ZStrings

Eric Traut
Connectix Corp.
Draft 1.6
Last Modified June 17, 2001 by Eric Traut

Background

The concept of ZStrings is not new. It has been used successfully for a number of years within Adobe's applications. However, we have extended it in several ways to make it more flexible.

This document provides an overview of ZStrings. It doesn't detail all the methods available within the ZString class and its associated classes. For this information, refer to the source code itself.

Issues

ZStrings attempt to address the following issues.

1. It provides a flexible mechanism for manipulating strings. It provides methods for concatenating, extracting and transforming strings in various ways.

2. It addresses differences in string handling between various platforms. The base class itself is designed to be completely cross-platform. And through various escape sequences, it abstracts differences in character encodings between platforms.

3. It provides a flexible way to override one or more strings at runtime, allowing for easy localization. Furthermore, it allows for "simultaneously localized" software – i.e. software that contains all the localized information within a single binary.

4. It provides a set of tools for managing string changes within a binary. String changes are important to track because any change in the English strings must be reflected in the localized versions of these strings.

Named Strings

At the core of ZStrings are the concept of "named strings". Named strings are what they sound like – strings that contain not only the string contents, but also meta-information that names the string uniquely within the program.

Defining a named string within source code is simple. Here's an example:

```
static const char sFileMenuTitle =
   "<Z name=VPC/Menu/File/Title>File</Z>";
```

Note the HTML or XML-like syntax. All named strings are enclosed in a “Z” tag which opens with a <Z> and ends with a </Z>. In addition, all named strings contain a “name=” parameter followed by a path-delimited name. The normal path convention is to use an abbreviated form of the application or module name followed by more and more specific information about the string. All names should be in English ASCII characters and should be delimited by “/” characters.

The current ZString implementation is pretty unforgiving with respect to the above syntax. Do not add extra spaces between the “<” and the “Z”, and only use one space between the “Z” and “name=”. In other words, don’t stray from the above notation, or your code will probably not work.

Additional Parameters

In addition to the name parameter, ZStrings may also have a limit parameter. This parameter sets the maximum allowed length of the string contents, which is calculated so that tags are only counted as one character instead of many. The limit parameter is not mandatory. It offers a way for engineers to specify the number of characters available for the string contents, helping to identify potential formatting problems from localization.

The form of the limit parameter is “limit=” followed by an integer. This parameter is located between the name parameter and the “>”, with at least one space before the limit parameter. For example:

```
static const char sDialogBoxContents =
   "<Z name=Dialog/Options/Error/Text limit=25>That option is "
   "not supported.</Z>";
```

No other additional parameters are supported at this time.

Escape Characters

The named string contents (i.e. the string between the “Z” tag delimiters) is normally straight 7-bit ASCII within the source code. Special characters that are not well-defined by the ASCII standard should be encoded using special escape sequences. These escape sequences were taken directly from the ISO standard adopted by HTML and XML, so they should be familiar to anyone with HTML experience. (See Appendix B for a complete list of escape characters.) Note that all escape sequences are case-sensitive. In fact, many of the sequences differ only in case, so care should be taken to use the correct capitalization.

Escape characters are especially important when localizing for cross-platform modules. The encoding of some non-ASCII characters differs between platforms. ZString escape sequences are correctly translated on each platform. For example, the Mac supports a special “ellipses” character, whereas many fonts in Windows don’t support this character. The ZString parser is smart enough to translate the “&hellip” escape character into an ellipses on the Mac, but three periods on Windows.

Replacement Parameters

In addition to the ISO-standard escape sequences, we have added one of our own to support replacement parameters. The “&replaceXX” sequence (where XX is a two-digit decimal number) can be replaced with another string at runtime. For example:

```
static const char sPrintWarningText =
   "<Z name=Doc/Print/Error/Text>The document &replace00 could "
   "not be printed.</Z>";
```

At runtime, the code could fill in the document name. When localizing strings, the &replaceXX escape sequences should generally be left intact.

Two-byte Character Support

Two-byte scripts present a problem for named strings in that the parser must be able to tell how long each character is. This is especially important when the parser is scanning for escape sequences. If the second byte of a two-byte character is the same as a “&” character, the parser would get confused.

To solve this problem, the parser requires a “multi-byte character map”. This map consists of 256 one-byte entries. On a one-byte system, every entry should be zero. On multi-byte systems, each table entry corresponding to the first character of a multi-byte character should contain the value *n*-1 where *n* is the number of bytes required to encode that character.

Override Dictionaries

Override dictionaries simply consist of a number of named ZString templates concatenated together. However, instead of using the <Z> tags, they use </O> tags. The ZString parser is able to accept either forms, and there is no functional difference between the two. However, a different tag letter was chosen to allow the extraction tool to ignore any override dictionaries when extracting the ZStrings from a binary.

The Localization Process

If an application uses named ZString templates throughout its code, it becomes very easy to localize the application. The application programmer needs to:

1. Write the program to use named ZStrings for all portions of the user interface (including menus, window titles, etc.)
2. Add code at the beginning of the application to load the override dictionary based on a user preference or the current system language.

The application localizer needs to:

1. Run the extraction tool to extract all of the named ZString templates from the application binary file. The extraction tool will search all parts of the binary – including resources, and it will output an HTML-style text file that can be opened within a web browser or an HTML editing program.
2. Spell check and grammar check all of the strings by looking at them within the web browser. Note that the browser will hide the noncontent portion of the named ZString templates (i.e. the <Z> tag) and only show the actual string contents. Furthermore, the browser will correctly interpret and display the meta-tags for special characters (e.g. &hellip will show up as "…" within the browser).
3. Fix any spelling errors within the code. Also, fix any errors or warnings flagged by the extraction tool (e.g. named ZStrings with duplicate names).
4. Once all of the errors have been fixed, the resulting HTML file can be handed to the translators for the various languages. They should be instructed to translate the portion of the text between the <Z> strings. They may want to do the localization with a simple ASCII text editor or use an HTML editor, which will automatically convert special characters back into tags that can be understood by the ZString parser (e.g. if the translator types "ü", the HTML editor will output &uuml).
5. The resulting translation files (still in HTML format) should then be run through the "override dictionary creation" tool. This will produce a platform-specific override dictionary (e.g. resource-based on the Mac, file-based on Windows and Linux). This tool will also flag any errors or warnings encountered during the override generation process.
6. The override dictionary should then be given back to the development engineer or the installer engineer to include in the application binary or installer package.

ZString Implementation Details

The ZString implementation consists of a set of cross-platform classes: ZString, ZStringData, ZStringParser, ZStringDictionary, ZStringTool.

The ZString class is a flexible string class. It doesn't contain the actual string data itself. Rather, it contains a reference to a ZStringData object. If this reference is NULL, the string is assumed to be empty.

A ZStringData object contains the string data along with a reference count. The reference count allows a single ZStringData object to be shared between multiple ZString objects. This also allows ZStrings to be copied very quickly (in a the time it takes to do a pointer copy and a reference count increment). When a ZString's destructor is called, the reference count for the ZStringData is decremented. And if the decrement count hits zero, the ZStringData is also disposed.

In general, a ZString object can be treated just like any intrinsic type. They are small enough that they can be passed by value or embedded in other objects. They don't contain any virtual methods, so there's no v-table dispatching speed penalty in their use. See below for usage examples.

The ZStringParser is a singleton class that must be subclassed on each platform. It is responsible for parsing named strings into the name and string portion. It's also responsible for finding the escape sequences and replacing them with the correct platform-specific character encodings. Finally, it maintains the multi-byte character table used for two-byte character support.

The ZStringDictionary class is also a singleton class. There is no reason to override it, as it is completely cross-platform. It maintains a dictionary of all named strings encountered so far. This is useful for two reasons. First, once a named string is parsed, it doesn't need to be retranslated each time it is used. Second, all of the named strings can be overridden at application launch time with localized versions of the strings.

The ZStringTool class is generally not used within the application source code itself. It provides the core implementation of tools to be used with ZStrings. The three most important tools are:

1. A ZString extraction tool. This tool scans a specified input file for any embedded named strings and prints out a list for the localization team.

2. A ZString comparison tool. This tool scans two files, a new file and an older binary or override file, to determine what strings have changed since the application was last localized.

3. A ZString override dictionary generation tool. This tool takes a file similar to the output of the extraction tool and creates an override dictionary that can be loaded at runtime.

For more information about the options available for these tools, please see the ZStringTool Options document.

ZString Usage

The ZString type contains a variety of operator override methods to do concatenation and casting. For example, you can add a C string to a ZString or add two ZStrings together. Here are a few examples:

```
// Two different initialization forms.
ZString  stringA = "Hello";
ZString  stringB("World");
```

```
// Unary addition operator (defined for types ZString,
// const char * and char.
stringA += ' ';

// Binary addition operator (defined for types ZString
// const char * and char.
ZString  stringC = stringA + stringB;

// Casting operator (defined for type const char *).
const char * cString = (char *)stringC;
```

Some conversions and operations require explicit method calls. For example:

```
// Look up a named string in the dictionary, adding
// it if necessary.
stringA.GetNamedString(sPrintWarningText);

// Replace a parameter within the string.
stringA.ReplaceParameter(0, docName);

// Convert to a Pascal string.
stringA.GetPString(pString, sizeof(pString));

// Set from a Pascal string.
stringA.SetPString(pString);

// Extract a delimited substring.
stringB = stringA.GetSubstring(3, '/');

// Determining string length.
length = stringA.GetLength();

// Freeing data associated with a ZString.
stringA = "";
```

Porting ZStrings

The source code for ZStrings is highly portable C++ code. It has been compiled on a half a dozen compilers, and has been ported to three platforms: Macintosh, Windows, and Linux. Ports to other platforms should require very little work.

Tools

Please refer to the separate document “ZStringTool-Documentation” for information about using the tools that allow for extraction, comparison, and the creation of override dictionaries.

Appendix A: Platform-Specific ZString Issues

Macintosh

The Mac makes extensive use of Pascal strings in its user interface toolbox. For that reason, the MacZString subclass contains an additional static method for dealing with Pascal strings. This method, GetNamedPString, allows the caller to look up a named string and convert it to Pascal in one call.

ZString override libraries on the Mac will generally be stored as resources. By convention, we will use the following resource types and IDs:

ZOvr - resource type for override dictionaries
ZTBt - resource type for two-byte character table overrides
ZLan - language specifier resource (always use ID 128)

128 ($0080) - resource ID for English override (normally unused)
129 ($0081) - resource ID for German override
130 ($0082) - resource ID for French override
131 ($0083) - resource ID for Japanese override
132 ($0084) - resource ID for Italian override
133 ($0085) - resource ID for Spanish override
134 ($0086) - resource ID for Portuguese override

By default, the ZString system will choose the same language as the system software. For example, if the MacOS is in German, the ZString initialization code will automatically load the German override dictionary. However, a program can be "locked in" to a particular language, overriding the default, using the "ZLan" 128 resource. This allows a program running on an English system, for example, to be displayed in German. A ZLan resource should contain a two-byte value that represents the resource ID for the desired language (see table above). For example, if you want to "lock in" German, the ZLan resource should consist of two bytes:$0081 (remember that ResEdit typically displays resource contents in hexadecimal).

Another issue associated with simultaneous localization of text involves font selection. We've traditionally used 'finf' (font info) resources to indicate the typeface, size and style for fonts. These can now be overridden by language. The standard English 'finf's are stored in their traditional location, starting at resource ID 1800. Overrides should be stored at the following base IDs:

128 ($0080) – base 'finf' resource ID for English
144 ($0090) – base 'finf' resource ID for German
160 ($00A0) – base 'finf' resource ID for French
176 ($00B0) – base 'finf' resource ID for Japanese
192 ($00C0) – base 'finf' resource ID for Italian
208 ($00D0) – base 'finf' resource ID for Spanish
224 ($00E0) – base 'finf' resource ID for Portuguese

In other words, the base address is calculated by taking the override ID (defined above), subtracting 128, then multiplying by 16, then adding back 128. The reason these values

differ by 16 is so that we can define up to 16 different ‘finf’s for each language. Currently, only five are defined inside of the AppShell framework used in many Mac programs at Connectix. They are listed below with their offset from the base ID above (e.g. the bold application font for German is 144 + 2 = 146). Note that if a particular ‘finf’ is missing, the code will fall back on the default (English) ‘finf’ resource.

0 – normal system font (12 point Charcoal)
1 – normal application font (9 point Geneva)
2 – bold application font (9 point Geneva)
3 – bigger normal application font (10 point Geneva)
4 – bigger bold application font (10 point Geneva)

If you store named string templates in resources, make sure to either lock down the resource before entering the string into the dictionary. The ZString classes assume that all named string templates are constant, static and persistent. If the template you’re using is not persistent, specify “true” for the “inDataIsVolatile” parameter to the GetNamedString method. This will cause the ZString dictionary code to make a persistent copy of the named string template.

The MacZString class also contains methods for building menus from ZString data, allowing them to be easily localized. They are used in conjunction with static data structures that define the layout of the menu. For example:

```
static const char sFileMenuTitleText[] = "<Z name=
Menu/Title/File>File</Z>";
static const char sCloseMenuItemText[] = "<Z name=
Menu/Item/Close>Close/W</Z>";
static const char sQuitMenuItemText[] = "<Z name=
Menu/Item/Quit>Quit/Q</Z>";

static const char *           sFileMenuInfo[] =
      {
            sCloseMenuItemText,
            "",
            sQuitMenuItemText,
            NULL
      };

Str255 menuItemString;
MacZString::GetNamedPString(sFileMenuTitleText, menuItemString);
MacZString::BuildMenu(::NewMenu(kFileMenuID, menuItemString),
sFileMenuInfo);
```

Note the metacharacters in the menu item names that are used to define the text for command-key equivalent.

Windows

Override dictionaries on Windows can be stored as either resources within the application binary or as separate override dictionary files. Therefore, the Win32ZString subclass

contains two forms of the method LoadOverrideDictionary – one that loads a resource and one that loads a file.

Linux

Override dictionaries are assumed to be stored as individual files within Linux.

Appendix B: Escape Character List

The following metacharacters are defined. Their names are taken directly from the HTML 4.0 standard which uses ISO standard names for its character sets. Most of the metacharacter symbols start with ampersands, but there are variants that are enclosed in HTML-style brackets ("<" and ">").

Typography

		Non-breaking space
&bull	•	Bullet
&ndash	–	En dash
&mdash	—	Em dash
 		Line break

Punctuation

&iexcl	¡	Inverted exclamation point (Spanish)
&iquest	¿	Inverted question mark
&hellip	…	Horizontal ellipsis
&lsquo	‘	Left single quotation mark
&rsquo	’	Right single quotation mark
&sbquo	‚	Single low-9 quotation mark
&ldquo	“	Left double quotation mark
&rdquo	”	Right double quotation mark
&bdquo	„	Double low-9 quotation mark

Symbols

&cent	¢	Cent symbol (American)
&pound	£	Pound sign (British)
&yen	¥	Yen symbol (Japanese)
&copy	©	Copyright symbol
&reg	®	Registered trademark
&trade	™	Trade mark sign
&micro	µ	Micro sign (Greek mu)

&para	¶		Paragraph symbol
&pi	π		Greek small letter pi
&amp	&		Ampersand
&lt	<		Less than
&gt	>		Greater than

Foreign Characters

&Agrave	À		Capital A, grave accent
&Aacute	Á		Capital A, acute accent
&Acirc		Â	Capital A, circumflex accent
&Atilde	Ã		Capital A, tilde
&Auml	Ä		Capital A, umlaut
&Aring	Å		Capital A, ring
&AElig	Æ		Capital AE ligature
&Ccedil	Ç		Capital C, cedilla
&Egrave	È		Capital E, grave accent
&Eacute	É		Capital E, acute accent
&Ecirc	Ê		Capital E, circumflex accent
&Euml	Ë		Capital E, umlaut
&Igrave	Ì		Capital I, grave accent
&Iacute	Í		Capital I, acute accent
&Icirc	Î		Capital I, circumflex accent
&Iuml	Ï		Capital I, umlaut
&Ntilde	Ñ		Capital N, tilde
&Ograve	Ò		Capital O, grave accent
&Oacute	Ó		Capital O, acute accent
&Ocirc		Ô	Capital O, circumflex accent
&Otilde	Õ		Capital O, tilde
&Ouml	Ö		Capital O, umlaut
&Oslash	Ø		Capital O, slash
&Ugrave	Ù		Capital U, grave accent
&Uacute	Ú		Capital U, acute accent
&Ucirc		Û	Capital U, circumflex accent
&Uuml	Ü		Capital U, umlaut
&szlig	ß		Small sz ligature, German
&agrave	à		Lowercase a, grave accent
&aacute	á		Lowercase a, acute accent
&acirc	â		Lowercase a, circumflex accent
&atilde		ã	Lowercase a, tilde
&auml	ä		Lowercase a, umlaut
&aring	å		Lowercase a, ring
&aelig	æ		Lowercase ae ligature

&ccedil	ç	Lowercase c, cedilla
&egrave	è	Lowercase e, grave accent
&eacute	é	Lowercase e, acute accent
&ecirc	ê	Lowercase e, circumflex accent
&euml	ë	Lowercase e, umlaut
&igrave	ì	Lowercase i, grave accent
&iacute	í	Lowercase i, acute accent
&icirc	î	Lowercase i, circumflex accent
&iuml	ï	Lowercase i, umlaut
&ntilde	ñ	Lowercase n, tilde
&ograve	ò	Lowercase o, grave accent
&oacute	ó	Lowercase o, acute accent
&ocirc	ô	Lowercase o, circumflex accent
&otilde	õ	Lowercase o, tilde
&ouml	ö	Lowercase o, umlaut
&oslash	ø	Lowercase o, slash
&ugrave	ù	Lowercase u, grave accent
&uacute	ú	Lowercase u, acute accent
&ucirc	û	Lowercase u, circumflex accent
&uuml	ü	Lowercase u, umlaut
&yuml	ÿ	Lowercase y, umlaut

The following metacharacters have been added for our own use:

Replacement	
&replaceNN	Text replacement (NN is two-digit decimal number)

Alternatively, a numeric form of the escape characters can be used within named strings. The numeric form uses an ampersand (“&”) followed by a pound sign (“#”) followed by three decimal digits followed by a semicolon (“;”). For example, “&hellip” is equivalent to “…”. Either form is supported by the ZString parser. The extraction tool by default converts all internal name strings to numeric form before writing them to the output file.

The &replaceNN metacharacter has no numeric equivalent and is therefore left as-is by the extraction tool.

Please note that if you use an HTML converter in to convert the special characters to there character code in Z strings, it might change the &replace function to &ampreplace. You will have to manually fix this by doing a search and replace.