other specifies the font size after scaling. In both cases, the size is stored internally as a scaling factor.

The function that specifies the after-scaling font size is an auxiliary function that sets the scaling factor for the current font based on the size of the font after scaling.

4.1.5 Texture Interpolation

When the size of a character is changed, the glyph texture is stretched or compressed because the character is being displayed at a size that is different from the original pixel size of the character. You

can specify whether to use the CTR system's texture interpolation feature when changing the size of a character. Note that, if you do not use texture interpolation, enlarging the character enlarges the pixel shapes that are displayed.

By default, texture interpolation is used both when enlarging and when reducing the size of a character.

4.1.6 Monospacing

This feature renders text using the same specified width for all characters, instead of using the various character widths associated with the characters in the font. This feature can be used to temporarily treat a font as if it had monospace characters even though each character in the font normally has a different width.

When you get the character width of a string for monospacing, the value calculated for all the characters uses the specified character width rather than the font's character width.

4.1.7 Cursor

The cursor position indicates the location where the next character is to be rendered.

`CharWriter` maintains the location of the last character rendered. When a Character Rendering member function is called, rendering begins at that location (the cursor position). The cursor position automatically moves to the right of the rendered character in preparation for rendering the next character, thus allowing for the display of a sequence of characters.

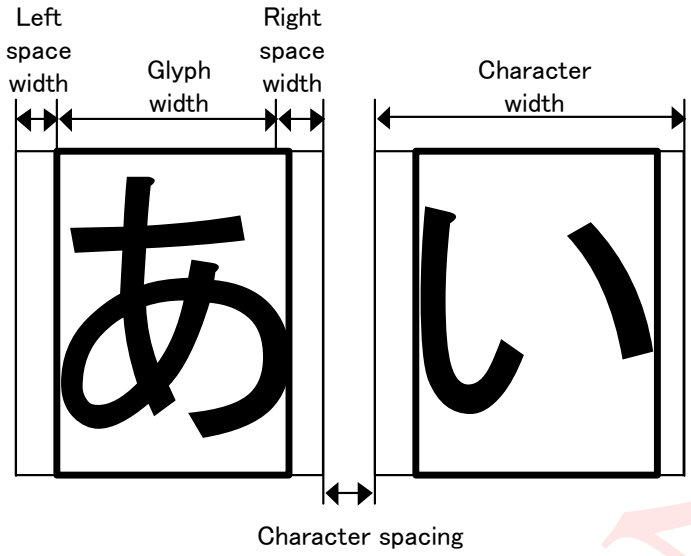
When a position is specified for the rendering of a character, the cursor moves to that position before the character is rendered.

4.2 Properties of TextWriter

4.2.1 Character Spacing

Character spacing specifies the spacing between characters when rendering text. More precisely, character spacing is the distance from the right edge of the right space width of one character to the left edge of the left space width of the next character. See Figure 4-1.

Figure 4-1 Character Spacing

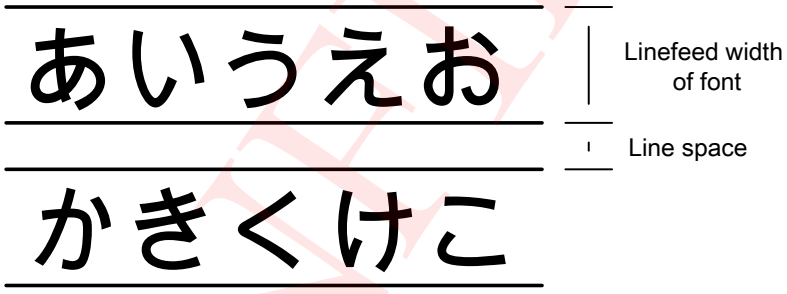


The default character spacing is zero.

4.2.2 Line Space

Line space specifies the spacing between lines when rendering multiple lines of characters. The line height uses the linefeed width of the font. Thus, the distance from the top edge of one line to the top edge of the next line is equal to the font's linefeed width plus the line space. See Figure 4-2.

Figure 4-2 Line Space

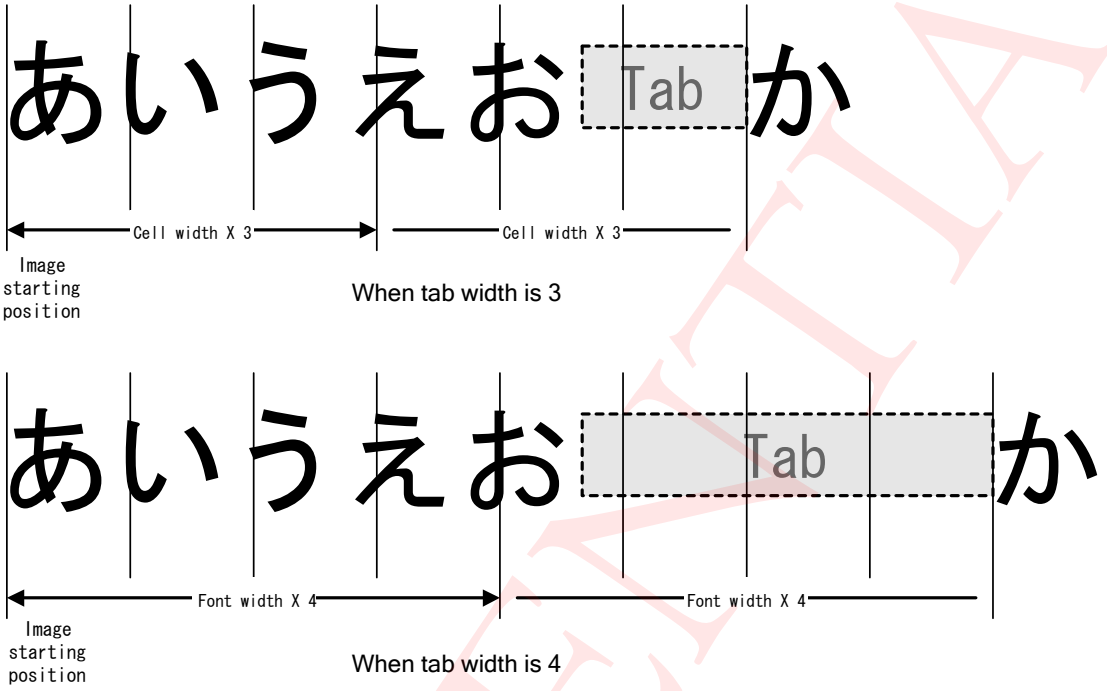


Since the linefeed processing in `TextWriter` is performed by `TagProcessor`, you can create a class derived from `TagProcessor` to override linefeed processing. The default line space is zero.

4.2.3 Tab Width

The tab width specifies how many multiples of the font width to position the character after the tab character. For example, if the tab width is set to 4 characters, the character after the tab is rendered at the position 4*n* pixels over from the current cursor position, where *n* equals the font width.

Figure 4-3 Tab Width



Because the tab character processing in `TextWriter` is performed by `TagProcessor`, you can derive a class from `TagProcessor` to override tab character processing. The default tab width is 4.

4.2.4 Rendering Flags

You can set and get various rendering flags that specify certain behaviors when rendering text.

Below are the definitions for the various flags related to position:

Code 4-1 Rendering Flags

```
enum PositionFlag
{
    //--- Specifies the horizontal alignment for each line when rendering multiple
lines at once
    HORIZONTAL_ALIGN_LEFT = 0x0,          // Align left
    HORIZONTAL_ALIGN_CENTER       = 0x1,          // Align centered
    HORIZONTAL_ALIGN_RIGHT        = 0x2,          // Align right

    //--- Specifies where in the horizontal direction to place the cursor position
within the text
    HORIZONTAL_ORIGIN_LEFT        = 0x00,               // To the left of the text
    HORIZONTAL_ORIGIN_CENTER      = 0x10,               // In the center of the
text
    HORIZONTAL_ORIGIN_RIGHT       = 0x20,               // To the right of the text
```

```
    //--- Specifies where in the vertical direction to place the cursor position
within the text
    VERTICAL_ORIGIN_TOP    = 0x000,        // Top edge of the text
    VERTICAL_ORIGIN_MIDDLE        = 0x100,        // Center of the text
    VERTICAL_ORIGIN_BOTTOM        = 0x200,        // Bottom of the text
    VERTICAL_ORIGIN_BASELINE      = 0x300,        // baseline for line one of text
    };
```

Figure 4-4 shows the changes in how characters are rendered when the values of the rendering flags are changed separately, using `HORIZONTAL_ALIGN_LEFT` | `HORIZONTAL_ORIGIN_LEFT` | `VERTICAL_ORIGIN_TOP` as the base settings.

Figure 4-4 Effect of Rendering Flags

4.2.5 Wrap Width

This specifies the automatic wrap width when rendering long strings. The wrapping feature can also be disabled by setting the wrap width to positive infinity.

The wrapping feature is internally implemented by requesting `¥n` processing to `TagProcessor` at the

place where wrapping should occur. As a result, if the `TagProcessor` configured for `TextWriter` does not treat ¥n as a newline character, the display will not be correct.

4.2.6 TagProcessor

`TagProcessor` specifies a class derived from `TagProcessor` that processes tagged text. You must define such a derived class before rendering tagged text. For more information, see Chapter 5 Rendering Tagged Text.

An instance of a class derived from `TagProcessor` is configured by default.

4.2.7 Format Expansion Buffer

The `TextWriter` text rendering functions that end in "`f`" are capable of formatted output, such as the standard `Printf` function. The internal implementation of formatted output involves first expanding the formatted text in a buffer and then performing normal text rendering. The temporary buffer in which the formatted text is expanded is called the Format Expansion buffer. This buffer must be prepared by the application and allocated to `TextWriter`. The buffer is shared by all instances of `TextWriter`, so you do not need to allocate the Format Expansion buffer for each individual instance.

In addition to the application preparing a buffer and assigning it to be the Format Expansion buffer, `TextWriter` can be configured to allocate a Format Expansion buffer on the stack. If you allocate the buffer on the stack, you must be careful not to cause a stack overflow.

If a Format Expansion buffer is not allocated on the stack, the buffer can be used to expand only one text string at a time. In such a scenario, the `Printf` function cannot be called inside the `Process` member function of a class derived from `TagProcessor` while `Printf` is still sending tagged text to the output.

By default, the buffer is allocated on the stack and can hold up to 256 characters.

5 Rendering Tagged Text

`TextWriter` provides a framework for interpreting and rendering tagged text. This chapter explains how to display your own tagged text.

5.1 Tagged Text

Tagged text is text in which special control symbols called tags are embedded. `TextWriter` can handle tags of any byte sequence provided that the character code of the first character is between 0x0000 and 0x001F. You are free to decide on all other aspects of the tag structure and length.

The tab character and the linefeed character are included within the 0x0000–0x001F range of character codes and are processed as a type of tag by the `TagProcessor` class.

The NULL character can be used as the first character of a tag. However, to prevent the NULL character from being recognized as a text terminator, you need to call `Print(const char*, int)` or `Print(const wchar_t*, int)`.

5.2 TextWriter Framework

The class derived from `TagProcessor` can be incorporated into the framework for rendering tagged text by using the `SetTagProcessor` member function to associate the class derived from `TagProcessor` with `TextWriter`.

When a character code within the range 0x0000–0x001F appears while `TextWriter` is rendering text, `TextWriter` passes the processing of the tag to the `Process` member function of the class derived from `TagProcessor` that has been configured in `TextWriter`. After the processing of the tag has been completed, this derived class informs `TextWriter` of the ending position of the tag, whereupon `TextWriter` resumes rendering text.

5.3 Implementing Classes Derived from TagProcessor

5.3.1 Process Member Function

The purpose of the `Process` member function is to process tags based on the function arguments and pass the results to `TextWriter`. The result of the `Process` member function is the actual result from processing the tag.

The `Process` member function performs all input/output for the `context->writer` passed as function arguments. In other words, it first gets the current rendering state from `context->writer`. Then, after processing the tag, it sets `context->writer` to the state for normal resumption of rendering and returns to `TextWriter`. In order for `TextWriter` of be informed of the position where rendering will resume, this position must be set in `context->str`. When control returns from the `Process` member function, `TextWriter` resumes the rendering of text from the position of `context->str`.

The Y-coordinate of the cursor position for `context->writer` always indicates the baseline,

regardless of the value of the rendering flag.

5.3.2 CalcRect Member Function

The purpose of the `CalcRect` member function is to process tags based on the function argument and then pass the range of effect to `TextWriter`. This function is used to calculate the size of the text rendering as a preliminary step to actual tag processing.

The `pRect` argument takes the region in which rendering will be performed according to the tag. A value does not need to be set in this argument for tags where no rendering occurs, such as tags that change the font.

The `CalcRect` member function generally serves as a substitute for functions like `SetCursor()` for text rendering processes after implementation of the Process member function. Note that, in addition to setting the value in the pRect argument, you also need to configure the `context->writer` properties as is done by the Process member function. If this is not done, subsequent rendering sizes may not be calculated correctly.

For example, assume that one of the tags to be processed changes the character size. You need to change the font size by making a call to `context->writer->SetFontSize()`. If this is not done, the unchanged font size will be used in size calculations for the rendering of subsequent text, leading to unexpected results.

5.3.3 Precautions

5.3.3.1 Format Expansion Buffer

The Format Expansion buffer cannot be used while it is being used elsewhere. Thus, in regards to text rendering functions that use format expansion to render tagged text, you must not use these functions inside the `Process` member function unless they are able to discern when the Format Expansion buffer is not available. Alternatively, you can either allocate the Format Expansion buffer on the stack or you can perform format expansion ahead of time.

5.3.3.2 Tags and Linefeeds

The processing of tag and linefeed characters is done by `TagProcessor`. Consequently, if the class derived from `TagProcessor` does not perform these processes, they will not be done.

If the tag and linefeed processes have not been overridden, the `Process` and `CalcRect` member functions call the member functions in the class derived from `TagProcessor` as shown below. Particular caution is required if using the automatic wrapping feature, because if the implementation of `TagProcessor` does not process `¥n` as a line break, the display will not be correct.

Code 5-1 Overriding the Process Member Function

```
TagProcessor::Operation
MyTagProcessor::Process(u16 code, PrintContext* context)
{
switch( code )
{
```

```
case '<First character of unique tag to process >': <processing>; break;
case '<First character of unique tag to process >': <processing>; break;
case '<First character of unique tag to process >': <processing>; break;
case '<First character of unique tag to process >': <processing>; break;
default:
// Transfer all other tags (those not being processed independently) to the
TagProcessor class
       return TagProcessor::Process(code, context);
      }
}
```

6 Revision History

Version	Revision Date	Description
1.1.0	2010/09/27	Changed the explanations about the method of rendering in chapters 2 and 3.
1.0.0	2010/07/29	Changed the format.
	2010/01/15	Revised the content of figures.
	2009/10/30	Initial version.