Text and Localization

Exploring Palm OS®

Written by Jean Ostrem
Edited by Christopher Bey
Technical assistance from Chris Schneider, Ken Krugler, Jason Parks, Scott Fisher, and JB Parrett

Exploring Palm OS: Text and Localization
Document Number 3111-003
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

Part III: Appendixes

About This Document

This book describes how to write easily localizable code for Palm OS®. Different countries represent characters, strings, numbers, and dates in different ways. This book describes how to write code that does not make assumptions about the representations of these items and runs properly for all languages that Palm OS supports.

This book does *not* cover the following:

- How to work with text fields or display text on the screen. See the "Displaying Text" chapter of *Exploring Palm OS: User Interface* for that information.
- The tools used to localize an application. Consult the documentation specific to your toolset for that information.
- How to work with or write a front-end processor (FEP) for text entry. Such material is described in *Exploring Palm OS: Creating a FEP*.

IMPORTANT: The *Exploring Palm OS* series is intended for developers creating native applications for Palm OS Cobalt. If you are interested in developing applications that work through PACE and that also run on earlier Palm OS releases, read the latest versions of the *Palm OS Programmer's API Reference* and *Palm OS Programmer's Companion* instead.

Who Should Read This Book

You should read this book if you are a Palm OS software developer who writes applications, libraries, pinlets, or any other type of program that manipulates text strings. Even if you do not plan to localize your application, it is still a good idea to follow the recommendations in this book. It will save you time if your plans change later.

Because virtually all Palm OS developers require some knowledge of how to work with text strings, this book is intended for developers with all levels of familiarity with Palm OS, from novice to expert. Novice programmers should first read *Exploring Palm OS: Programming Basics* to gain an understanding of the basic structure of a Palm OS application.

Expert Palm OS programmers will find that much of the material in this book is familiar and may want to just skim it. Differences between the Palm OS Garnet API set and the Palm OS Cobalt API set are outlined in *Exploring Palm OS: Porting Applications to Palm OS Cobalt*.

What This Book Contains

This book contains the following information:

- Part I contains conceptual information and how-to information.
  - Chapter 1, "Text," on page 3 describes how to use the Palm OS managers that help you work with text strings: the Text Manager and the String Manager.
  - Chapter 2, "Implementing Global Find," on page 19 describes how to integrate your application into the Palm OS Global Find facility.
  - Chapter 3, "Localized Applications," on page 27 describes how to use other managers, such as the Locale Manager, that help you write a locale-independent application.
- Part II contains reference information organized into the following chapters:
  - Chapter 4, "Find," on page 39 describes structures, types, and functions used when implementing the Global Find facility.
  - Chapter 5, "Locale Manager Types," on page 51 describes structures and types used in the Locale Manager.
  - Chapter 6, "Locale Manager," on page 59 describes Locale Manager functions.
  - Chapter 7, "String Manager," on page 69 describes the String Manager.
  - Chapter 8, "Text Manager," on page 85 describes the Text Manager.

- Appendix A, "Language-specific Information," on page 135 contains implementation-specific details for some of the languages to which Palm OS is localized. Read it if you are translating your application to one of those languages.

Changes to This Book

3111-003

- Updated list of supported source and destination encodings in `TxtConvertEncoding()`.

3111-002

- Clarified text throughout the document and corrected code samples in Listing 1.2, Listing 1.5, Listing 1.8, Listing 2.2, and in the `TxtGetNextChar()` function description.
- Corrected time zone information.
- The `charEncodingDstBestFitFlag` for `TxtConvertEncoding()` is always supported in Palm OS Cobalt. The `textMgrStrictFlag` in the Text Manager feature constant is no longer used.
- Corrected description of `TxtNameToEncoding()` parameter.

3111-001

- Initial version

The *Exploring Palm OS* Series

This book is a part of the *Exploring Palm OS* series. Together, the books in this series document and explain how to use the APIs exposed to third-party developers by the fully ARM-native versions of Palm OS, beginning with Palm OS Cobalt. Each of the books in the *Exploring Palm OS* series explains one aspect of the Palm operating system and contains both conceptual and reference documentation for the pertinent technology.

As of this writing, the complete *Exploring Palm OS* series consists of the following titles:

- *Exploring Palm OS: Programming Basics*

- *Exploring Palm OS: Memory, Databases, and Files*
- *Exploring Palm OS: User Interface*
- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Porting Applications to Palm OS Cobalt*

Additional Resources

- Documentation

  PalmSource publishes its latest versions of documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/
- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training
- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/support/kb/

Part I
Concepts

This part contains conceptual information for the text and localization managers. It covers:

1

Text

This chapter describes how to work with text in the user interface—whether it's text the user has entered or text that your application has created to display on the screen. When you work with text, you must take special care to do so in a way that makes your application easily localizable. This chapter describes how to write code that manipulates characters and strings in such a way that it works properly for any language that is supported by Palm OS®. It covers:

Character Encodings

Computers represent the characters in an alphabet with a numeric code. The set of numeric codes for a given alphabet is called a **character encoding**. Of course, a character encoding contains more than codes for the letters of an alphabet. It also encodes punctuation, numbers, control characters, and any other characters deemed necessary. The set of characters that a character encoding represents is called a **character set**.

Different languages use different alphabets. Most European languages use the Latin alphabet. The Latin alphabet is relatively small, so its characters can be represented using a single-byte encoding ranging from 32 to 255. On the other hand, Asian languages such as Chinese, Korean, and Japanese require their own alphabets, which are much larger. These larger character sets are represented by a combination of single-byte and double-byte numeric codes ranging from 32 to 65,535.

Although Palm OS supports multiple character encodings, only one of these encodings is active at a time. For example, a French device uses the Palm OS Latin encoding, which is identical to the Microsoft Windows code page 1252 character encoding (an extension of ISO

Latin 1) but includes Palm-specific characters in the control range. A Japanese device, on the other hand would use the Palm OS Shift JIS character encoding, which is identical to Microsoft Windows code page 932 (an extension of Shift JIS) but includes Palm-specific characters in the control range. These two devices use different character encodings even though they both use the same version of Palm OS.

No matter what the encoding is on a device, PalmSource guarantees that the low ASCII characters (0 to 0x7F) are the same. The exception to this rule is 0x5C, which is a yen symbol on Japanese devices and a backslash on most others.

The Palm OS Text Manager allows you to work with text, strings, and characters independent of the character encoding. If you use Text Manager functions and don't work directly with string data, your code should work on any system, regardless of which language and character encoding the device supports.

Characters

Depending on the device's supported languages, Palm OS may encode characters using either a single-byte encoding or a multi-byte encoding. Because you do not know which character encoding is used until runtime, *you should never make an assumption about the number of bytes a character occupies in a string*.

For the most part, your application does not need to know which character encoding is used, and in fact, it should make no assumptions about the encoding or about the size of characters. Instead, your code should use Text Manager functions to manipulate characters. This section describes how to work with characters correctly. It covers:

Declaring Character Variables

Declare all character variables to be of type `wchar32_t`. `wchar32_t` is a 32-bit unsigned type that can accommodate characters of any encoding. Don't use `char`. `char` is an 8-bit variable that cannot accommodate larger character encodings.

```
wchar32_t ch; // Right. 32-bit character.
char ch; // Wrong. 8-bit character.
```

When you receive input characters through the `keyDownEvent`, you'll receive a `wchar32_t` value. (That is, the `data.keyDown.chr` field is a `wchar32_t`.)

While character variables are declared as `wchar32_t`, string variables are still declared as `char *`, even though they may contain multi-byte characters. See the section "Strings" for more information on strings.

Using Character Constants

Character constants are defined in several header files. The header file `Chars.h` contains characters that are guaranteed to be supported on all systems regardless of the encoding. Other header files exist for each supported character encoding and contain characters specific to that encoding. The character encoding-specific header files are not included in the `PalmOS.h` header by default because they define characters that are not available on every system.

To make it easier for the compiler to find character encoding problems with your project, make a practice of using the character constants defined in these header files rather than directly assigning a character variable to a value. For example, suppose your code contained this statement:

```
wchar32_t ch = 'å'; // WRONG! Don't use.
```

This statement may work on a Latin system, but it would cause problems on an Asian-language system because the å character does not exist. If you instead assign the value this way:

```
wchar32_t ch = chrSmall_A_RingAbove;
```

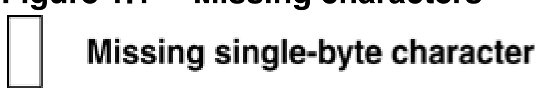
you'll find the problem at compile time because the `chrSmall_A_RingAbove` constant is defined in `CharLatin.h`, which is not included by default.

Missing and Invalid Characters

If during application testing, you see an open rectangle displayed on the screen, you have a missing character.

A **missing character** is one that is valid within the character encoding but the current font is not able to display it. In this case, nothing is wrong with your code other than you have chosen the wrong font. The system displays an open rectangle in place of a missing single-byte character (see Figure 1.1).

Figure 1.1 Missing characters



In multi-byte character encodings, a character may be missing as described above, or it may be invalid. In single-byte character encodings, there's a one-to-one correspondence between numeric values and characters to represent. This is not the case with multi-byte character encodings. In multi-byte character encodings, there are more possible values than there are characters to represent. Thus, a character variable could end up containing an **invalid character**—a value that doesn't actually represent a character.

If the system is asked to display an invalid character, it prints an open rectangle for the first invalid byte. Then it starts over at the next byte. Thus, the next character displayed and possibly even the remaining text displayed is probably not what you want. Check your code for the following:

- Truncating strings. You might have truncated a string in the middle of a multi-byte character.
- Appending characters from one encoding set to a string in a different encoding.

- Arithmetic on character variables that could result in an invalid character value.
- Arithmetic on a string pointer that could result in pointing to an intra-character boundary. See "Performing String Pointer Manipulation" for more information.
- Use of standard C string functions. Many of these functions are not multi-byte aware and can return invalid results for strings that contain multi-byte characters.
- Assumptions that a character always occupies only one byte in a string.

Use the Text Manager function `TxtCharIsValid()` to determine whether a character is valid or not.

Retrieving a Character's Attributes

The Text Manager defines certain functions that retrieve a character's attributes, such as whether the character is alphanumeric, and so on. You can use these functions on any character, regardless of its size and encoding.

A character also has attributes unique to its encoding. Functions to retrieve those attributes are defined in the header files specific to the encoding.

Virtual Characters

Virtual characters are nondisplayable characters that trigger special events in the operating system, such as displaying low battery warnings or displaying the keyboard dialog. Virtual characters should never occur in any data and should never appear on the screen.

The Palm OS uses character codes 256 decimal and greater for virtual characters. The range for these characters may actually overlap the range for "real" characters (characters that should appear on the screen). The `keyDownEvent` distinguishes a virtual character from a displayable character by setting the `commandKeyMask` bit in the structure's `modifiers` field.

The best way to check for virtual characters, including virtual characters that represent the hard keys, is to use the `TxtCharIsVirtual()` function. See Listing 1.1.

Listing 1.1 Checking for virtual characters

```
if (TxtCharIsVirtual (eventP->data.keyDown.modifiers,
   eventP->data.keyDown.chr)) {
   if (TxtCharIsHardKey (event->data.keyDown.modifiers,
      event->data.keyDown.chr)) {
      // Handle hard key virtual character.
   } else {
      // Handle standard virtual character.
   }
} else {
   // Handle regular character.
}
```

Retrieving the Character Encoding

Occasionally, you may need to determine which character encoding is being used. For example, your application might use specifically optimized code when it's being run on a device that uses the Palm OS Latin character encoding. You can retrieve the character encoding using the `LmGetSystemLocale()` function as shown in Listing 1.2.

Listing 1.2 Retrieving the character encoding

```
CharEncodingType encoding;
char* encodingName;

encoding = LmGetSystemLocale(NULL);
if (encoding == charEncodingPalmSJIS) {
   // encoding for Palm Shift JIS
} else if (encoding == charEncodingPalmLatin) {
   // extension of ISO Latin 1
} else {
   // Note: Palm OS licensees may add support for other
   // character encodings.
}
```

```
// The following Text Manager function returns the
// official name of the encoding as required by
// Internet applications.
encodingName = TxtEncodingName(encoding);
```

Strings

Strings are made up of characters that occupy from one to four bytes each. As stated previously, the standard character variable, `wchar32_t`, is four bytes long. However, when you add a character to a string, the operating system may shrink it down to a single byte if it's a low ASCII character. Thus, any string that you work with may contain a mix of single-byte and multi-byte characters.

When working with text as strings, you can use any of the following:

- Standard C Library string functions

  Palm OS Cobalt supports the standard C library including the standard C string functions. These functions *only* manipulate strings containing single-byte characters. Do not use these functions on strings that may contain multi-byte characters.

  For example, if your application displays a numeric text field in which the user may enter some sort of application setting, it's acceptable to manipulate the string that you receive from the numeric text field using the standard C library calls. If a string may contain letters, you should use String Manager or Text Manager calls to make your application easily localizable.

- The String Manager

  The String Manager is closely modeled after the standard C library functions like `strcpy()`, `strcat()`, and so on. In some cases, the String Manager functions call through to their standard C library counterparts. In other cases, the String Manager function has been modified to become multi-byte aware.

- The Text Manager

  The Text Manager specifically provides support for multi-byte strings. Use the Text Manager functions when:

  - A String Manager equivalent is not available.
  - The length of the matching strings are important. For example, to compare two strings, you can use either `StrCompare()` or `TxtCompare()`. The difference between the two is that `StrCompare()` does not return the length of the characters that matched. `TxtCompare()` does.

This section discusses the following topics:

TIP: All Palm OS functions that return the length of a string, such as `FldGetTextLength()` and `StrLen()`, always return the size of the string in bytes, not the number of characters in the string. Similarly, functions that work with string offsets always use the offset in bytes, not characters.

Manipulating Strings

Any time that you want to work with character pointers, you need to be careful not to point to an intra-character boundary (a middle or end byte of a multi-byte character). For example, any time that you want to set the insertion point position in a text field or set the text field's selection, you must make sure that you use byte offsets that point to inter-character boundaries. (The **inter-character boundary** is both the start of one character and the end of the previous character, except when the offset points to the very beginning or very end of a string.)

Suppose you want to iterate through a string character by character. Traditionally, C code uses a character pointer or byte counter to iterate through a string a character at a time. Such code will not work properly on systems with multi-byte characters. Instead, if you want to iterate through a string a character at a time, use Text Manager functions:

- TxtGetNextChar() retrieves the next character in a string.
- TxtGetPreviousChar() retrieves the previous character in a string.
- TxtSetNextChar() changes the next character in a string and can be used to fill a string buffer.

Each of these three functions returns the size of the character in question, so you can use it to determine the offset to use for the next character. For example, Listing 1.3 shows how to iterate through a string character by character until a particular character is found.

Listing 1.3 Iterating through a string or text

```
char* buffer; // assume this exists
size_t bufLen = StrLen(buffer);
// Length of the input text.
wchar32_t ch = 0;
size_t i = 0;
while ((i < bufLen) && (ch != chrAsterisk))
   i+= TxtGetNextChar(buffer, i, &ch));
```

The Text Manager also contains functions that let you determine the size of a character in bytes without iterating through the string:

- TxtCharSize() returns how much space a given character will take up inside of a string.
- TxtCharBounds() determines the boundaries of a given character within a given string.

Listing 1.4 Working with arbitrary limits

```
size_t charStart, charEnd;
char *fldTextP = FldGetTextPtr(fld);
TxtCharBounds(fldTextP,
   min(kMaxBytesToProcess, FldGetTextLength(fld)),
   &charStart, &charEnd);
// process only the first charStart bytes of text.
```

Performing String Pointer Manipulation

Never perform any pointer manipulation on strings you pass to the Text Manager unless you use Text Manager calls to do the manipulation. For Text Manager functions to work properly, the string pointer must point to the first byte of a character. If you use Text Manager functions when manipulating a string pointer, you can be certain that your pointer always points to the beginning of a character. Otherwise, you run the risk of pointing to an inter-character boundary.

Listing 1.5 String pointer manipulation

```
// WRONG! buffer + kMaxStrLength is not
// guaranteed to point to start of character.
buffer[kMaxStrLength] = '\0';

// Right. Truncate at a character boundary.
size_t offset = TxtGetTruncationOffset(buffer,
   kMaxStrLength);
buffer[offset] = chrNull;
```

Truncating Displayed Text

If you're performing drawing operations, you often have to determine where to truncate a string if it's too long to fit in the available space. Several functions help you perform this task on strings with multi-byte characters:

- `WinDrawTruncChars()` — This function draws a string within a specified width, determining automatically where to truncate the string. If it can, it draws the entire string. If the string doesn't fit in the space, it draws one less than the number of characters that fit and then ends the string with an ellipsis (...).

  Note, however, that the Window Manager drawing functions are deprecated for many uses and should not be mixed with the Palm OS Cobalt graphics context functions.

- `FntTruncateString()` — This function performs the same task as `WinDrawTruncChars()` except that it does not draw the text to the screen. You might use this if you are using a bitmapped font to display the text drawn using the Palm OS Cobalt graphics context functions. See Listing 1.6.

Listing 1.6 Drawing multiple lines of text in a bitmapped font

```
fcoord_t y;
char *msg, *dstMsg;
size_t pixelWidth = 160;
GcHandle gc = GcGetCurrentContext();
Boolean truncated = false;

FntSetFont(stdFont);
GcSetFont(gc, GcCreateFontFromID(stdFont));
dstMsg = (char *)malloc(StrLen(msg)+1);
truncated = FntTruncateString(dstMsg, msg, FntGetFont(),
   pixelWidth, true);
GcDrawTextAt(gc, 0.0, y, dstMsg, StrLen(dstMsg));
GcReleaseContext(gc);
```

- `GcFontStringBytesInWidth()` — This function works with scalable fonts. It returns the size in bytes of the substring that can be displayed in a specified width.

  Listing 1.7 shows how to use `GcFontStringBytesInWidth()` to determine how many lines are necessary to write a string to the screen (without considering word wrapping). This example passes the width of the screen as the pixel position so that upon return, `widthToOffset` contains the byte offset of the last character in the string that can be displayed on a single line. The characters up to and including the one at `widthToOffset` are drawn, then the `msg` pointer is advanced in the string by `widthToOffset` characters, and `GcFontStringBytesInWidth()` is again called to find out how many characters fit on the next line of text. The process is repeated until all of the characters in the string have been drawn.

Listing 1.7 Drawing multiple lines of text in scalable font

```
fcoord_t y;
char *msg;
size_t widthToOffset = 0;
size_t pixelWidth;
size_t msgLength = StrLen(msg);
GcHandle gc = GcGetCurrentContext();
GcFontHandle standardFont = GcCreateFont("palmos-plain");
FontHeightType fontHeight;
```

```
RectangleType winBounds;

// Set the pixel offset to the width of the screen.
// The scalable font functions expect native coordinates.
WinSetCoordinateSystem(kCoordinatesNative);
GcSetCoordinateSystem(gc, kCoordinatesNative);
WinGetWindowBounds(&winBounds);
pixelWidth = winBounds.extent.x;

GcGetFontHeight(standardFont, &fontHeight);
GcSetFont(gc, standardFont);

// Begin drawing the string to the screen.
while (msg && *msg) {
   widthToOffset = GcFontStringBytesInWidth(standardFont,
      msg, pixelWidth);
   GcDrawTextAt(gc, 0.0, y, msg, widthToOffset);
   y = y + fontHeight.ascent + fontHeight.descent +
      fontHeight.leading;
   msg += widthToOffset;
   msgLength = StrLen(msg);
}
GcReleaseContext(gc);
```

Comparing Strings

Use the Text Manager functions `TxtCompare()` and `TxtCaselessCompare()` to perform comparisons of localizable strings.

In character encodings that use multi-byte characters, some characters have both single-byte and double-byte representations. One string might use the single-byte representation and another might use the multi-byte representation. Users expect the characters to match regardless of how many bytes a string uses to store that character. `TxtCompare()` and `TxtCaselessCompare()` can accurately match single-byte characters with their multi-byte equivalents.

Because a single-byte character might be matched with a multi-byte character, two strings might be considered equal even though they have different lengths. For this reason, `TxtCompare()` and `TxtCaselessCompare()` take two parameters in which they pass back the length of matching text in each of the two strings. See their function descriptions for more information.

The String Manager functions `StrCompare()` and `StrCaselessCompare()` are equivalent to `TxtCompare()` and `TxtCaselessCompare()`, but they do not pass back the length of the matching text.

These Text Manager and String Manager comparison routines use text tables for comparisons, and they are potentially slow. If you want to compare strings that you know contain only 7-bit ASCII characters (for example, the strings are completely internal to the program and never appear in the user interface), use the standard C library functions such as `strcmp()` instead.

A special case of performing string comparison is implementing the Global Find facility. For more information on implementing this feature in your application, see Chapter 2, "Implementing Global Find," on page 19.

Dynamically Creating String Content

When working with strings in a localized application, you never hard code them. Instead, you store strings in a resource and use the resource to display the text. If you need to create the contents of the string at runtime, store a template for the string as a resource and then substitute values as needed.

For example, consider the Edit view of the Memo application. Its title bar contains a string such as "Memo 3 of 10." The number of the memo being displayed and the total number of memos cannot be determined until runtime.

To create such a string, use a template resource and the Text Manager function `TxtParamString()`. `TxtParamString()` allows you to search for the sequence ^0, ^1, up to ^3 and replace each of these with a different string. If you need more parameters, you can use `TxtReplaceStr()`, which allows you to replace up to ^9; however, `TxtReplaceStr()` only allows you to replace one of these sequences at a time.

In the Memo title bar example, you'd create a string resource that looks like this:

```
Memo ^0 of ^1
```

And your code might look like this:

Listing 1.8 Using string templates

```
static void EditViewSetTitle (void)
{
   char* titleTemplateP;
   FormPtr frm;
   char posStr[maxStrIToALen+1];
   char totalStr[maxStrIToALen+1];
   uint16_t pos;
   uint16_t length;

   // Format as strings, the memo's postion within
   // its category, and the total number of memos
   // in the category.
   pos = DmGetPositionInCategory(MemoPadDB, CurrentRecord,
      RecordCategory);
   StrIToA (posStr, pos+1);

   if (MemosInCategory == memosInCategoryUnknown)
      MemosInCategory = DmNumRecordsInCategory
         (MemoPadDB, RecordCategory);
   StrIToA (totalStr, MemosInCategory);

   // Get the title template string.  It contains ^0 and ^1
   // chars which we replace with the position of
   // CurrentRecord within CurrentCategory and with the total
   // count of records in CurrentCategory ().
   titleTemplateP = MemHandleLock (gAppDbP, DmGetResource
      (gAppDbP, strRsc, EditViewTitleTemplateStringString));

   EditViewTitlePtr = TxtParamString(titleTemplateP, posStr,
      totalStr, NULL, NULL);

   // Now set the title to use the new title string.
   frm = FrmGetFormPtr(MemoPadEditForm);
   FrmSetTitle (frm, EditViewTitlePtr);
   MemPtrUnlock(titleTemplateP);
}
```

Summary of Text API

Text Manager

Accessing Text

TxtPreviousCharSize()	TxtGetPreviousChar()
TxtGetNextChar()	TxtCharSize()
TxtGetChar()	TxtNextCharSize()

Changing Text

TxtReplaceStr()	TxtSetNextChar()
TxtConvertEncoding()	TxtTransliterate()
TxtParamString()	TxtTruncateString()

Segmenting Text

TxtGetTruncationOffset()	TxtCharBounds()
TxtGetWordWrapOffset()	TxtWordBounds()

Searching/Comparing Text

TxtCaselessCompare()	TxtCompare()
TxtFindString()	TxtPrepFindString()

Obtaining a Character's Attributes

TxtCharIsAlNum()	TxtCharIsAlpha()
TxtCharIsDelim()	TxtCharIsGraph()
TxtCharIsDigit()	TxtCharIsPrint()
TxtCharIsLower()	TxtCharIsUpper()
TxtCharIsSpace()	TxtCharXAttr()
TxtCharIsValid()	TxtCharIsHex()
TxtCharIsCntrl()	TxtCharIsHardKey()
TxtCharIsPunct()	TxtCharAttr()
TxtCharIsVirtual()	

Obtaining Character Encoding Information

TxtStrEncoding()	TxtEncodingName()
TxtMaxEncoding()	TxtCharEncoding()
TxtNameToEncoding()	TxtGetEncodingFlags()

Text Manager	
Working with Multi-byte Characters	
TxtByteAttr()	
Working with Single-byte Characters	
sizeOf7BitChar()	

String Manager	
Length of a String	
StrLen()	
Comparing Strings	
StrCompare()	StrNCompare()
StrCaselessCompare()	StrNCaselessCompare()
StrCompareAscii()	StrNCompareAscii()
Changing Strings	
StrCat()	StrNCat()
StrCopy()	StrNCopy()
StrToLower()	StrLCat()
StrLCopy()	
Searching Strings	
StrStr()	StrChr()
Converting	
StrAToI()	StrIToA()
StrIToH()	
Localized Numbers	
StrDelocalizeNumber()	StrLocalizeNumber()

2

Implementing Global Find

The Global Find facility allows a user to search all databases on the device for a particular string and then jump to the application that supports the database containing the matching record he or she wants to see.

Find is launched when the user taps the Find icon in the status bar. To integrate your application with the Global Find facility, you must do the following:

1. Create an `APP_LAUNCH_PREFS_RESOURCE` with ID 0 in your application's resource file. See the book *Resource File Formats* for more information on this resource.

TIP: Be sure you use a resource ID of 0.

2. In the resource, set the `ALPF_FLAG_NOTIFY_FIND` attribute to `true`.
3. In your application's `PilotMain()` function, implement responses to these launch codes:
   - `sysAppLaunchCmdFind`
   - sysAppLaunchCmdGoTo
   - `sysAppLaunchCmdSaveData`

Implementing sysAppLaunchCmdFind

The Find Manager sends each application registered to receive it the `sysAppLaunchCmdFind` launch code. This launch code is a signal that you should search your application's databases for the string passed to you in the launch code's parameter block. To implement the `sysAppLaunchCmdFind` launch code, you should do the following:

1. Open your application's record database using the open mode parameter that is passed to your application in the launch code's parameter block.
2. Use `FindDrawHeader()` to pass back to the Find Manager a string that is used to delineate your application's search results from those of all other applications in the Find Results dialog. This string is only used if your application has matching results.
3. Search each record using `TxtFindString()`. `TxtFindString()` accurately matches single-byte characters with their corresponding multi-byte characters, and it passes back the length of the matched text. You'll need to know the length of the matching text to highlight it when the system requests that you display the matching record.
4. If a match is found, do the following:
   a. Send information about the matching result back to the Find Manager using `FindSaveMatch()`. The parameters you pass to `FindSaveMatch()` are used in the `sysAppLaunchCmdGoTo` parameter block if the user selects your record in the Find Results dialog.
   b. If `FindSaveMatch()` returns `false`, it means that there is room to display the matching record. Call `WinDrawChars()` to send the text to the Find Results dialog. Use `FindGetLineBounds()` to determine where to draw, and use `WinDrawChars()` to do the drawing.

      When a Find is in progress, `WinDrawChars()` does not draw directly to the screen. The Find Manager traps all `WinDrawChars()` calls and copies the string into a buffer that it later draws to the screen.

IMPORTANT: Do not use `GcDrawTextAt()` in conjunction with Global Find. If you do, your string is not displayed.

5. If your database is potentially large, you should occasionally check the event queue to see if there is another event pending. For example, the user might start a find but then press one of the device's hard keys, which would generate a

`keyDownEvent`. Listing 2.1 shows an example of a search that checks the event queue every 16 records.

Listing 2.1 shows an example of a function that should be called in response to the `sysAppLaunchCmdFind` launch code.

Listing 2.1 Implementing Global Find

```
static void Search (FindParamsType *findParams)
{
   uint16_t pos;
   char * header;
   uint16_t recordNum;
   MemHandle recordH;
   MemHandle headerStringH;
   RectangleType r;
   Boolean done;
   Boolean match;
   DmOpenRef dbP;
   status_t err;
   DatabaseID dbID;
   MemoDBRecordPtr memoRecP;
   size_t longPos;
   size_t matchLength;

   // Locate the Memo database.
   dbID = DmFindDatabaseByTypeCreator(ty, cr, dmFindAllDB,
      NULL);
   if (!dbID) {
      findParams->more = false;
      return;
   }
   dbP = DmOpenDatabase(dbID,
      (DmOpenModeType)findParams->dbAccesMode);
   if (!dbP) {
      findParams->more = false;
      return;
   }

   // Display the heading line.
   headerStringH = DmGetResource(gAppResDBRef, strRsc,
      FindMemoHeaderStr);
   header = MemHandleLock(headerStringH);
   done = FindDrawHeader(findParams, header);
   MemHandleUnlock(headerStringH);
   DmReleaseResource(headerStringH);
   if (done)
```

```
      goto Exit;

   // Search the memos for the "find" string.
   recordNum = findParams->recordNum;
   while (true) {
      if ((recordNum & 0x000f) == 0 && // every 16th record
         EvtSysEventAvail(true)) {
         // Stop the search process.
         findParams->more = true;
         break;
      }

      recordH = DmQueryNextInCategory (dbP, &recordNum,
         dmAllCategories);

      // Have we run out of records?
      if (!recordH) {
         findParams->more = false;
         break;
      }

      memoRecP = MemHandleLock (recordH);

      // Search for the string passed,  if it's found display
      // the title of the memo.
      match = TxtFindString (&(memoRecP->note),
         findParams->strToFind, &longPos, &matchLength);

      pos = longPos;

      if (match) {
         // Add the match to the find paramter block,  if
         // there is no room to display the match the
         // following function will return true.
         done = FindSaveMatch (findParams, recordNum, pos, 0,
            matchLength, cardNo, dbID);

         if (!done) {
            // Get the bounds of the region where we will
            // draw the results.
            FindGetLineBounds (findParams, &r);

            // Display the title of the description.
            DrawMemoTitle (&(memoRecP->note), r.topLeft.x+1,
                r.topLeft.y, r.extent.x-2);

            findParams->lineNumber++;
         }
```

```
      }

      MemHandleUnlock(recordH);
      if (done) break;

      recordNum++;
   }

Exit:
   DmCloseDatabase (dbP);
}
```

Implementing sysAppLaunchCmdGoTo

When the user taps one of the results displayed in the Find Results dialog, the system sends a `sysAppLaunchCmdGoTo` launch code to the application containing the matching record. In most cases, your application should use the information in the launch code's parameter block to locate the matching record and display it, with the matching text highlighted.

Listing 2.2 shows how the Memo sample application responds to the `sysAppLaunchCmdGoto` launch code. It enqueues a `frmGotoEvent` for its Edit form, passing to this event information about which record to display. See the Memo sample application in the SDK for full source code.

Listing 2.2 Displaying the matching record

```
static void GoToRecord (GoToParamsPtr goToParams,
   Boolean launchingApp)
{
   uint16_t recordNum;
   EventType event;

   recordNum = goToParams->recordNum;
   ...

   // Send an event to goto a form and select the
   // matching text.
   MemSet (&event, sizeof(EventType), 0);

   event.eType = frmLoadEvent;
   event.data.frmLoad.formID = EditView;
```

```
   EvtAddEventToQueue (&event);

   MemSet (&event, sizeof(EventType), 0);
   event.eType = frmGotoEvent;
   event.data.frmGoto.recordNum = recordNum;
   event.data.frmGoto.matchPos = goToParams->matchPos;
   event.data.formGoto.matchLen = goToParams->matchLen;
   event.data.frmGoto.matchFieldNum =
      goToParams->matchFieldNum;
   event.data.frmGoto.formID = EditView;
   EvtAddEventToQueue (&event);
   ...
}
```

Implementing sysAppLaunchCmdSaveData

Your application receives the `sysAppLaunchCmdSaveData` launch code only if it is the current application when the user taps the Find icon. This launch code gives your application a chance to save any data that the user is in the process of entering before the Find is launched so that the search results are what the user expects.

For example, suppose that the user is editing a contact in the Address Book to change the first name from "Ted" to "Theodore." Before tapping the Done button, the user decides to do a search for any other records that contain "Ted." The user expects that the current record will not appear in the Find Results dialog.

For this reason, Palm OS® sends the Address Book the `sysAppLaunchCmdSaveData` launch code to give the application a chance to clean up any activity before the Find begins. Address Book responds to this launch code by calling `FrmSaveAllForms()`. `FrmSaveAllForms()` enqueues a `frmSaveEvent` for each open form. The event handler for the Edit form responds to the `frmSaveEvent` by saving the field that is currently being edited to the database. This way, its response to the `sysAppLaunchCmdFind` launch code will successfully pass over this record.

Summary of Find Manager API

Find Manager	
FindDrawHeader() FindGetLineBounds() FindSaveMatch()	TxtFindString() TxtPrepFindString()

3

Localized Applications

This chapter discusses these localization topics:

In addition to this chapter, also see Chapter 1, "Text," on page 3, which describes how to work with text and characters in a way that makes your application easily localizable.

This chapter does not cover how to actually perform localization of resources. For more information on this subject, see the documentation for your toolset.

Localization Guidelines

If there is a possibility that your application is going to be localized, you should follow these guidelines when you start planning the application. It's a good idea to follow these guidelines even if you don't think your application is going to be localized.

- If you use the English language version of the software as a guide when designing the layout of the screen, try to allow:
  - extra space for strings
  - larger dialogs than the English version requires
- Don't put language-dependent strings in code. If you have to display text directly on the screen, remember that a one-line warning or message in one language may need more than

one line in another language. See the section "Strings" on page 9 in Chapter 1, "Text," for further discussion.

- Don't depend on the physical characteristics of a string, such as the number of characters, the fact that it contains a particular substring, or any other attribute that might disappear in translation.
- Internal database names must use only 7-bit ASCII characters (0x20 through 0x7E). Store the user-visible name of your application in an `APP_ICON_NAME_RESOURCE` so that it can be translated to other languages.
- Use the functions described in this book when working with characters, strings, numbers, and dates.
- Consider using string templates as described in the section "Dynamically Creating String Content" on page 15 in Chapter 1. Use as many parameters as possible to give localizers greater flexibility. Avoid building sentences by concatenating substrings together, as this often causes translation problems.

  In general, avoid using `sprintf()` for localizable strings.
- Abbreviations may be the best way to accommodate the particularly scarce screen real estate on the Palm Powered™ device.
- Remember that user interface elements such as lists, fields, and tips scroll if you need more space.

The book *Exploring Palm OS: User Interface Guidelines* provides further user interface guidelines.

Locales

A **locale** specifies a place and is used to determine which formats, languages, and encodings to use for locale-specific items such as dates, numbers, and strings. In Palm OS®, a locale is identified by both a language and a country. In general, the language determines which character encoding is used on the device. The country specifies a dialect. For example, the language "English" uses different dialects for "USA" and for "Britain."

The system maintains several locale variables, allowing it to tailor system resources and functionality to fit the locale in which the device is being used. Most of the time, these variables all point to the same locale.

- The **ROM locale** is the locale stored in ROM on the device. The ROM locale is used to initialize the other locale variables after a hard reset. Use `LmGetROMLocale()` if you need to determine the ROM locale.
- The **system locale** is the locale that Palm OS code uses to obtain various locale settings. The system locale is initially set to the ROM locale, but it can be different. For example, on an EFIGS ROM (which is a ROM for English, French, Italian, German, and Spanish), the user chooses from among several languages when Palm OS starts up, and doing so changes the system locale but not the ROM locale. Use `LmGetSystemLocale()` to determine which is the system locale. This function also returns the character encoding used on the device.
- The **formats locale** is the locale that most applications should use to obtain locale-specific settings. The formats locale is initially the same as the system locale, but the user can change it in the Formats Preference panel. Use `LmGetFormatsLocale()` to obtain this locale. You can also use `LmSetFormatsLocale()` to change it, but do not do so without the user's permission.

  In most cases, the user is able to override the locale-specific settings using individual preferences on the Formats Preference panel. Therefore, you should check for a preference setting where one is available. See "Obtaining Locale Information" on page 33 for more information.
- The **overlay locale** is the locale that the Database Manager uses to decide which overlay to load. Thus, the overlay locale controls the target language of system and application user interface elements. It is initially set to the system locale. Use `DmGetOverlayLocale()` to obtain the value of this locale, and use `DmSetOverlayLocale()` if you need to change it.
- The **fallback overlay locale** is the locale that the Database Manager uses if an overlay for the overlay locale does not exist. By default, it uses the ROM locale. Use `DmGetFallbackOverlayLocale()` to obtain the value of

this locale and use `DmSetFallbackOverlayLocale()` if you need to change it.

Overlays

You localize Palm OS resource databases using **overlays**. Localization overlays provide a way of localizing a software module without requiring a recompile or modification of the software. Each overlay database is a separate resource database that provides an appropriately localized set of resources for a single software module (the PRC file, or **base database**) and a single target **locale** (language and country).

No requirements are placed on the base database, so for example, third parties can construct localization overlays for existing applications without forcing any modifications by the original application developer. In rare cases, you might want to disable the use of overlays to prevent third parties from creating overlays for your application. To do so, you should include an `APP_LAUNCH_PREFS_RESOURCE` with ID 0 in the database and set its `ALPF_FLAG_NO_OVERLAY` flag to `true`.

An overlay database has the same creator as the base database, but its type is `'ovly'`, and a suffix identifying the target locale is appended to its name. For example, `Datebook.prc` might be overlaid with a database named `Datebook_jaJP`, which indicates that this overlay is for Japan. Each overlay database has an `OVERLAY_RESOURCE` with ID 1000.

When a resource file is opened, the Database Manager looks for an overlay matching the base database and the overlay locale. The overlay database's name must match the base database's name, its suffix must match the locale's suffix, and it must have an `OVERLAY_RESOURCE` with ID 1000. If the name, suffix, and overlay resource are all correct and the overlay passes various checks to ensure it's appropriate for use with the base database, the overlay is opened in addition to the base database. When the base database is closed, its overlay is closed as well.

The overlay is opened in read-only mode and is hidden from the programmer. When you open a database, you'll receive a reference to the base database, not the overlay. You can simply make Database

Manager calls like you normally would, and the Database Manager accesses the overlay where appropriate.

When accessing a localizable resource, do not use functions that search for a resource only in the database you specify. For example, see Listing 3.1.

Listing 3.1 Wrong way to access resources

```
// WRONG! searches only one database.
DmOpenRef dbP = DmNextOpenResDatabase(NULL);
uint16_t resIndex = DmFindResource(dpP, strRsc, strRscID,
   NULL);
MemHandle resH = DmGetResourceByIndex(dbP, resIndex);
```

In Listing 3.1, `dbP` is a reference to the most recently opened database, which is typically the overlay version of the database. Passing this reference to `DmFindResource()` means that you are searching only the overlay database for the resource. If you're searching for a non-localized resource, `DmFindResource()` won't be able to locate it. Instead, you should use `DmGetResource()`, which searches a database and its overlay for a resource

Listing 3.2 Correct way to access resources

```
// Right. DmGetResource searches both databases.
MemHandle resH = DmGetResource(dbRef, strRsc, strRscID);
```

The Database Manager opens an overlay only if the base database is opened in read-only mode. If you open a resource database in read-write mode, the associated overlay is not opened. What's more, if you modify an overlaid resource in the base database, the overlay database may no longer be valid. Thus if you change the base database, you must also change the overlay database.

For more information on overlays and resource databases, see *Exploring Palm OS: Memory, Databases, and Files*.

Dates and Times

If your application deals with dates and times, it should abide by the values the user has set in the system preference for date and time display. The default preferences at startup vary among locales, and the default values can be overridden by the user.

To check the system preferences call `PrefGetPreference()` with one of the values listed in the second column of Table 3.1. The third column lists an enumerated type that helps you interpret the value.

Table 3.1 Date and time preferences

Preference	Name	Returns a value of type
Date formats (i.e., month first or day first)	`prefDateFormat`, `prefLongDateFormat`	`DateFormatType`
Time formats (i.e., use a 12-hour clock or use a 24-hour clock)	`prefTimeFormat`	`TimeFormatType`
Start day of week (i.e., Sunday or Monday)	`prefWeekStartDay`	0 (Sunday) or 1 (Monday)
Local time zone	Use the `gettimezone()` function.	Minutes east of Greenwich Mean Time (GMT), also known as Universal Coordinated Time (UTC).

To work with dates and times in your code, use the Date and Time Manager API. It contains functions such as `DateToAscii()`, `TimeToAscii()`, `DayOfMonth()`, `DayOfWeek()`, `DaysInMonth()`, `TimeToInt()`, `TimeIs24HourFormat()`, and `DateTemplateToAscii()`, which allow you to work with dates and times independent of the user's preference settings.

Numbers

If your application displays large numbers or floating-point numbers, you must check and make sure you are using the appropriate thousands separator and decimal separator for the device's country by doing the following (see Listing 3.3):

1. Store number strings using US conventions, which means using a comma (,) as the thousands separator and a decimal point (.) as the decimal separator.
2. Use `PrefGetPreference()` and `LmGetNumberSeparators()` to retrieve information about how the number should be displayed.
3. Use `StrLocalizeNumber()` to perform the localization.
4. If a user enters a number that you need to manipulate in some way, convert it to the US conventions using `StrDelocalizeNumber()`.

Listing 3.3 Working with numbers

```
// store numbers using US conventions.
char *jackpot = "20,000,000.00";
char thou; // thousand separator
char dp; // decimal separator

// Retrieve user's preferred number format.
LmGetNumberSeparators((NumberFormatType)
   PrefGetPreference(prefNumberFormat), &thou, &dp);
// Localize jackpot number. Converts "," to thou
// and "." to dp.
StrLocalizeNumber(jackpot, thou, dp);
// Display string.
// Assume inputNumber is a number user entered,
// convert it to US conventions this way. Converts
// thou to "," and dp to "."
StrDelocalizeNumber(inputNumber, thou, dp);
```

Obtaining Locale Information

Some applications may require information about a specific locale. For example, an application might need to know the country name for a locale.

The information that most applications require is stored in the system preferences structure and can be obtained using `PrefGetPreference()`. This is the recommended way of obtaining locale-specific settings because the user can override many of these settings. Applications should always honor the user's preferences rather than the locale defaults.

Other locale-specific settings can not be set by the user and are not stored in the system preferences. Instead, these settings are stored in a private resource that contains information about several possible locales, including the locale currently used by the system. For example, the user cannot change the symbol used for the local currency. If your application needs this information, it must use the Locale Manager function `LmGetLocaleSetting()` to retrieve it. Listing 3.4 shows how to use `LmGetLocaleSetting()`.

Listing 3.4 Retrieving a locale setting using Locale Manager

```
LmLocaleType locale;
char currencySymbol[kMaxCurrencySymbolLen+1];
uint16_t index;

// Find out what the formats locale is.
LmGetFormatsLocale(&locale);

// Find out which index in the locale resource
// contains info about that locale.
index = LmBestLocaleToIndex(&locale);

// Get the currency symbol stored in the locale at
// that index.
LmGetLocaleSetting(index, lmChoiceCurrencySymbol,
   currencySymbol, sizeof(currencySymbol));
```

Table 3.2 shows which types of information about the formats locale should be retrieved from the system preferences and which types should be retrieved from the locale resource. Of course, if you want to retrieve information about a different locale or if you want to look up the default used for the formats locale, you would always use the Locale Manager instead of the Preferences Manager.

Table 3.2 Obtaining locale information

Value	Function used to retrieve value
Language code	`LmGetLocaleSetting(..., lmChoiceLocale, ...)`
Locale description	`LmGetFormatsLocale()`
Country code	`LmGetLocaleSetting(..., lmChoiceLocale, ...)`
Country name	`LmGetLocaleSetting(..., lmChoiceCountryName, ...)`
Currency name	`LmGetLocaleSetting(..., lmChoiceCurrencyName, ...)`
Currency symbol	`LmGetLocaleSetting(..., lmChoiceCurrencySymbol, ...)`
Unique currency symbol	`LmGetLocaleSetting(..., lmChoiceUniqueCurrencySymbol, ...)`
Measurement system (metric or English)	`PrefGetPreference(prefMeasurementSystem)`
Number formats	`PrefGetPreference(prefNumberFormat)`
Number of decimal places for monetary values	`LmGetLocaleSetting(..., lmChoiceCurrencyDecimalPlaces, ...)`
Starting day of the week	`PrefGetPreference(prefWeekStartDay)`
Date formats	`PrefGetPreference(prefDateFormat)` `PrefGetPreference(prefLongDateFormat)`
Time format	`PrefGetPreference(prefTimeFormat)`
Time zone	Use the function `gettimezone()`.

Summary of Localization API

Numbers	
LmGetNumberSeparators() StrDelocalizeNumber()	StrLocalizeNumber()

Locale Manager	
LmBestLocaleToIndex() LmCountryToISOName() LmGetFormatsLocale() LmGetLocaleSetting() LmGetNumLocales() LmGetROMLocale()	LmGetSystemLocale() LmISONameToCountry() LmISONameToLanguage() LmLanguageToISOName() LmLocaleToIndex() LmSetFormatsLocale()

Date and Time Manager	
DateAdjust() DateDaysToDate() DateTemplateToAscii() DateToAscii() DateToDays() DateToDOWDMFormat() DateToInt() DayOfMonth() DayOfWeek() DaysInMonth()	TimAdjust() TimDateTimeToSeconds() TimeGetFormatSeparator() TimeGetFormatSuffix() TimeIs24HourFormat() TimeToAscii() TimeToInt() TimeZoneToAscii() TimSecondsToDateTime() TimTimeZoneToUTC() TimUTCToTimeZone()

Database Manager
DmGetOverlayLocale() DmSetOverlayLocale() DmGetFallbackOverlayLocale() DmSetFallbackOverlayLocale()

Part II
Reference

This part contains reference material for the text and localization managers. It covers:

4

Find

This chapter describes the Global Find facility API declared in the header file `Find.h`. It covers the following topics:

For more information on the Find Manager, see Chapter 2, “Implementing Global Find,” on page 19.

Find Structures and Types

FindMatchType Struct

Purpose A structure that the Find Manager uses to save information about each matching record.

Declared In `Find.h`

Prototype

```
typedef struct {
   DatabaseID appDbID;
   DatabaseID dbID;
   Boolean foundInCaller;
   uint8_t reserved;
   uint16_t reserved2;
   uint32_t recordNum;
   uint32_t recordID;
   uint32_t matchFieldNum;
   size_t matchPos;
   size_t matchLen;
   uint32_t matchCustom;
} FindMatchType;
typedef FindMatchType *FindMatchPtr
```

Fields

`appDbID`
: Database ID of the application with the matching record.

`dbID`
: Database ID of the record database in which the match was found.

`foundInCaller`
: If `true`, the matching record was found in the application that was active when the user launched Find.

`reserved`
: Reserved for future use.

`reserved2`
: Reserved for future use.

`recordNum`
: The index of the record containing the matching text.

`recordID`
: The unique ID of the record containing the matching text.

`matchFieldNum`
: The index of the text field that should display the matching text.

`matchPos`
: Byte offset of the start of the matching text within the record.

`matchLen`
: The number of bytes of matching text.

`matchCustom`
: Application-specific information.

See Also `GoToParamsType`, `FindSaveMatch()`

FindParamsType Struct

Purpose Parameter block for `sysAppLaunchCmdFind`.

Declared In `Find.h`

Prototype
```
typedef struct {
   uint32_t dbAccesMode;
   uint32_t recordNum;
   uint32_t recordID;
   Boolean more;
   char strAsTyped[maxFindStrLen+1];
   char strToFind[maxFindStrPrepLen+1];
   Boolean continuation;
   uint16_t lineNumber;
   RectangleType bounds;
   uint16_t numMatches;
   Boolean searchedCaller;
   uint8_t reserved1;
   uint8_t reserved2;
   uint8_t reserved3;
   DatabaseID callerAppDbID;
   DatabaseID appDbID;
   DmSearchStateType searchState;
} FindParamsType
typedef FindParamsType *FindParamsPtr
```

Fields

`dbAccesMode`
: Mode in which to open the application's database. Pass this directly to `DmOpenDatabase()` as the *mode* parameter. Its value is either `dmModeReadOnly` or `dmModeReadOnly | dmModeShowSecret`. (See `DmOpenModeType` for more information.)

`recordNum`
: Index of last record that contained a match. Start the search from this location. Do not set this value directly. Instead, call `FindSaveMatch()` when you have a matching record.

`recordID`
: Unique ID of the last record that contained a match. Do not set this value directly. Instead, call `FindSaveMatch()` when you have a matching record.

`more`
: No longer used.

`strAsTyped`
: Search string as the user entered it.

`strToFind`
: Normalized version of the search string. The method by which a search string is normalized varies depending on the version of Palm OS® and the character encoding supported by the device. You pass `strToFind` directly to the `TxtFindString()` search function.

`continuation`
: No longer used.

`lineNumber`
: Line number of the next line that displays the results. Do not set this field directly. It is incremented by a call to `FindDrawHeader()`.

`bounds`
: The current size of the form that the Find Manager is displaying. This field is used internally by the Find Manager.

`numMatches`
: The current number of matches. Do not set this field directly. Instead, call `FindSaveMatch()`, which increments it for you.

`searchedCaller`
: If `true`, the application that was active at the time the user tapped the Find icon has responded to this launch code. This application is always searched before any others.

NOTE: In Palm OS Cobalt version 6.0, the current application is always searched regardless of its `APP_LAUNCH_PREFS_RESOURCE` settings.

`reserved1`
: Reserved for future use.

`reserved2`
: Reserved for future use.

`reserved3`
: Reserved for future use.

`callerAppDbID`
Database ID of the application that was active when the user tapped the Find button. Do not change the value of this field; the system sets it and uses it when searching for application databases.

`appDbID`
The ID of your application's resource database. Do not set this field directly; the system sets it and uses it when searching for application databases.

`searchState`
System use only.

Find Constants

Size Constants

Purpose Specify the maximum sizes of search strings.

Declared In `Find.h`

Constants `#define maxFindStrLen 48`
The maximum length in bytes of the string the user can enter in the Find dialog. This is the maximum length of the `strAsTyped` field in `FindParamsType`.

`#define maxFindStrPrepLen 64`
The maximum length of a normalized string to be searched for using the Find facility. A normalized string has already had TxtPrepFindString() called on it.

Find Launch Codes

sysAppLaunchCmdFind

Purpose Sent when the user has entered text in the Find dialog. The application should search for the string that the user entered and return any records matching the find request.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdFind 1`

Parameters The launch code's parameter block pointer references a `FindParamsType` structure.

Comments The system only send this launch code to applications that have an `APP_LAUNCH_PREFS_RESOURCE` of ID 0 with the flag `ALPF_FLAG_NOTIFY_FIND` set to `true`.

NOTE: In Palm OS Cobalt version 6.0, the current application is always searched regardless of its `APP_LAUNCH_PREFS_RESOURCE` settings.

Most applications that store text in a database should support this launch code. When they receive it, they should search all records for matches to the find string and return all matches. Chapter 2 describes how to respond to this launch code.

The system displays the results of the query in the Find Results dialog. The system continues the search with each application until all of the applications on the device have had a chance to respond.

An application can also integrate the find operation in its own user interface and send the launch code to a particular application.

Applications that support this launch code should also support `sysAppLaunchCmdSaveData` and `sysAppLaunchCmdGoTo`.

Find Functions and Macros

Find Function

Purpose System use only function that handles the Global Find feature. Applications should not call this function.

Declared In `Find.h`

Prototype `void Find (GoToParamsPtr goToP)`

Parameters ↔ *goToP*
A pointer to a `GoToParamsType` structure. Usually `NULL`.

Returns Nothing.

Comments The system calls this function when it receives a `keyDownEvent` with the `vchrFind` virtual character. If you want to implement your own Find, the correct thing to do is to intercept this event before the call to `SysHandleEvent()` in your event loop.

FindDrawHeader Function

Purpose Draws the header line that separates, by application, the list of found items.

Declared In `Find.h`

Prototype
```
Boolean FindDrawHeader (FindParamsPtr findParams,
    char const *title)
```

Parameters → *findParams*
Pointer to the `sysAppLaunchCmdFind` launch code's parameter block. See `FindParamsType`.

→ *title*
String to display as the title for the current application.

Returns Always returns `false`.

Comments Call this function once at the beginning of your application's response to the `sysAppLaunchCmdFind` launch code. This function draws a header for your application's find results. The header separates the search results from your application with the search results from another application.

The header is only drawn if your application successfully locates a matching result.

If your application searches multiple databases, you may also use `FindDrawHeader()` as a separator between databases.

FindGetLineBounds Function

Purpose Returns the bounds of the next available line for displaying a match in the Find Results dialog.

Declared In `Find.h`

Prototype

```
void FindGetLineBounds
    (const FindParamsType *findParams,
    RectanglePtr r)
```

Parameters

→ *findParams*
Pointer to the `sysAppLaunchCmdFind` launch code's parameter block. See `FindParamsType`.

← *r*
The bounds of the area that should contain the next line of results.

Returns Nothing.

Comments Call this function when you've found a match that should be displayed in the Find Results dialog, and then call `WinDrawChars()`, passing it the location returned in *r*. `WinDrawChars()` does not draw directly to the Find Results dialog. The Find Manager traps all `WinDrawChars()` calls while a Find is in progress and copies the string to a buffer. The Find Manager displays the string at the appropriate location when it has a screen full of data to display.

FindSaveMatch Function

Purpose Saves the record and position within the record of a Find match. This information is saved so that it's possible to later navigate to the match.

Declared In `Find.h`

Prototype

```
Boolean FindSaveMatch (FindParamsPtr findParams,
    uint32_t recordNum, uint32_t recordID,
    size_t matchPos, size_t matchLen,
    uint32_t fieldNum, uint32_t appCustom,
    DatabaseID dbID)
```

Parameters

→ *findParams*
: Pointer to the `sysAppLaunchCmdFind` launch code's parameter block (`FindParamsType`).

→ *recordNum*
: Record index. This parameter sets the `recordNum` field in the `sysAppLaunchCmdGoTo` parameter block (`GoToParamsType`).

→ *recordID*
: Unique ID of the record containing a match. This parameter sets the `recordID` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *matchPos*
: Byte offset of the start of the matching string in the record. This parameter sets the `matchPos` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *matchLen*
: The number of bytes of matched text found in the record. This parameter sets the `matchLen` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *fieldNum*
: Index of the text field in which the matching string should be displayed. This parameter sets the `matchFieldNum` field in the `sysAppLaunchCmdGoTo` parameter block.

If your application's database is a schema database, use this field to set the column ID in which the matching text was found.

→ *appCustom*
Extra data the application can save with a match. This parameter sets the `matchCustom` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *dbID*
Database ID of the database that contains the match. This parameter sets the `dbID` field in the `sysAppLaunchCmdGoTo` parameter block.

Returns `true` if the application should exit from the search.

Comments Call this function when your application finds a record with a matching string (`TxtFindString()` returns `true`). This function saves the information you pass. If the user taps this selection in the Find Results dialog, the information is retrieved and used to set up the `sysAppLaunchCmdGoTo` launch code's parameter block.

FindSaveMatchV40 Function

Purpose Saves the record and position within the record of a text search match. This information is saved so that it's possible to later navigate to the match.

Declared In `Find.h`

Prototype
```
Boolean FindSaveMatchV40(FindParamsPtr findParams,
    uint16_t recordNum, uint16_t pos,
    uint16_t fieldNum, uint32_t appCustom,
    uint16_t cardNo, LocalID dbID)
```

Parameters

→ *findParams*
Pointer to the `sysAppLaunchCmdFind` launch code's parameter block. See `FindParamsType`.

→ *recordNum*
Record index. This parameter sets the `recordNum` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *pos*
Offset of the match string from start of record. This parameter sets the `matchPos` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *fieldNum*
Field number that the string was found in. This parameter sets the `matchFieldNum` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *appCustom*
Extra data the application can save with a match. This parameter sets the `matchCustom` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *cardNo*
Card number of the database that contains the match. This parameter sets the `dbCardNo` field in the `sysAppLaunchCmdGoTo` parameter block.

→ *dbID*
Local ID of the database that contains the match. This parameter sets the `dbID` field in the `sysAppLaunchCmdGoTo` parameter block.

Returns `true` if the application should exit from the search.

Compatibility This function differs from `FindSaveMatch()` in that it does not have *recordID* or *matchLen* parameters and its *matchNum*, *fieldNum*, and *matchPos* parameters are all smaller. In earlier Palm OS releases, applications used the *appCustom* parameter to store the length of the matching string. These applications should now call `FindSaveMatch()` and pass the length of the matching string in *matchLen*.

FindStrInStrV50 Function

Purpose Performs a case-blind prefix search for a string in another string. This function assumes that the string to find has already been normalized for searching.

Declared In `Find.h`

Prototype
```
Boolean FindStrInStrV50 (char const *strToSearch,
    char const *strToFind, uint16_t *posP)
```

Parameters

→ *strToSearch*
String to search.

→ *strToFind*
Normalized version of the text string to be found.

← *`posP`*

If a match is found, contains the offset of the match within *`strToSearch`*.

Returns `true` if the string was found.

Compatibility Do not use this function. Instead use `TxtFindString()`, which can successfully search strings that containing multi-byte characters and can return the length of the matching text.

5

Locale Manager Types

This chapter describes the types and constants declared in `LocaleMgrTypes.h`. It covers:

For more information on the Locale Manager, see the chapter "Localized Applications" on page 27.

Locale Manager Structures and Types

CountryType Typedef

Purpose Legacy type that defines an old style of country code constants. Use `LmCountryType` instead.

Declared In `LocaleMgrTypes.h`

Prototype `typedef uint8_t CountryType`

LanguageType Typedef

Purpose Legacy type that defines an old style of language code constants. Use `LmLanguageType` instead.

Declared In `LocaleMgrTypes.h`

Prototype `typedef uint8_t LanguageType`

LmCountryType Typedef

Purpose Identifies a country in the ISO 3166 standard.

Declared In `LocaleMgrTypes.h`

Prototype `typedef uint16_t LmCountryType`

Comments The country type constants have the following format:

`c`*CountryName*

where *CountryName* is the name of the country. There is one constant for each country identified in the ISO 3166 standard, which currently defines 239 countries.

The following table shows examples of the country type constants. For a complete list, see the `LocaleMgrTypes.h` file.

Constant	Value	Description
`cAustralia`	`'AU'`	Australia
`cAustria`	`'AT'`	Austria
`cBelgium`	`'BE'`	Belgium

LmLanguageType Typedef

Purpose Identifies a language in the ISO 639 standard.

Declared In `LocaleMgrTypes.h`

Prototype `typedef uint16_t LmLanguageType`

Comments The language type constants have the following format:

`l`*LanguageName*

where *LanguageName* is the name of the language. There is one constant for each language specified in the ISO 639 standard, which currently defines 137 languages.

The following table shows examples of the language type constants. For a complete list, see the `LocaleMgrTypes.h` file.

Constant	Value	Description
`lEnglish`	`'en'`	English
`lFrench`	`'fr'`	French
`lGerman`	`'de'`	German

LmLocaleType Struct

Purpose Defines the country and language used in a locale.

Declared In `LocaleMgrTypes.h`

Prototype

```
struct _LmLocaleType {
   LmLanguageType language;
   LmCountryType country;
}
typedef struct _LmLocaleType LmLocaleType
```

Fields

`language`
: An `LmLanguageType` constant that identifies the language spoken in the current locale.

`country`
: An `LmCountryType` constant that identifies the locale's country, which helps to identify the language dialect. For example, a language of `lEnglish` specifies a different dialect if the country is `cUnitedKingdom` than if it is `cUnitedStates`.

NumberFormatType Typedef

Purpose Specifies how numbers are formatted.

Declared In `LocaleMgrTypes.h`

Prototype `typedef Enum8 NumberFormatType`

Comments The `NumberFormatType` constants values are not public because you should never have to check them directly. Retrieve the `NumberFormatType` from the preference or the locale and pass it directly to `LmGetNumberSeparators()` to retrieve the appropriate separator characters for thousands and decimals.

See Also "Numbers"

Locale Manager Constants

LmLocaleSettingChoice Typedef

Purpose Defines constants that you can pass to the `LmGetLocaleSetting()` function to specify which locale setting to retrieve.

Declared In `LocaleMgrTypes.h`

Prototype `typedef uint16_t LmLocaleSettingChoice`

Constants

`#define lmChoiceCountryName ((LmLocaleSettingChoice)5)`
A string buffer of size `kMaxCountryNameLen+1` containing the name of the locale's country.

`#define lmChoiceCurrencyDecimalPlaces ((LmLocaleSettingChoice)15)`
A `uint16_t` containing the number of decimal places that monetary values are typically given.

`#define lmChoiceCurrencyName ((LmLocaleSettingChoice)12)`
A string buffer of size `kMaxCurrencyNameLen+1` bytes containing the name of the currency used in this locale.

`#define lmChoiceCurrencySymbol ((LmLocaleSettingChoice)13)`
A string buffer of size `kMaxCurrencySymbolLen+1` bytes containing the symbol used to denote monetary values in this locale.

`#define lmChoiceDateFormat ((LmLocaleSettingChoice)6)`
A `DateFormatType` containing the short date format used in this locale. For example:

95/12/31

`#define lmChoiceInboundDefaultVObjectEncoding ((LmLocaleSettingChoice)23)`

A `CharEncodingType` containing the inbound encoding for vObjects with no CHARSET property.

`#define lmChoiceLocale ((LmLocaleSettingChoice)1)`

An `LmLocaleType` structure containing the locale's language and country codes.

`#define lmChoiceLongDateFormat ((LmLocaleSettingChoice)7)`

A `DateFormatType` containing the long date format used in this locale. For example:

31 Dec 1995

`#define lmChoiceMeasurementSystem ((LmLocaleSettingChoice)16)`

A `MeasurementSystemType` containing the measurement system (metric system or English system) used in this locale.

`#define lmChoiceNumberFormat ((LmLocaleSettingChoice)11)`

A `NumberFormatType` containing the format used for numbers, with regards to the thousands separator and the decimal point, in this locale.

`#define lmChoiceOutboundVObjectEncoding ((LmLocaleSettingChoice)22)`

A `CharEncodingType` containing the outbound encoding for vObjects.

`#define lmChoicePrimaryEmailEncoding ((LmLocaleSettingChoice)20)`

A `CharEncodingType` containing the first attempt at email encoding.

`#define lmChoicePrimarySMSEncoding ((LmLocaleSettingChoice)18)`

A `CharEncodingType` containing the first attempt at SMS encoding.

`#define lmChoiceSecondaryEmailEncoding ((LmLocaleSettingChoice)21)`

A `CharEncodingType` containing the second attempt at email encoding.

`#define lmChoiceSecondarySMSEncoding ((LmLocaleSettingChoice)19)`

A `CharEncodingType` containing the second attempt at SMS encoding.

`#define lmChoiceSupportsLunarCalendar ((LmLocaleSettingChoice)17)`

A Boolean that specifies if the locale uses the Chinese Lunar Calendar. If `true`, the locale uses the calendar.

`#define lmChoiceTimeFormat ((LmLocaleSettingChoice)8)`

A `TimeFormatType` containing the format used for time values in this locale.

`#define lmChoiceTimeZone ((LmLocaleSettingChoice)10)`

An `int16_t` containing the locale's default time zone given as the number of minutes east of Greenwich Mean Time (GMT).

`#define lmChoiceUniqueCurrencySymbol ((LmLocaleSettingChoice)14)`

A string buffer of size `kMaxCurrencySymbolLen+1` bytes containing the unique symbol for monetary values.

For example, the symbol `$` is used both for US dollars and Portuguese escudos. The unique currency symbol for US dollars is `US$`.

`#define lmChoiceWeekStartDay ((LmLocaleSettingChoice)9)`

A `uint16_t` containing the first day of the week (Sunday or Monday) in this locale. Days of the week are numbered from 0 to 6 starting with Sunday = 0.

Locale Manager Errors

Purpose Error constants used by the Locale Manager.

Declared In `LocaleMgrTypes.h`

`#define lmErrBadLocaleIndex (lmErrorClass | 2)`

A locale index is out of range.

`#define lmErrBadLocaleSettingChoice (lmErrorClass | 3)`
An unrecognized value was used for a `LmLocaleSettingChoice` constant.

`#define lmErrSettingDataOverflow (lmErrorClass | 4)`
The buffer passed to `LmGetLocaleSetting()` is too small for the specified value.

`#define lmErrUnknownLocale (lmErrorClass | 1)`
An unrecognized value was passed for a `LmLocaleType` structure. Note that Palm OS® does not provide locales for all valid country and language combinations. For example, there currently is no locale defined for `cIsrael` and `lHebrew`.

Locale Manager Size Constants

Purpose Specify the size of strings to allocate for some of the locale settings.

Declared In `LocaleMgrTypes.h`

Constants `#define kMaxCountryNameLen 31`
The maximum length of a country name string.

`#define kMaxCurrencyNameLen 31`
The maximum length of a currency name string.

`#define kMaxCurrencySymbolLen 10`
The maximum length of a currency symbol string.

Comments These constants do not count the terminating null character. Therefore, you need to allocate a string of size `kMaxCountryNameLen`+1 to hold a country name, for example.

Locale Wildcard Constants

Purpose Constants that can be used as wildcard values when searching for a locale using `LmLocaleToIndex()` or `LmBestLocaleToIndex()`.

Declared In `LocaleMgrTypes.h`

Constants `#define lmAnyCountry ((LmCountryType)'\?\?')`
Specifies any country.

```
#define lmAnyLanguage ((LmLanguageType)'\?\?')
```

Specifies any language.

See Also LmLanguageType, LmBestLocaleToIndex(), LmLocaleToIndex()

6

Locale Manager

This chapter describes the Locale Manager API as described in the header file `LocaleMgr.h`. It discusses the following topics:

For more information on the Locale Manager, see the chapter "Localized Applications" on page 27.

Locale Manager Functions and Macros

LmBestLocaleToIndex Function

Purpose Converts an `LmLocaleType` to an index.

Declared In `LocaleMgr.h`

Prototype

```
uint16_t LmBestLocaleToIndex
    (const LmLocaleType *iLocale)
```

Parameters → *`iLocale`*
The locale to convert. This locale can use the constants `lmAnyCountry` or `lmAnyLanguage` as wildcards.

Returns The index of the known locale that most closely matches *`iLocale`*.

Comments This function first tries to find a locale that matches both the language and country of *`iLocale`*. If it does not exist, it then tries to match only the country and only the language. If it cannot find a match for either the country or the language, it returns the index of the first locale (in other words, it returns an index of 0).

Example The following example shows using `LmGetFormatsLocale()` to return the locale used to set the display preferences for such things as dates and numbers, and then passing that to `LmBestLocaleToIndex()` to obtain a valid index to pass to `LmGetLocaleSetting()`.

```
LmLocaleType locale;
char oValue[kMaxCurrencySymbolLen+1];
uint16_t index;

LmGetFormatsLocale(&locale);
index = LmBestLocaleToIndex(&locale);
LmGetLocaleSetting(index, lmChoiceCurrencySymbol, oValue,
  sizeof(oValue));
```

See Also `LmLocaleToIndex()`

LmCountryToISOName Function

Purpose Converts an `LmCountryType` to a string.

Declared In `LocaleMgr.h`

Prototype

```
status_t LmCountryToISOName
    (LmCountryType iCountry, char *oISONameStr)
```

Parameters

→ *`iCountry`*
An `LmCountryType` variable specifying the country code as it is stored in a locale.

← *`oISONameStr`*
A string that is at least three bytes long. Upon return, this string contains the country code converted to a string.

Returns `errNone` upon success or `lmErrUnknownLocale` if *`iCountry`* is not a valid country code.

Comments The Database Manager uses this function to convert the overlay locale's country code into a string so that it can construct an overlay database name.

See Also `LmISONameToCountry()`, `LmISONameToLanguage()`, `LmLanguageToISOName()`

LmGetFormatsLocale Function

Purpose Returns the formats locale.

Declared In `LocaleMgr.h`

Prototype
```
void LmGetFormatsLocale
    (LmLocaleType *oFormatsLocale)
```

Parameters
← *oFormatsLocale*
An `LmLocaleType` that identifies the formats locale.

Returns Nothing.

Comments The **formats locale** is initially set to the system locale; however, the users can change the formats locale in the Formats Preference panel if they prefer a different locale.

See Also `LmSetFormatsLocale()`, `LmGetSystemLocale()`

LmGetLocaleSetting Function

Purpose Returns the requested setting for a given locale.

Declared In `LocaleMgr.h`

Prototype
```
status_t LmGetLocaleSetting(uint16_t iLocaleIndex,
    LmLocaleSettingChoice iChoice, void *oValue,
    uint16_t iValueSize)
```

Parameters
→ *iLocaleIndex*
Index of the locale whose settings you want to retrieve. Use `LmLocaleToIndex()` or `LmBestLocaleToIndex()` to obtain this value.

→ *iChoice*
The `LmLocaleSettingChoice` constant for the setting you want to retrieve.

← *oValue*
The value of the *iChoice* setting.

→ *iValueSize*
The size of the *oValue* buffer. The size of this buffer depends on the value of *iChoice*.

Returns One of the following values:

`errNone`
Success.

`lmErrBadLocaleIndex`
`iLocaleIndex` is out of range.

`lmErrSettingDataOverflow`
The *`oValue`* buffer is too small to hold the setting's value.

`lmErrBadLocaleSettingChoice`
The *`iChoice`* parameter contains an unknown or unsupported value.

Comments This function accesses the private locale system resource and returns the requested information in the *`oValue`* parameter. The size and type of the *`oValue`* parameter depends on which setting you want to retrieve. The `LmLocaleSettingChoice` documentation describes the type of data returned in *`oValue`* for each setting. For fixed-size values, make sure that the size of the *`oValue`* buffer is exactly the size of the returned value. It should be neither larger than nor smaller than the size of the returned value.

This function returns the default settings for the locale. Users can override many of the locale settings using the Preferences application. Applications should always honor the user's preferences rather than the locale defaults. For this reason, it's recommended that if a corresponding system preference is available, you should check it instead (using `PrefGetPreference()`). Use `LmGetLocaleSetting()` only if you want to retrieve values that the user cannot override (such as the country name or currency symbol) or if you want to retrieve information about a locale other than the current locale.

See Also `LmGetNumLocales()`, `LmLocaleToIndex()`

LmGetNumberSeparators Function

Purpose Gets localized number separators.

Declared In `LocaleMgr.h`

Prototype
```
void LmGetNumberSeparators
    (NumberFormatType iNumberFormat,
    char *oThousandSeparatorChar,
    char *oDecimalSeparatorChar)
```

Parameters

→ *iNumberFormat*
The format to use (see `NumberFormatType`).

← *oThousandSeparatorChar*
A pointer to the character used for the thousands separator. This is not a string. It does not have the terminating null character.

← *oDecimalSeparatorChar*
A pointer to the character used for the decimal separator. This is not a string. It does not have the terminating null character.

Returns Nothing.

Comments The format to use is stored in the system preferences. You can obtain it by passing `prefNumberFormat` to `PrefGetPreference()`.

See Also `StrLocalizeNumber()`, `StrDelocalizeNumber()`, "Numbers"

LmGetNumLocales Function

Purpose Returns the number of known locales.

Declared In `LocaleMgr.h`

Prototype `uint16_t LmGetNumLocales (void)`

Parameters None.

Returns The number of locales that the locale system resource defines.

Comments Use this function to obtain the range of possible values that you can pass as an index to `LmGetLocaleSetting()`. If `LmGetNumLocales()` returns 3, then `LmGetLocaleSetting()` accepts indexes in the range of 0 to 2.

This function returns only the number of locales for which the ROM has locale information. It does not return the number of locales that could possibly be defined. For example, the system resource currently contains no locale whose language is `lHebrew` and country is `cIsrael`, even though that is a valid locale.

LmGetROMLocale Function

Purpose Returns the ROM locale.

Declared In `LocaleMgr.h`

Prototype

```
CharEncodingType LmGetROMLocale
    (LmLocaleType *oROMLocale)
```

Parameters ← *oROMLocale*

Points to an `LmLocaleType` that identifies the ROM locale. Pass `NULL` if you don't want to retrieve this value.

Returns A `CharEncodingType` constant that specifies the character encoding used in the ROM locale. Note that this character encoding is not necessarily the encoding in use.

Comments The **ROM locale** is the default locale stored in the ROM. On certain ROMs, such as an EFIGS ROM, the system locale differs from the ROM locale after the user chooses a language. The ROM locale is used as the fallback overlay locale (used when the Database Manager cannot find a database for the overlay locale) unless that has been explicitly changed with the function `DmSetFallbackOverlayLocale()`. If the device is hard reset, the system locale is reset to the ROM locale.

See Also `LmGetSystemLocale()`

LmGetSystemLocale Function

Purpose Returns the system locale.

Declared In `LocaleMgr.h`

Prototype
```
CharEncodingType LmGetSystemLocale
    (LmLocaleType *oSystemLocale)
```

Parameters ← *`oSystemLocale`*

Points to an `LmLocaleType` struct that identifies the system locale. Pass `NULL` if you don't want to retrieve this value.

Returns A `CharEncodingType` constant that specifies the character encoding used in the system locale.

Comments You typically use this function only to obtain the character encoding used on the device. The system locale is used for various system settings. Applications should instead use the values stored in the user's preferences or those in the formats locale (`LmGetFormatsLocale()`), which the user can change in the Formats Preference panel.

See Also LmGetFormatsLocale(), LmGetSystemLocale()

LmISONameToCountry Function

Purpose Converts a country code string into an `LmCountryType` constant.

Declared In `LocaleMgr.h`

Prototype
```
status_t LmISONameToCountry
    (const char *iISONameStr,
    LmCountryType *oCountry)
```

Parameters → *`iISONameStr`*

A string containing a two-character ASCII ISO 3166 country code.

← *`oCountry`*

The corresponding `LmCountryType` constant.

Returns `errNone` upon success or `lmErrUnknownLocale` if *`iISONameStr`* does not contain a valid country code.

See Also `LmCountryToISOName()`, `LmISONameToLanguage()`, `LmLanguageToISOName()`

LmISONameToLanguage Function

Purpose Converts a language code string into an `LmLanguageType` constant.

Declared In `LocaleMgr.h`

Prototype
```
status_t LmISONameToLanguage
    (const char *iISONameStr,
    LmLanguageType *oLanguage)
```

Parameters

→ *iISONameStr*
A string containing a two-character ASCII ISO 639 language code.

← *oLanguage*
The corresponding `LmLanguageType` constant.

Returns `errNone` upon success or `lmErrUnknownLocale` if *iISONameStr* does not contain a valid language code.

See Also `LmISONameToCountry()`, `LmISONameToCountry()`, `LmLanguageToISOName()`

LmLanguageToISOName Function

Purpose Converts an `LmLanguageType` to a string.

Declared In `LocaleMgr.h`

Prototype
```
status_t LmLanguageToISOName
    (LmLanguageType iLanguage, char *oISONameStr)
```

Parameters

→ *iLanguage*
An `LmLanguageType` variable specifying the language code as it is stored in a locale.

← *oISONameStr*
A string that is at least three bytes long. Upon return, this string contains the language code converted to a string.

Returns `errNone` upon success or `lmErrUnknownLocale` if *iLanguage* is not a valid language code.

Comments The Database Manager uses this function to convert the overlay locale's language code into a string so that it can construct an overlay database name.

See Also `LmISONameToCountry()`, `LmISONameToCountry()`, `LmCountryToISOName()`

LmLocaleToIndex Function

Purpose Converts an `LmLocaleType` to an index suitable for passing to `LmGetLocaleSetting()`.

Declared In `LocaleMgr.h`

Prototype

```
status_t LmLocaleToIndex
    (const LmLocaleType *iLocale,
    uint16_t *oLocaleIndex)
```

Parameters

→ *`iLocale`*
The locale to convert. This locale can use the constants `lmAnyCountry` or `lmAnyLanguage` as wildcards.

← *`oLocaleIndex`*
The index of *`iLocale`* upon return.

Returns `errNone` upon success or `lmErrUnknownLocale` if the locale could not be found.

See Also `LmBestLocaleToIndex()`

LmSetFormatsLocale Function

Purpose Sets the formats locale and changes the preference settings for locale-specific items to the default values from this locale.

Declared In `LocaleMgr.h`

Prototype

```
status_t LmSetFormatsLocale
    (const LmLocaleType *iFormatsLocale)
```

Parameters

→ *`iFormatsLocale`*
An `LmLocaleType` to use for the formats locale.

Returns `errNone` upon success or `memErrNotEnoughSpace` if the Locale Manager fails to allocate memory for an internal structure identifying the preferences.

Comments The Formats Preference panel uses this function when the user selects a new country from the country pop-up list. It then changes the formats locale and all locale-dependent preferences to the settings in the first locale that it finds matching the country.

Applications should not call this function without first confirming with the user that the formats locale should be changed.

See Also `LmGetFormatsLocale()`

7

String Manager

This chapter provides reference material for the String Manager. The String Manager API is declared in the header file `StringMgr.h`. This chapter covers:

For more information, see Chapter 1, "Text," on page 3.

String Manager Constants

String Manager Constants

Purpose Constants defined in `StringMgr.h`.

Declared In `StringMgr.h`

Constants

`#define maxStrIToALen 12`
: Maximum length of a string to pass to `StrIToA()` not including the terminating null character.

`#define StrPrintF sprintf`
: Convenience macro that maps calls to the old `StrPrintF()` function to the standard C library function `sprintf()`. If you need to ensure the old functionality, use `StrPrintFV50()`.

`#define StrVPrintF vsprintf`
: Convenience macro that maps calls to the old `StrVPrintF()` function to the standard C library function `vsprintf()`. If you need to ensure the old functionality, use `StrVPrintFV50()`.

String Manager Functions and Macros

StrAToI Function

Purpose Converts a string to an integer.

Declared In `StringMgr.h`

Prototype
```
int32_t StrAToI (const char *str)
```

Parameters → *str*
A string to convert.

Returns The integer.

StrCaselessCompare Function

Purpose Compares two strings with case, size, and accent insensitivity.

Declared In `StringMgr.h`

Prototype
```
int16_t StrCaselessCompare (const char *s1,
   const char *s2)
```

Parameters → *s1*
A string.

→ *s2*
A string.

Returns 0 if the strings match.

A positive number if *s1* > *s2*.

A negative number if *s1* < *s2*.

Comments `StrCaselessCompare()` correctly performs locale-specific sorting and handles strings with multi-byte characters, whereas the standard C library function `stricmp()` does not. If the string to be compared will not be visible to the user and does not contain any locale-sensitive data, it is more efficient to use `stricmp()`.

This function differs from `TxtCaselessCompare()` in that it always compares the two strings in their entirety and does not return the length of the matching text.

See Also `StrNCaselessCompare()`, `StrCompare()`, `StrNCompare()`

StrCat Function

Purpose Concatenates one null-terminated string to another.

Declared In `StringMgr.h`

Prototype `char *StrCat (char *dst, const char *src)`

Parameters → *dst*
The null-terminated destination string.

→ *src*
The null-terminated source string.

Returns The destination string.

Comments This function calls through to the standard `strcat()` function.

StrChr Function

Purpose Looks for a character within a string.

Declared In `StringMgr.h`

Prototype `char *StrChr (const char *str, wchar32_t chr)`

Parameters → *str*
The string to be searched.

→ *chr*
The character to search for.

Returns A pointer to the first occurrence of *chr* in *str*. Returns `NULL` if the character is not found.

Comments Use this function instead of the standard `strchr()` function.

This function can handle both single-byte characters and multi-byte characters correctly. However, you should make sure that you pass a `wchar32_t` variable to `StrChr()` instead of a `char`. If you pass a `char` variable, the function sign-extends the variable to a `wchar32_t`, which causes problems if the value is 0x80 or higher.

See Also `StrStr()`

StrCompare Function

Purpose Case-sensitive comparison of two strings.

Declared In `StringMgr.h`

Prototype `int16_t StrCompare (const char *s1, const char *s2)`

Parameters

→ *s1*
A string.

→ *s2*
Another string.

Returns 0 if the strings match.

A positive number if *s1* sorts after *s2* alphabetically.

A negative number if *s1* sorts before *s2* alphabetically.

Comments `StrCompare()` correctly performs locale-specific sorting and handles strings with multi-byte characters, whereas the standard C library function `strcmp()` does not. If the string to be compared will not be visible to the user and does not contain any locale-sensitive data, it is more efficient to use `strcmp()`.

This function differs form `TxtCompare()` in that it always compares the two strings in their entirety and does not return the length of the matching text.

See Also `StrNCompare()`, `StrNCaselessCompare()`, `TxtCaselessCompare()`

StrCompareAscii Function

Purpose Compares two ASCII strings.

Declared In `StringMgr.h`

Prototype

```
int16_t StrCompareAscii (const char *s1,
   const char *s2)
```

Parameters

→ *s1*
A string.

→ *s2*
Another string.

Returns 0 if the strings match.

A positive number if *s1* sorts after *s2* alphabetically.

A negative number if *s1* sorts before *s2* alphabetically.

Comments This function calls through to the standard `strcmp()` function.

See Also `StrCompare()`, `StrNCompare()`, `TxtCompare()`, `StrCaselessCompare()`, `StrNCaselessCompare()`, `TxtCaselessCompare()`, `StrNCompareAscii()`

StrCopy Function

Purpose Copies one string to another.

Declared In `StringMgr.h`

Prototype `char *StrCopy (char *dst, const char *src)`

Parameters

→ *dst*
The destination string.

→ *src*
The source string.

Returns The destination string.

Comments This function calls through to the standard `strcpy()` function. It does not work properly with overlapping strings.

StrDelocalizeNumber Function

Purpose Delocalizes a number passed in as a string. Converts the number from any localized notation to US notation (decimal point and thousandth comma).

Declared In `StringMgr.h`

Prototype

```
char *StrDelocalizeNumber (char *s,
    char thousandSeparator, char decimalSeparator)
```

Parameters

→ *s*
The number as an ASCII string.

→ *thousandSeparator*
Current thousand separator.

→ *decimalSeparator*
Current decimal separator.

Returns A pointer to the changed number and modifies the string in *s*.

Comments The current *thousandSeparator* and *decimalSeparator* can be determined by obtaining the value of the `prefNumberFormat` preference using `PrefGetPreference()` and then passing the returned `NumberFormatType` to `LmGetNumberSeparators()`.

Example The following code shows how to use `StrDelocalizeNumber()`.

```
char *localizedNum;
NumberFormatType numFormat;
char thousandsSeparator, decimalSeparator;

numFormat = (NumberFormatType)
  PrefGetPreference(prefNumberFormat);
LmGetNumberSeparators(numFormat, &thousandsSeparator,
   &decimalSeparator);
StrDelocalizeNumber(localizedNum, thousandsSeparator,
decimalSeparator);
```

See Also `StrLocalizeNumber()`, `LmGetNumberSeparators()`

StrIToA Function

Purpose Converts an integer to ASCII.

Declared In `StringMgr.h`

Prototype `char *StrIToA (char *s, int32_t i)`

Parameters ← *s*
A string of length `maxStrIToALen+1` in which to store the results.

→ *i*
Integer to convert.

Returns The result string.

See Also `StrAToI()`, `StrIToH()`

StrIToH Function

Purpose Converts an integer to hexadecimal ASCII.

Declared In `StringMgr.h`

Prototype `char *StrIToH (char *s, uint32_t i)`

Parameters

← *s*
: A string in which to store the results.

→ *i*
: Integer to convert.

Returns The string *s*.

See Also `StrIToA()`

StrLCat Function

Purpose Concatenates one string to another, clipping the destination string to a maximum of *siz* bytes (including the null character at the end).

Declared In `StringMgr.h`

Prototype

```
size_t StrLCat (char *dst, const char *src,
    size_t siz)
```

Parameters

→ *dst*
: The null-terminated destination string.

→ *src*
: The null-terminated source string.

→ *siz*
: Maximum length in bytes for *dst*, including the terminating null character.

Returns The length in bytes of *dst* if the entire string *src* were appended to it. If *siz* is less than the return value of this function, then you know that the *src* string is truncated in *dst*.

Comments Use this function instead of the standard C library function `strlcat()`. It correctly handles multi-byte character strings. Specifically, it truncates any partial characters that appear at the end of the string and replaces them with null characters.

See Also `StrNCat()`, `StrCat()`

StrLCopy Function

Purpose Multi-byte version of the standard C library function `strlcpy()`.

Declared In `StringMgr.h`

Prototype
```
size_t StrLCopy (char *dst, const char *src,
   size_t siz)
```

Parameters

→ *dst*
The null-terminated destination string.

→ *src*
The null-terminated source string.

→ *siz*
Maximum length in bytes for *dst*, including the terminating null character.

Returns The number of bytes in the *src* string, not including the null terminator. If this value is greater than or equal to *siz*, then truncation occurred.

Comments Use this function instead of the standard C library function `strlcpy()`. It correctly handles multi-byte character strings. Specifically, it truncates any partial characters that appear at the end of the string and replaces them with null characters.

See Also `StrCopy()`, `StrNCopy()`

StrLen Function

Purpose Computes the length of a string.

Declared In `StringMgr.h`

Prototype
```
size_t StrLen (const char *src)
```

Parameters

→ *src*
A string.

Returns The length of the string in bytes.

Comments This function calls through to the standard `strlen()` function. It always correctly returns the number of bytes used to store the string. Remember that on systems that support multi-byte characters, the number returned does not always equal the number of characters.

StrLocalizeNumber Function

Purpose Converts a number (passed in as a string) to localized format, using a specified thousands separator and decimal separator.

Declared In `StringMgr.h`

Prototype
```
char *StrLocalizeNumber (char *s,
   char thousandSeparator, char decimalSeparator)
```

Parameters

↔ *s*
: Numeric ASCII string to localize. Upon return, contains the same string with all occurrences of "," replaced by *thousandSeparator* and all occurrences of "." with *decimalSeparator*.

→ *thousandSeparator*
: Localized thousand separator.

→ *decimalSeparator*
: Localized decimal separator.

Returns The changed string.

Comments The current *thousandSeparator* and *decimalSeparator* can be determined by obtaining the value of the `prefNumberFormat` preference using `PrefGetPreference()` and then passing the returned `NumberFormatType` to `LmGetNumberSeparators()`.

See Also `StrDelocalizeNumber()`

StrNCaselessCompare Function

Purpose Compares two strings out to *n* characters with case, size, and accent insensitivity.

Declared In `StringMgr.h`

Prototype
```
int16_t StrNCaselessCompare (const char *s1,
   const char *s2, size_t n)
```

Parameters

→ *s1*
: The first string.

→ *s2*
: The second string.

→ *n*
Length in bytes of the text to compare.

Returns 0 if the strings match.

A positive number if *s1* > *s2*.

A negative number if *s1* < *s2*.

Comments `StrNCaselessCompare()` correctly performs locale-specific sorting and handles strings with multi-byte characters.

This function differs from `TxtCaselessCompare()` only in that it does not return the length of the matching strings.

See Also `StrNCompare()`, `StrCaselessCompare()`, `TxtCaselessCompare()`, `StrCompare()`

StrNCat Function

Purpose Concatenates one string to another clipping the destination string to a maximum of *n* bytes (including the null character at the end).

IMPORTANT: The Palm OS® implementation of `StrNCat()` differs from the implementation in the standard C library. See the Comments section for details.

Declared In `StringMgr.h`

Prototype

```
char *StrNCat (char *dst, const char *src,
   size_t n)
```

Parameters

→ *dst*
The null-terminated destination string.

→ *src*
The source string.

→ *n*
Maximum length in bytes for *dst*, including the terminating null character.

Returns The destination string.

Comment This function differs from the standard C `strncat()` function in these ways:

- `StrNCat()` treats the parameter *n* as the maximum length in bytes for *dst*. That means it will copy at most *n* – `StrLen(`*dst*`)` – 1 bytes from *src*. The standard C function always copies *n* bytes from *src* into *dst*. (It copies the entire *src* into *dst* if the length of *src* is less than *n*).
- If the length of the destination string reaches *n* – 1, `StrNCat()` stops copying bytes from *src* and appends the terminating null character to *dst*. If the length of the destination string is already greater than or equal to *n* – 1 before the copying begins, `StrNCat()` does not copy any data from *src*.
- In the standard C function, if *src* is less than *n*, the entire *src* string is copied into *dst* and then the remaining space is filled with null characters. `StrNCat()` does not fill the remaining space with null characters in released ROMs. In debug ROMs, `StrNCat()` fills the remaining bytes with the value `0xFE`.

On systems with multi-byte character encodings, this function makes sure that it does not copy part of a multi-byte character. If the last byte copied from *src* contains the high-order or middle byte of a multi-byte character, `StrNCat()` backs up in *dst* until the byte after the end of the previous character, and replaces that byte with a null character.

StrNCompare Function

Purpose Compares two strings out to *n* bytes. This function is case and accent sensitive.

Declared In `StringMgr.h`

Prototype

```
int16_t StrNCompare (const char *s1,
    const char *s2, size_t n)
```

Parameters

→ *s1*
  A string.

→ *s2*
  A string.

→ *n*
  Length in bytes of text to compare.

Returns 0 if the strings match.

A positive number if *s1* > *s2*.

A negative number if *s1* < *s2*.

Comments `StrNCompare()` correctly performs locale-specific sorting and handles strings with multi-byte characters, whereas the standard C library function `strncmp()` does not. If the string to be compared will not be visible to the user and does not contain any locale-sensitive data, it is more efficient to use `strncmp()`.

This function differs form `TxtCompare()` only in that it does not return the length of the matching text.

See Also `StrCompare()`, `StrNCaselessCompare()`, `StrCaselessCompare()`, `TxtCaselessCompare()`, `StrNCompareAscii()`

StrNCompareAscii Function

Purpose Compares two ASCII strings out to *n* bytes. This function calls through to the standard `strncmp()` function.

Declared In `StringMgr.h`

Prototype

```
int16_t StrNCompareAscii (const char *s1,
   const char *s2, size_t n)
```

Parameters

→ *s1*
A string.

→ *s2*
A string.

→ *n*
Length in bytes of text to compare.

Returns 0 if the strings match.

A positive number if *s1* sorts after *s2* alphabetically.

A negative number if *s1* sorts before *s2* alphabetically.

StrNCopy Function

Purpose Copies up to *n* bytes from a source string to the destination string. Terminates *dst* string at index *n*–1 if the source string length was *n*–1 or less.

Declared In `StringMgr.h`

Prototype
```
char *StrNCopy (char *dst, const char *src,
    size_t n)
```

Parameters

→ *dst*
The destination string.

→ *src*
The source string.

→ *n*
Maximum number of bytes to copy from *src* string.

Returns The destination string.

Comments On systems with multi-byte character encodings, this function makes sure that it does not copy part of a multi-byte character. If the *n*th byte of *src* contains the high-order or middle byte of a multi-byte character, `StrNCopy()` backs up in *dst* until the byte after the end of the previous character and replaces the remaining bytes (up to *n*–1) with nulls.

Be aware that the *n*th byte of *dst* upon return may contain the last byte of a multi-byte character. If you plan to terminate the string by setting its last character to null, you must not pass the entire length of the string to `StrNCopy()`. If you do, your code may overwrite the final byte of the last character.

```
// WRONG! You may overwrite part of multi-byte
// character.
char dst[n];
StrNCopy(dst, src, n);
dst[n-1] = chrNull;
```

Instead, if you write to the last byte of the destination string, pass one less than the size of the string to `StrNCopy()`.

```
// RIGHT. Instead pass n-1 to StrNCopy.
char dst[n];
StrNCopy(dst, src, n-1);
dst[n-1] = chrNull;
```

StrPrintFV50 Function

Purpose Implements a subset of the standard C `sprintf()` call, which writes formatted output to a string.

Declared In `StringMgr.h`

Prototype
```
int16_t StrPrintFV50 (char *s,
   const char *formatStr, ...)
```

Parameters

← *s*
A string into which the results are written.

→ *formatStr*
The format specification string.

`...`
Zero or more arguments to be formatted as specified by *formatStr*.

Returns Number of characters written to destination string. Returns a negative number if there is an error.

Comments This function internally calls `StrVPrintFV50()` to do the formatting. See that function's description for details on which format specifications are supported.

Compatibility This function is obsolete and provided for backward compatibility only. Use the standard C library function `sprintf()` instead.

See Also `StrVPrintFV50()`

StrStr Function

Purpose Looks for a substring within a string.

Declared In `StringMgr.h`

Prototype
```
char *StrStr (const char *str, const char *token)
```

Parameters

→ *str*
The string to be searched.

→ *token*
The string to search for.

Returns A pointer to the first occurrence of *token* in *str* or `NULL` if not found.

Comments Use this function instead of the standard `strstr()` function to handle multi-byte character strings. This function makes sure that it does not match only part of a multi-byte character. If the matching strings begins at an inter-character boundary, then this function returns `NULL`.

> **NOTE:** If the value of the *`token`* parameter is the empty string, this function returns `NULL`. This is different than the standard `strstr()` function, which returns *`str`* when *`token`* is the empty string.

See Also `StrChr()`

StrToLower Function

Purpose Converts all the characters in a string to lowercase.

Declared In `StringMgr.h`

Prototype
```
char *StrToLower (char *dst, const char *src)
```

Parameters

← *`dst`*
A string.

→ *`src`*
A null-terminated string.

Returns The destination string.

StrVPrintFV50 Function

Purpose Implements a subset of the standard C `vsprintf()` call, which writes formatted output to a string.

Declared In `StringMgr.h`

Prototype
```
int16_t StrVPrintFV50 (char *s,
    const char *formatStr, _Palm_va_list arg)
```

Parameters

← *`s`*
A string into which the results are written. This string is always terminated by a null terminator.

→ *formatStr*
The format specification string.

→ *arg*
Pointer to a list of zero or more parameters to be formatted as specified by the *formatStr* string.

Returns Number of characters written to destination string, not including the null terminator. Returns a negative number if there is an error.

Comments Like the C `vsprintf()` function, this function is designed to be called by your own function that takes a variable number of arguments and passes them to this function.

Currently, only the conversion specifications `%d`, `%i`, `%u`, `%x`, `%s`, and `%c` are implemented by `StrVPrintF()` (and related functions). Optional modifiers that are supported include: `+`, `-`, <space>, `*`, <digits>, `h` and `l` (long). Refer to a C reference book for more details on how these conversion specifications work.

Compatibility This function is obsolete and provided for backward compatibility only. Use the standard C library function `vsprintf()` instead.

Example The following code sample shows how to use this call:

```
#include <unix_stdarg.h>
void MyPrintF(char *s, char *formatStr, ...)
{
  va_list args;
  char text[0x100];
  va_start(args, formatStr);
  StrVPrintFV50(text, formatStr, args);
  va_end(args);
  MyPutS(text);
}
```

See Also `StrPrintFV50()`

8

Text Manager

This chapter provides information about the Text Manager API declared in `TextMgr.h` by discussing these topics:

For more information on the Text Manager, see the chapter "Text" on page 3.

Text Manager Structures and Types

CharEncodingType Typedef

Purpose Specifies possible character encodings.

Declared In `TextMgr.h`

Prototype `typedef uint16_t CharEncodingType`

Comments A given device supports a single character encoding. Palm OS® Cobalt devices support either the Palm OS version of Windows code page 1252[1] (an extension of ISO Latin 1) or the Palm OS version of Windows code page 932[1] (an extension of Shift JIS). In addition, Palm OS licensees and some third-party developers provide support for additional character encodings including Big-5, Hebrew, Arabic, Thai, Korean, and Cyrillic.

The character encoding constants generally follow the format:

`charEncoding`*`Name`*

1. This encoding is identical to its Windows counterpart with some additional characters added in the control range.

where *Name* is the name of the character encoding.

The following table shows examples of the character encoding constants. For a complete list, see the `TextMgr.h` file.

Constant	Description
`charEncodingUnknown`	Unknown to this version of Palm OS
`charEncodingAscii`	ISO 646-1991
`charEncodingISO8859_1`	ISO 8859 Part 1 (also known as ISO Latin 1). This encoding is commonly used for the Roman alphabet
`charEncodingPalmLatin`	Palm OS version of Microsoft Windows code page 1252. This encoding is identical to code page 1252 with Palm-specific characters added in the control range.
`charEncodingShiftJIS`	Encoding for 0208-1990 with single-byte Japanese Katakana. This encoding is commonly used for Japanese alphabets.
`charEncodingPalmSJIS`	Palm OS version of Microsoft Windows code page 932. This encoding is identical to code page 932, with Palm-specific characters added in the control range and with a Yen symbol instead of the Reverse Solidus at location 0x5c.
`charEncodingCP1252`	Microsoft Windows extensions to ISO 8859 Part 1

Constant	Description
`charEncodingCP932`	Microsoft Windows extensions to Shift JIS
`charEncodingUTF8`	Eight-bit safe encoding for Unicode

TxtConvertStateType Struct

Purpose Maintains state across calls to `TxtConvertEncoding()`. It is essentially opaque; simply declare a structure of this type and pass a pointer to your structure when making multiple calls to `TxtConvertEncoding()` for a single source text buffer.

Declared In `TextMgr.h`

Prototype

```
typedef struct {
   uint8_t ioSrcState[kTxtConvertStateSize];
   uint8_t ioDstState[kTxtConvertStateSize];
} TxtConvertStateType
```

Comments `kTxtConvertStateSize` is simply a constant that determines the size of the source and destination state buffers.

Text Manager Constants

Byte Attribute Flags

Purpose Flags that identify the possible locations of a given byte within a multi-byte character.

Declared In `TextMgr.h`

Constants

`#define byteAttrFirst 0x80`
First byte of multi-byte character.

`#define byteAttrHighLow (byteAttrFirst | byteAttrLast)`
Either the first byte of a multi-byte character or the last byte of a multi-byte character.

`#define byteAttrLast 0x40`
Last byte of multi-byte character.

`#define byteAttrMiddle 0x20`
Middle byte of multi-byte character.

`#define byteAttrSingle 0x01`
Single-byte character.

`#define byteAttrSingleLow (byteAttrSingle | byteAttrLast)`
Either a single-byte character or the low-order byte of a multi-byte character.

Comments If a byte is valid in more than one location of a character, multiple return bits are set. For example, 0x40 in the Shift JIS character encoding is valid as a single-byte character and as the low-order byte of a double-byte character. Thus, the return value for `TxtByteAttr(0x40)` on a Shift JIS system has both the `byteAttrSingle` and `byteAttrLast` bits set.

Every byte in a stream of double-byte data must be either a single byte, a high byte, a single/low byte (`byteAttrSingleLow`), or a high/low byte (`byteAttrHighLow`).

See Also `TxtByteAttr()`

Character Attributes

Purpose Flags that identify various character attributes.

Declared In `TextMgr.h`

Constants `#define charAttrAlNum (charAttr_DI | charAttr_LO | charAttr_UP | charAttr_XA)`
Alphanumeric characters

`#define charAttrAlpha (charAttr_LO | charAttr_UP | charAttr_XA)`
Alphabetic characters

`#define charAttrCntrl (charAttr_BB | charAttr_CN)`
Control characters

`#define charAttrDelim (charAttr_SP | charAttr_PU)`
Delimiters

```
#define charAttrGraph (charAttr_DI | charAttr_LO |
  charAttr_PU | charAttr_UP | charAttr_XA)
```
Printable, non-space characters

```
#define charAttrPrint (charAttr_DI | charAttr_LO |
  charAttr_PU | charAttr_SP | charAttr_UP |
  charAttr_XA)
```
Printable characters

```
#define charAttrSpace (charAttr_CN | charAttr_SP |
  charAttr_XS)
```
Whitespace characters

`#define charAttr_BB 0x00000080`
BEL, BS, etc.

`#define charAttr_CN 0x00000040`
CR, FF, HT, NL, VT

`#define charAttr_DI 0x00000020`
'0'-'9'

`#define charAttr_DO 0x00000400`
Characters that appear on the display but never in user data, such as the ellipsis character

`#define charAttr_LO 0x00000010`
'a'-'z' and lowercase extended characters

`#define charAttr_PU 0x00000008`
Punctuation

`#define charAttr_SP 0x00000004`
Space

`#define charAttr_UP 0x00000002`
'A'-'Z' and uppercase extended characters

`#define charAttr_XA 0x00000200`
Extra alphabetic

`#define charAttr_XD 0x00000001`
'0'-'9', 'A'-'F', 'a'-'f'

`#define charAttr_XS 0x00000100`
Extra space

Character Encoding Attributes

Purpose Constants used to interpret the return value of `TxtGetEncodingFlags()`.

Declared In `TextMgr.h`

Constants `#define charEncodingOnlySingleByte 0x00000001`
The character encoding consists only of single-byte characters.

`#define charEncodingHasDoubleByte 0x00000002`
The character encoding contains one or more double-byte characters.

`#define charEncodingHasLigatures 0x00000004`
The character encoding has ligatures.

`#define charEncodingRightToLeft 0x00000008`
The character encoding supports a writing system that primarily renders text right-to-left.

Encoding Conversion Constant Modifiers

Purpose Constants to OR with the destination character encoding (`CharEncodingType`) passed to `TxtConvertEncoding()`.

Declared In `TextMgr.h`

Constants `#define charEncodingDstBestFitFlag 0x8000`
Causes `TxtConvertEncoding()` to make an extra effort to convert characters in the source encoding to similar (if not equal) characters in the destination encoding.

Comments As an example, when converting from `charEncodingUCS2` to `charEncodingPalmSJIS`, no mapping exists for `U+00A1` (INVERTED EXCLAMATION MARK) because this character doesn't exist in `charEncodingPalmSJIS`. In this case, `TxtConvertEncoding()` returns `txtErrNoCharMapping`. If you OR the `charEncodingDstBestFitFlag` with the destination character encoding, however, `TxtConvertEncoding()` converts the character to `chrExclamationMark` (which is close). Generally, the operating system tries to support as many code page 1252 characters as possible in the "best fit" table.

If `charEncodingDstBestFitFlag` is set and either the source or destination encoding is unknown, `TxtConvertEncoding()` copies anything that is 7-bit ASCII from the source to the destination. It then returns `txtErrUnknownEncodingFallbackCopy`. The rules for unknown characters apply during this 7-bit copy; if an inconvertible character is encountered, the substitution string (if one has been specified) is used in its place, and `txtErrNoCharMapping` is returned instead.

Encoding Conversion Substitution Constants

Purpose Values used to substitute in `TxtConvertEncoding()`.

Declared In `TextMgr.h`

Constants `#define textSubstitutionDefaultLen 1`
The length in bytes of `textSubstitutionDefaultStr`.

`#define textSubstitutionDefaultStr "?"`
Can be passed to `TxtConvertEncoding()` as the substitution string parameter. The substitution string contains a character that is used in the destination string if a character from the source string is not recognized in the destination encoding.

`#define textSubstitutionEncoding charEncodingUTF8`
The encoding used for the substitution string parameter of `TxtConvertEncoding()`. The string you pass for the substitution string parameter is always assumed to be in this encoding.

Size Constants

Purpose Constants that specify sizes of items used in the Text Manager.

Declared In `TextMgr.h`

Constants `#define kTxtConvertStateSize 32`
Used in the `TxtConvertStateType` structure to specify the maximum size of the source and destination encodings.

`#define maxCharBytes 4`
Maximum size a single `wchar32_t` character will occupy in a text string.

`#define maxEncodingNameLength 40`
Maximum length in bytes of any character encoding name.

Text Manager Error Constants

Purpose Error constants.

Declared In `TextMgr.h`

Constants

`#define txtErrConvertOverflow (txtErrorClass | 4)`
The destination buffer is not large enough to contain the converted text.

`#define txtErrConvertUnderflow (txtErrorClass | 5)`
The end of the source buffer contains a partial character.

`#define txtErrMalformedText (txtErrorClass | 9)`
An error in the source text encoding has been discovered.

`#define txtErrNoCharMapping (txtErrorClass | 7)`
The device does not contain a mapping between the source and destination encodings for at least one of the characters in the source string.

`#define txtErrTranslitOverflow (txtErrorClass | 3)`
The destination buffer is not large enough to contain the converted string.

`#define txtErrTranslitOverrun (txtErrorClass | 2)`
The source and destination buffers point to the same memory location and performing the requested operation would cause the function to overwrite unprocessed data in the input buffer.

`#define txtErrTranslitUnderflow (txtErrorClass | 8)`
The end of the source buffer contains a partial character.

`#define txtErrUknownTranslitOp (txtErrorClass | 1)`
The transliteration operation constant value is not recognized

`#define txtErrUnknownEncoding (txtErrorClass | 6)`
One of the specified encodings is unknown or can't be handled.

`#define txtErrUnknownEncodingFallbackCopy (txtErrorClass | 10)`
Either the source or destination encoding is unknown, and the best fit flag was set in the destination encoding.

Text Manager Feature Settings

Purpose Text Manager settings that can be obtained or set in the `sysFtrNumTextMgrFlags` feature.

Declared In `TextMgr.h`

Constants

`#define textMgrBestFitFlag 0x00000004`
The `TxtConvertEncoding()` function can use the `charEncodingDstBestFitFlag`. See "Encoding Conversion Constant Modifiers" on page 90 for more information. This flag is always set in Palm OS Cobalt.

`#define textMgrExistsFlag 0x00000001`
The Text Manager is installed on the device. This flag is always set in Palm OS Cobalt.

`#define textMgrStrictFlag 0x00000002`
No longer used.

TranslitOpType Typedef

Purpose Specifies the transliteration operation to be performed by a given call to `TxtTransliterate()`. Each character encoding contains its own set of special transliteration operations, the values for which begin at `translitOpCustomBase`.

Declared In `TextMgr.h`

Prototype `typedef uint16_t TranslitOpType`

Constants

`#define translitOpStandardBase 0`
Base value at which character-encoding-independent transliterations are defined.

`#define translitOpUpperCase 0`
Convert all characters to uppercase.

`#define translitOpLowerCase 1`
Convert all characters to lowercase.

`#define translitOpReserved2 2`
Reserved for future use.

`#define translitOpReserved3 3`
Reserved for future use.

`#define translitOpPreprocess 0x8000`
OR this value with another transliteration flag to have the `TxtTransliterate()` function return the space requirements for the result.

`#define translitOpCustomBase 1000`
Base value at which character-encoding specific transliteration constants begin.

Text Manager Functions and Macros

CHAR_ENCODING_VALUE Macro

Purpose Macro used to set the values of the character encoding constants.

Declared In `TextMgr.h`

Prototype `#define CHAR_ENCODING_VALUE (value)`

Parameters → *value*
An integer value.

Returns A `CharEncodingType` value.

Comments Applications do not need to use this macro.

sizeOf7BitChar Macro

Purpose Returns the true size of a low-ASCII character.

Declared In `TextMgr.h`

Prototype `#define sizeOf7BitChar (c)`

Parameters → *c*
A character constant.

Returns The value 1.

Comments In C, checking the size of a character constant returns the size of an integer. For example, `sizeof('a')` returns 2. Because of this, it's safest to use the `sizeOf7BitChar()` macro to document buffer size and string length calculations. Note that this can only be used with low-ASCII characters, as anything else might be the high byte of a double-byte character.

TxtByteAttr Function

Purpose Returns the possible locations of a given byte within a multi-byte character.

Declared In `TextMgr.h`

Prototype `uint8_t TxtByteAttr (uint8_t iByte)`

Parameters → *iByte*

A byte representing all or part of a valid character.

Returns A byte with one or more of the Byte Attribute Flags set.

Comments Text Manager functions that need to determine the byte positioning of a character use `TxtByteAttr()` to do so. You rarely need to use this function yourself.

TxtCaselessCompare Function

Purpose Performs a case-insensitive comparison of two text buffers.

Declared In `TextMgr.h`

Prototype

```
int16_t TxtCaselessCompare (const char *s1,
    size_t s1Len, size_t *s1MatchLen,
    const char *s2, size_t s2Len,
    size_t *s2MatchLen)
```

Parameters → *s1*

The first text buffer to compare.

→ *s1Len*

The length in bytes of the text pointed to by *s1*.

← *s1MatchLen*
Points to the offset of the first character in *s1* that determines the sort order. Pass `NULL` for this parameter if you don't need to know this number.

→ *s2*
The second text buffer to compare.

→ *s2Len*
The length in bytes of the text pointed to by *s2*.

← *s2MatchLen*
Points to the offset of the first character in *s2* that determines the sort order. Pass `NULL` for this parameter if you don't need to know this number.

Returns One of the following values:

< 0 If *s1* occurs before *s2* in alphabetical order.

> 0 If *s1* occurs after *s2* in alphabetical order.

0 If the two substrings that were compared are equal.

Comments In certain character encodings (such as Shift JIS), one character may be accurately represented as either a single-byte character or a multi-byte character. `TxtCaselessCompare()` accurately matches a single-byte character with its multi-byte equivalent. For this reason, the values returned in *s1MatchLen* and *s2MatchLen* are not always equal.

You must make sure that the parameters *s1* and *s2* point to the start of a valid character. That is, they must point to the first byte of a multi-byte character or they must point to a single-byte character; if they don't, results are unpredictable.

See Also `StrCaselessCompare()`, `TxtCompare()`, `StrCompare()`

TxtCharAttr Function

Purpose Returns a character's attributes.

Declared In `TextMgr.h`

Prototype `uint32_t TxtCharAttr (wchar32_t iChar)`

Parameters → *iChar*
Any valid character.

Returns An integer with any of the Character Attributes bits set.

Comments The character passed to this function must be a valid character given the system encoding.

This function is used in the Text Manager's character attribute macros (`TxtCharIsAlNum()`, `TxtCharIsCntrl()`, and so on). The macros perform operations analogous to the standard C functions `isPunct()`, `isPrintable()`, and so on. Usually, you'd use one of these macros instead of calling `TxtCharAttr()` directly.

To obtain attributes specific to a given character encoding, use `TxtCharXAttr()`.

See Also `TxtCharIsValid()`

TxtCharBounds Function

Purpose Returns the boundaries of a character containing the byte at a specified offset in a string.

Declared In `TextMgr.h`

Prototype
```
wchar32_t TxtCharBounds (const char *iTextP,
   size_t iOffset, size_t *oCharStart,
   size_t *oCharEnd)
```

Parameters → *iTextP*
The text buffer to search.

→ *iOffset*
A valid offset into the buffer *iTextP*. This location may contain a byte in any position (start, middle, or end) of a multi-byte character.

← *oCharStart*
Points to the starting offset of the character containing the byte at *iOffset*.

← *oCharEnd*
Points to the ending offset of the character containing the byte at *iOffset*.

Returns The character located between the offsets *oCharStart* and *oCharEnd*.

Comments Use this function to determine the boundaries of a character in a string or text buffer.

`TxtCharBounds()` is often slow and should be used only where needed. If the byte at *iOffset* is valid in more than one location of a character, the function must search back toward the beginning of the text buffer until it finds an unambiguous byte to determine the appropriate boundaries.

You must make sure that the parameter *iTextP* points to the beginning of the string. That is, if the string begins with a multi-byte character, *iTextP* must point to the first byte of that character; if it doesn't, results are unpredictable.

TxtCharEncoding Function

Purpose Returns the minimum encoding required to represent a character.

Declared In `TextMgr.h`

Prototype `CharEncodingType TxtCharEncoding (wchar32_t iChar)`

Parameters → *iChar*
A valid character.

Returns A `CharEncodingType` value that indicates the minimum encoding required to represent *iChar*. If the character isn't recognizable, `charEncodingUnknown` is returned.

Comments The minimum encoding is the encoding that represents the fewest number of characters while still containing the character specified in *iChar*. For example, if the character is a blank or a tab character, the minimum encoding is `charEncodingAscii` because these characters can be represented in single-byte ASCII. If the character is a ü, the minimum encoding is `charEncodingISO8859_1`.

This function is used by `TxtStrEncoding()`, which is the function that most applications should use to determine the character encoding for tagging text (for instance, for email).

Use `TxtMaxEncoding()` to determine the order of encodings.

Palm OS only supports a single character encoding at a time. Because of this, the result of `TxtCharEncoding()` is always logically equal to or less than the encoding used on the current system. That is, you'll only receive a return value of `charEncodingISO8859_1` if you're running on a US or European system and you pass a non-ASCII character.

See Also `TxtStrEncoding()`, `TxtMaxEncoding()`

TxtCharIsAlNum Macro

Purpose Indicates if the character is alphanumeric.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsAlNum (`*`ch`*`)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a letter in an alphabet or a numeric digit, `false` otherwise.

See Also `TxtCharIsDigit()`, `TxtCharIsAlpha()`

TxtCharIsAlpha Macro

Purpose Indicates if a character is a letter in an alphabet.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsAlpha (`*`ch`*`)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a letter in an alphabet, `false` otherwise.

See Also `TxtCharIsAlNum()`, `TxtCharIsLower()`, `TxtCharIsUpper()`

TxtCharIsCntrl Macro

Purpose Indicates if a character is a control character.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsCntrl (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a non-printable character, such as the bell character or a carriage return; `false` otherwise.

TxtCharIsDelim Macro

Purpose Indicates if a character is a delimiter.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsDelim (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a word delimiter (whitespace or punctuation), `false` otherwise.

TxtCharIsDigit Macro

Purpose Indicates if the character is a decimal digit.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsDigit (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is 0 through 9, `false` otherwise.

See Also `TxtCharIsAlNum()`, `TxtCharIsHex()`

TxtCharIsGraph Macro

Purpose Indicates if a character is a graphic character.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsGraph (ch)`

Parameters → *ch*

A valid character.

Returns `true` if the character is a graphic character, `false` otherwise.

Comments A graphic character is any character visible on the screen, in other words, letters, digits, and punctuation marks. A blank space is not a graphic character because it is not visible.

This macro differs from `TxtCharIsPrint()` in that it returns `false` if the character is whitespace. `TxtCharIsPrint()` returns `true` if the character is whitespace.

TxtCharIsHardKey Macro

Purpose Returns true if the character is one of the hard keys on the device.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsHardKey (m, c)`

Parameters → *m*

The value passed in the `modifiers` field of the `keyDownEvent`.

→ *c*

The character from the `keyDownEvent`.

Returns `true` if the character is one of the built-in hard keys on the device, `false` otherwise.

TxtCharIsHex Macro

Purpose Indicates if a character is a hexadecimal digit.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsHex (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a hexadecimal digit from 0 to F, `false` otherwise.

See Also `TxtCharIsDigit()`

TxtCharIsLower Macro

Purpose Indicates if a character is a lowercase letter.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsLower (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a lowercase letter, `false` otherwise.

See Also `TxtCharIsAlpha()`, `TxtCharIsUpper()`

TxtCharIsPrint Macro

Purpose Indicates if a character is printable.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsPrint (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is not a control character, `false` otherwise.

Comments This macro differs from `TxtCharIsGraph()` in that it returns `true` if the character is whitespace. `TxtCharIsGraph()` returns `false` if the character is whitespace.

If you are using a debug ROM and you pass a virtual character to this macro, a fatal alert is generated.

See Also `TxtCharIsValid()`

TxtCharIsPunct Macro

Purpose Indicates if a character is a punctuation mark.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsPunct (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is a punctuation mark, `false` otherwise.

TxtCharIsSpace Macro

Purpose Indicates if a character is a whitespace character.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsSpace (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is whitespace such as a blank space, tab, or newline; `false` otherwise.

TxtCharIsUpper Macro

Purpose Indicates if a character is an uppercase letter.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsUpper (ch)`

Parameters → *ch*
A valid character.

Returns `true` if the character is an uppercase letter, `false` otherwise.

See Also `TxtCharIsAlpha()`, `TxtCharIsLower()`

TxtCharIsValid Function

Purpose Determines whether a character is valid given the Palm OS character encoding.

Declared In `TextMgr.h`

Prototype `Boolean TxtCharIsValid (wchar32_t iChar)`

Parameters → *`iChar`*
A character.

Returns `true` if *`iChar`* is a valid character; `false` if *`iChar`* is not a valid character.

See Also `TxtCharAttr()`, `TxtCharIsPrint()`

TxtCharIsVirtual Macro

Purpose Returns whether a character is a virtual character or not.

Declared In `TextMgr.h`

Prototype `#define TxtCharIsVirtual (m, c)`

Parameters → *`m`*
The value passed in the `modifiers` field of the `keyDownEvent`.

→ *`c`*
The character from the `keyDownEvent`.

Returns `true` if the character *`c`* is a virtual character, `false` otherwise.

Comments Virtual characters are nondisplayable characters that trigger special events in the operating system, such as displaying low battery warnings or displaying the keyboard dialog. Virtual characters should never occur in any data and should never appear on the screen.

TxtCharSize Function

Purpose Returns the number of bytes required to store the character in a string.

Declared In `TextMgr.h`

Prototype `size_t TxtCharSize (wchar32_t iChar)`

Parameters → *iChar*
A valid character.

Returns The number of bytes required to store the character in a string.

Comments Although character variables are always multi-byte long `wchar32_t` values, in some character encodings such as Shift JIS, characters in strings are represented by a mix of one or more bytes per character. If the character can be represented by a single byte (its high-order bytes are 0), it is stored in a string as a single-byte character.

See Also `TxtCharBounds()`

TxtCharXAttr Function

Purpose Returns the extended attribute bits for a character.

Declared In `TextMgr.h`

Prototype `uint32_t TxtCharXAttr (wchar32_t iChar)`

Parameters → *iChar*
A valid character.

Returns An unsigned 32-bit value with one or more extended attribute bits set. For specific return values, look in the header files that are specific to certain character encodings (`CharLatin.h` or `CharShiftJIS.h`).

Comments To interpret the results, you must know the character encoding being used. The function `LmGetSystemLocale()` returns the character encoding used on the device as one of the `CharEncodingType` values. You can pass `NULL` as the parameter to `LmGetSystemLocale()` if you don't want to retrieve any other locale information.

See Also `TxtCharAttr()`, "Retrieving the Character Encoding"

TxtCompare Function

Purpose Performs a case-sensitive comparison of all or part of two text buffers.

Declared In `TextMgr.h`

Prototype

```
int16_t TxtCompare (const char *s1, size_t s1Len,
   size_t *s1MatchLen, const char *s2,
   size_t s2Len, size_t *s2MatchLen)
```

Parameters

→ *s1*
The first text buffer to compare.

→ *s1Len*
The length in bytes of the text pointed to by *s1*.

← *s1MatchLen*
Points to the offset of the first character in *s1* that determines the sort order. Pass `NULL` for this parameter if you don't need to know this number.

→ *s2*
The second text buffer to compare.

→ *s2Len*
The length in bytes of the text pointed to by *s2*.

← *s2MatchLen*
Points to the offset of the first character in *s2* that determines the sort order. Pass `NULL` for this parameter if you don't need to know this number.

Returns One of the following values:

< 0	If *s1* occurs before *s2* in alphabetical order.
> 0	If *s1* occurs after *s2* in alphabetical order.
0	If the two substrings that were compared are equal.

Comments This function performs a case-sensitive comparison. If you want to perform a case-insensitive comparison, use `TxtCaselessCompare()`.

The *s1MatchLen* and *s2MatchLen* parameters are not as useful for the `TxtCompare()` function as they are for the `TxtCaselessCompare()` function because `TxtCompare()` implements a multi-pass sort algorithm. For example, if you use

`TxtCaselessCompare()` to compare the string "celery" with the string "Cauliflower," it returns a positive value to indicate that "celery" sorts after "Cauliflower," and it returns a match length of 1 to indicate that the second letter determines the sort order ("e" comes after "a"). However, because `TxtCompare()` ultimately does a case-sensitive comparison, comparing the string "c" to the string "C" produces a negative result and a match length of 0.

In certain character encodings (such as Shift JIS), one character may be accurately represented as either a single-byte character or a multi-byte character. `TxtCompare()` accurately matches a single-byte character with its multi-byte equivalent. For this reason, the values returned in *s1MatchLen* and *s2MatchLen* are not always equal.

You must make sure that the parameters *s1* and *s2* point to the start of a a valid character. That is, they must point to the first byte of a multi-byte character or they must point to a single-byte character; if they don't, results are unpredictable.

See Also `StrCompare()`, `TxtFindString()`

TxtConvertEncoding Function

Purpose Converts a text buffer from one character encoding to another.

Declared In `TextMgr.h`

Prototype

```
status_t TxtConvertEncoding(Boolean newConversion,
    TxtConvertStateType *ioStateP,
    const char *srcTextP, size_t *ioSrcBytes,
    CharEncodingType srcEncoding, char *dstTextP,
    size_t *ioDstBytes,
    CharEncodingType dstEncoding,
    const char *substitutionStr,
    size_t substitutionLen)
```

Parameters

→ *newConversion*

Set to `true` if this function call is starting a new conversion, or `false` if this function call is a continuation of a previous conversion.

↔ *ioStateP*
: If *newConversion* is `false`, this parameter must point to a `TxtConvertStateType` structure containing the same data used for the previous invocation. If *newConversion* is `true` and no subsequent calls are planned, this parameter can be `NULL`.

→ *srcTextP*
: The source text buffer. If *newConversion* is `true`, this must point to the start of a text buffer. If *newConversion* is `false`, it may point to a location in the middle of a text buffer. In either case, it must point to an inter-character boundary.

↔ *ioSrcBytes*
: A pointer to the size, in bytes, of the text starting at *srcTextP* that needs to be converted. Upon return, **ioSrcBytes* contains the number of bytes successfully processed.

  If *srcTextP* is null-terminated and you want *dstTextP* to be null terminated, include a byte for the null terminator in this size.

→ *srcEncoding*
: The character encoding that the source text uses. See `CharEncodingType`.

↔ *dstTextP*
: The destination text buffer, which must be large enough to hold the result of converting *srcTextP* to the specified encoding. You can pass `NULL` for the *dstTextP* parameter to determine the required length of the buffer before actually doing the conversion; the required length is returned in *ioDstBytes*.

  `TxtConvertEncoding()` does not write the terminating null character to *dstTextP* unless one is present in *srcTextP* and *ioSrcBytes* includes space for it.

↔ *ioDstBytes*
: A pointer to the length, in bytes, of *dstTextP*. Upon return, **ioDstBytes* contains the number of bytes required to represent the source text in the new encoding.

→ *dstEncoding*
: The character encoding to which to convert *srcTextP*. See CharEncodingType for a description of the possible values. Note that the encoding can be modified, giving you greater control over the conversion process; see "Encoding Conversion Constant Modifiers" on page 90.

→ *substitutionStr*
: A string to be substituted for any invalid or inconvertible characters that occur in the source text. This string must be valid in the encoding specified by the constant `textSubstitutionEncoding`. If this parameter is `NULL`, `TxtConvertEncoding()` immediately returns if it encounters an invalid character.

You can pass the constant `textSubstitutionDefaultStr` for this parameter to have a question mark used as the substitution string.

→ *substitutionLen*
: The number of bytes in *substitutionStr*, not including the terminating null byte.

If you use `textSubstitutionDefaultStr` for *substitutionStr*, use `textSubstitutionDefaultLen` for this parameter.

Returns `errNone` upon success or one of the following if an error occurs:

`txtErrConvertOverflow`
: The destination buffer is not large enough to contain the converted text.

`txtErrConvertUnderflow`
: The end of the source buffer contains a partial character.

`txtErrMalformedText`
: An error in the source text encoding has been discovered.

`txtErrNoCharMapping`
: The device does not contain a mapping between the source and destination encodings for at least one of the characters in *srcTextP*.

`txtErrUnknownEncoding`
: One of the specified encodings is unknown or can't be handled.

`txtErrUnknownEncodingFallbackCopy`
Either the source or destination encoding is unknown, and the best fit flag was set in the destination encoding. Before returning this error code, `TxtConvertEncoding()` copies anything that is 7-bit ASCII from the source text buffer to the destination text buffer.

Comments This function converts *`ioSrcBytes`* of text in *`srcTextP`* from the *`srcEncoding`* to the *`dstEncoding`* character encoding and returns the result in *`dstTextP`*.

The supported encodings for *`srcEncoding`* and *`dstEncoding`* are locale-dependent. See "Encodings Supported by Various Locales" on page 112. However, this function is most commonly used to convert between an encoding used on the Internet and the device's encoding; therefore, all locales support conversions between most Unicode character sets and the device's encoding. If you use any of the following character encodings, the conversion should work:

- The device's character encoding as returned by the function `LmGetSystemLocale()`
- Any of the following, which can be retrieved using `LmGetLocaleSetting()`:
  - `lmChoiceInboundDefaultVObjectEncoding` (as *`srcEncoding`* only)
  - `lmChoicePrimarySMSEncoding` (as *`dstEncoding`* only)
  - `lmChoiceSecondarySMSEncoding` (as *`dstEncoding`* only)
  - `lmChoicePrimaryEmailEncoding` (as *`dstEncoding`* only)
  - `lmChoiceSecondaryEmailEncoding` (as *`dstEncoding`* only)
  - `lmChoiceOutboundVObjectEncoding` (as *`dstEncoding`* only)

TIP: If you're converting text that was received from the Internet, the encoding name is passed along with the text data. Use the `TxtNameToEncoding()` function to convert the name to a `CharEncodingType` value.

If the function encounters an inconvertible character in the source text, it puts *`substitutionStr`* in the destination buffer in that character's place and continues the conversion. When the conversion is complete, it returns `txtErrNoCharMapping` to indicate that an error occurred (assuming that no other higher-priority error occurred during the conversion). If *`substitutionStr`* is `NULL`, the function stops the conversion and immediately returns `txtErrNoCharMapping`. *`ioSrcBytes`* is set to the offset of the inconvertible character, *`dstTextP`* contains the converted string up to that point, and *`ioDstBytes`* contains the size of the converted text. You can examine the character at *`ioSrcBytes`* and choose to move past it and continue the conversion. Follow the rules for making repeated calls to `TxtConvertEncoding()` as described below.

Calling TxtConvertEncoding() in a Loop

You can make repeated calls to `TxtConvertEncoding()` in a loop if you only want to convert part of the input buffer at a time. When you make repeated calls to this function, the first call should use `true` for *`newConversion`*, and *`srcTextP`* should point to the start of the text buffer. All subsequent calls should use the following values:

`newConversion`

`false`.

`ioStateP`

The same data that was returned by the previous invocation.

`srcTextP`

The location where this call should begin converting. Typically, this would be the previous *`srcTextP`* plus the number of bytes returned in *`ioSrcBytes`*.

If you are skipping over an inconvertible character, *`srcTextP`* must point to the character after that location.

ioSrcBytes
The number of bytes that this function call should convert.

dstTextP
A pointer to a location where this function can begin writing the converted string. You might choose to have each function call write to a different destination buffer. To have successive calls write to the same buffer, pass the previous *dstTextP* plus the number of bytes returned in *ioDstBytes* each time.

ioDstBytes
The number of bytes available for output in the *dstTextP* buffer. In other words, the number of bytes remaining.

Encodings Supported by Various Locales

Each device's ROM contains a system-use only locale module that contains tables `TxtConvertEncoding()` uses to convert one encoding to another. Therefore, the encodings that `TxtConvertEncoding()` supports are dependent upon the ROM's locale. The locale module provides support for Unicode, the device encoding, and a set of related or locale-important encodings. The following tables summarize the set of encodings supported in `TxtConvertEncoding()` by various locales.

Table 8.1 Source encodings for Latin ROMs

charEncodingUCS2	charEncodingUCS4
charEncodingUTF16	charEncodingUTF32
charEncodingUTF16BE	charEncodingUTF32BE
charEncodingUTF16LE	charEncodingUTF32LE
charEncodingUTF8	charEncodingPalmLatin
charEncodingAscii	charEncodingGSM
charEncodingISO8859_1	charEncodingCP1252

Table 8.2 Destination encodings for Latin ROMs

charEncodingUCS2	charEncodingUCS4
charEncodingUTF16	charEncodingUTF32
charEncodingUTF16BE	charEncodingUTF32BE
charEncodingUTF16LE	charEncodingUTF32LE
charEncodingUTF8	charEncodingPalmLatin
charEncodingAscii	charEncodingGSM
charEncodingISO8859_1	charEncodingCP1252

Table 8.3 Source encodings for Shift JIS ROMs

charEncodingUCS2	charEncodingUCS4
charEncodingUTF16	charEncodingUTF32
charEncodingUTF16BE	charEncodingUTF32BE
charEncodingUTF16LE	charEncodingUTF32LE
charEncodingUTF8	charEncodingPalmSJIS
charEncodingAscii	charEncodingISO8859_1
charEncodingCP1252	charEncodingGSM
charEncodingShiftJIS	charEncodingCP932
charEncodingISO2022Jp	

Table 8.4 Destination encodings for Shift JIS ROMs

charEncodingUCS2	charEncodingUCS4
charEncodingUTF16	charEncodingUTF32
charEncodingUTF16BE	charEncodingUTF32BE
charEncodingUTF16LE	charEncodingUTF32LE

Table 8.4 Destination encodings for Shift JIS ROMs *(continued)*

charEncodingUTF8	charEncodingPalmSJIS
charEncodingShiftJIS	charEncodingCP932
charEncodingGSM	charEncodingISO2022Jp
charEncodingISO8859_1	charEncodingCP1252
charEncodingAscii	

Table 8.5 Source encodings for GB ROMs

charEncodingUCS2	charEncodingUTF8
charEncodingUTF16	charEncodingUTF16LE
charEncodingUTF16BE	charEncodingUTF32
charEncodingUTF32BE	charEncodingUTF32LE
charEncodingUCS4	charEncodingPalmGB
charEncodingGB2312	charEncodingGBK
charEncodingISO2022CN	charEncodingBig5
charEncodingBig5_HKSCS	charEncodingAscii
charEncodingCP1252	charEncodingISO8859_1
charEncodingGSM	

Table 8.6 Destination encodings for GB ROMs

charEncodingUCS2	charEncodingUTF8
charEncodingUTF16	charEncodingUTF16LE
charEncodingUTF16BE	charEncodingUTF32
charEncodingUTF32BE	charEncodingUTF32LE
charEncodingUCS4	charEncodingPalmGB

Table 8.6 Destination encodings for GB ROMs *(continued)*

charEncodingGB2312	charEncodingGBK
charEncodingISO2022CN	charEncodingAscii
charEncodingISO8859_1	charEncodingGSM

TxtEncodingName Function

Purpose Obtains a character encoding's name.

Declared In `TextMgr.h`

Prototype

```
const char *TxtEncodingName
    (CharEncodingType iEncoding)
```

Parameters

→ *iEncoding*
One of the `CharEncodingType` values, indicating a character encoding.

Returns A constant string containing the name of the encoding.

Comments Use this function to obtain the official name of the character encoding, suitable to pass to an Internet application or any other application that requires the character encoding's name to be passed along with the data.

See Also `TxtNameToEncoding()`

TxtFindString Function

Purpose Performs a case-insensitive search for a string in another string.

Declared In `TextMgr.h`

Prototype

```
Boolean TxtFindString (const char *iSrcStringP,
    const char *iTargetStringP, size_t *oFoundPos,
    size_t *oFoundLen)
```

Parameters

→ *iSrcStringP*
The string to be searched.

→ *iTargetStringP*
Prepared version of the string to be found. This string should either be passed directly from the `strToFind` field in the

[`sysAppLaunchCmdFind`]() launch code's parameter block or it should be prepared using the function [`TxtPrepFindString()`]().

← *`oFoundPos`*
Pointer to the offset of the match in *`iSrcStringP`*.

← *`oFoundLen`*
Pointer to the length in bytes of the matching text.

Returns `true` if the function finds *`iTargetStringP`* within *`iSrcStringP`*; `false` otherwise.

If found, the values pointed to by the *`oFoundPos`* and *`oFoundLen`* parameters are set to the starting offset and the length of the matching text. If not found, the values pointed to by *`oFoundPos`* and *`oFoundLen`* are set to 0.

The search that `TxtFindString()` performs is locale-dependent. On most ROMs with Latin-based encodings, `TxtFindString()` returns `true` only if the string is at the beginning of a word. On Shift JIS encoded ROMs, `TxtFindString()` returns `true` if the string is located anywhere in the word.

You must make sure that the parameters *`iSrcStringP`* and *`iTargetStringP`* point to the start of a valid character. That is, they must point to the first byte of a multi-byte character, or they must point to a single-byte character; if they don't, results are unpredictable.

See Also [`TxtCaselessCompare()`]()

TxtGetChar Function

Purpose Retrieves the character starting at the specified offset within a text buffer.

Declared In `TextMgr.h`

Prototype
```
wchar32_t TxtGetChar (const char *iTextP,
    size_t iOffset)
```

Parameters → *`iTextP`*
Pointer to the text buffer to be searched.

→ *iOffset*
A valid offset into the buffer *iTextP*. This offset must point to an inter-character boundary.

Returns The character at *iOffset* in *iTextP*.

Comments You must make sure that the parameter *iTextP* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

See Also TxtGetNextChar(), TxtSetNextChar()

TxtGetEncodingFlags Function

Purpose Returns the attributes of a particular character encoding.

Declared In `TextMgr.h`

Prototype

```
uint32_t TxtGetEncodingFlags
    (CharEncodingType iEncoding)
```

Parameters

→ *iEncoding*
A `CharEncodingType` value specifying a character encoding.

Returns An unsigned integer with one or more of the Character Encoding Attributes flags set.

TxtGetNextChar Function

Purpose Retrieves the character starting at the specified offset within a text buffer.

Declared In `TextMgr.h`

Prototype

```
size_t TxtGetNextChar (const char *iTextP,
    size_t iOffset, wchar32_t *oChar)
```

Parameters

→ *iTextP*
Pointer to the text buffer to be searched.

→ *iOffset*
A valid offset into the buffer *iTextP*. This offset must point to an inter-character boundary.

← *oChar*
The character at *iOffset* in *iTextP*. Pass NULL for this parameter if you don't need the character returned.

Returns The size in bytes of the character at *iOffset*. If *oChar* is not NULL upon entry, it points to the character at *iOffset* upon return.

Comments You must make sure that the parameter *iTextP* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

Example You can use this function to iterate through a text buffer character-by-character in this way:

```
size_t i = 0;
wchar32_t ch;
while (i < bufferLength) {
    i += TxtGetNextChar(buffer, i, &ch);
    //do something with ch.
}
```

See Also TxtGetChar(), TxtGetPreviousChar(), TxtSetNextChar()

TxtGetPreviousChar Function

Purpose Retrieves the character before the specified offset within a text buffer.

Declared In TextMgr.h

Prototype
```
size_t TxtGetPreviousChar (const char *iTextP,
    size_t iOffset, wchar32_t *oChar)
```

Parameters

→ *iTextP*
Pointer to the text buffer to be searched.

→ *iOffset*
A valid offset into the buffer *iTextP*. This offset must point to an inter-character boundary.

← *oChar*
The character immediately preceding *iOffset* in *iTextP*. Pass NULL for this parameter if you don't need the character returned.

Returns The size in bytes of the character preceding `iOffset` in `iTextP`. If `oChar` is not `NULL` upon entry, then it points to the character preceding `iOffset` upon return. Returns 0 if `iOffset` is at the start of the buffer (that is, `iOffset` is 0).

Comments This function is often slower to use than `TxtGetNextChar()` because it must determine the appropriate character boundaries if the byte immediately before the offset is valid in more than one location (start, middle, or end) of a multi-byte character. To do this, it must work backwards toward the beginning of the string until it finds an unambiguous byte.

You must make sure that the parameter `iTextP` points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

Example You can use this function to iterate through a text buffer character-by-character in this way:

```
wchar32_t ch;
// Find the start of the character containing the last byte.
TxtCharBounds (buffer, bufferLength - 1, &start, &end);
i = start;
while (i > 0) {
    i -= TxtGetPreviousChar(buffer, i, &ch);
    //do something with ch.
}
```

TxtGetTruncationOffset Function

Purpose Returns the appropriate byte position for truncating a text buffer such that it is at most a specified number of bytes long.

Declared In `TextMgr.h`

Prototype
```
size_t TxtGetTruncationOffset
    (const char *iTextP, size_t iOffset)
```

Parameters

→ *`iTextP`*
    Pointer to a text buffer.

→ *`iOffset`*
    An offset into the buffer `iTextP`.

Returns The appropriate byte offset for truncating *`iTextP`* at a valid inter-character boundary. The return value may be less than or equal to *`iOffset`*.

Comments You must make sure that the parameter *`iTextP`* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

TxtGetWordWrapOffset Function

Purpose Locates an appropriate place for a line break in a text buffer.

Declared In `TextMgr.h`

Prototype
```
size_t TxtGetWordWrapOffset (const char *iTextP,
    size_t iOffset)
```

Parameters

→ *`iTextP`*
Pointer to a text buffer.

→ *`iOffset`*
A valid offset where the search should begin. The search is performed backward starting from this offset.

Returns The offset of a character that can begin on a new line (typically, the beginning of the word that contains *`iOffset`* or last word before *`iOffset`*). If an appropriate break could not be found, returns *`iOffset`*.

Comments The `FntWordWrap()` function calls `TxtGetWordWrapOffset()` to locate an appropriate place to break the text. The returned offset points to the character that should begin the next line.

This function starts at *`iOffset`* and works backward until it finds a character that typically occurs between words (for example, white space or punctuation). Then it moves forward until it locates the character that begins a word (typically, a letter or number). Note that this function may return an offset value that is greater than the one passed in if the offset passed in occurs immediately before white space or in the middle of white space.

TxtMaxEncoding Function

Purpose Returns the higher of two encodings.

Declared In `TextMgr.h`

Prototype
```
CharEncodingType TxtMaxEncoding
    (CharEncodingType a, CharEncodingType b)
```

Parameters
→ *a*
A `CharEncodingType` to compare.

→ *b*
Another `CharEncodingType` to compare.

Returns The higher of *a* or *b*. One character encoding is higher than another if it is more specific. For example code page 1252 is "higher" than ISO 8859-1 because it represents more characters than ISO 8859-1.

Comments This function is used by `TxtStrEncoding()` to determine the encoding required for a string.

See Also `TxtCharEncoding()`, `CharEncodingType`

TxtNameToEncoding Function

Purpose Returns an encoding's constant given its name.

Declared In `TextMgr.h`

Prototype
```
CharEncodingType TxtNameToEncoding
    (const char *iEncodingName)
```

Parameters
→ *iEncodingName*
One of the string constants containing the official name of an encoding. You can find a list of official names at this URL: `http://www.iana.org/assignments/character-sets`.

Returns One of the `CharEncodingType` constants. Returns `charEncodingUnknown` if the specified encoding could not be found.

Comments Use this function to convert a character encoding's name as received from an Internet application into the character encoding constant that some Text Manager functions require.

This function properly converts aliases for a character encoding. For example, passing the strings "us-ascii", "ASCII", "cp367", and "IBM367" all return `charEncodingAscii`.

All locales can access the Text Manager's character set list, which contains the standard set of aliases for the locales that Palm OS supports. Each locale may add its own aliases to the list as well. For example, a device with the Shift JIS encoding might add its own set of aliases, which would be unknown in other locales.

See Also `TxtEncodingName()`

TxtNextCharSize Macro

Purpose Returns the size of the character starting at the specified offset within a text buffer.

Declared In `TextMgr.h`

Prototype `#define TxtNextCharSize (`*`iTextP`*`, `*`iOffset`*`)`

Parameters

→ *`iTextP`*
Pointer to the text buffer to be searched.

→ *`iOffset`*
A valid offset into the buffer *`iTextP`*. This offset must point to an inter-character boundary.

Returns The size in bytes of the character at *`iOffset`*.

Comments You must make sure that the parameter *`iTextP`* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

See Also `TxtGetNextChar()`

TxtParamString Function

Purpose Replaces substrings within a string with the specified values.

Declared In `TextMgr.h`

Prototype
```
char *TxtParamString (const char *inTemplate,
    const char *param0, const char *param1,
    const char *param2, const char *param3)
```

Parameters

→ *`inTemplate`*
: The string containing the substrings to replace.

→ *`param0`*
: String to replace ^0 with or `NULL`.

→ *`param1`*
: String to replace ^1 with or `NULL`.

→ *`param2`*
: String to replace ^2 with or `NULL`.

→ *`param3`*
: String to replace ^3 with or `NULL`.

Returns A pointer to a locked relocatable chunk in the dynamic heap that contains the appropriate substitutions.

Comments This function searches *`inTemplate`* for occurrences of the sequences ^0, ^1, ^2, and ^3. When it finds these, it replaces them with the corresponding string passed to this function. Multiple instances of each sequence will be replaced.

The replacement strings can also contain the substitution strings, provided they refer to a later parameter. That is, the *`param0`* string can have references to ^1, ^2, and ^3, the *`param1`* string can have references to ^2 and ^3, and the *`param2`* string can have references to ^3. Any other occurrences of the substitution strings in the replacement strings are ignored. For example, if *`param3`* is the string “^0”, any occurrences of ^3 in *`inTemplate`* are replaced with the string “^0”.

You must make sure that the parameter *`inTemplate`* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn’t, results are unpredictable.

`TxtParamString()` allocates space for the returned string in the dynamic heap through a call to `MemHandleNew()`, and then returns the result of calling `MemHandleLock()` with this handle. Your code is responsible for freeing this memory when it is no longer needed.

See Also `TxtReplaceStr()`, `FrmCustomAlert()`

TxtPrepFindString Function

Purpose Prepares a string for use in `TxtFindString()`.

Declared In `TextMgr.h`

Prototype

```
size_t TxtPrepFindString (const char *iSrcTextP,
   size_t iSrcLen, char *oDstTextP,
   size_t iDstSize)
```

Parameters

→ *iSrcTextP*
The text to be searched for. Must not be `NULL`.

→ *iSrcLen*
The number of bytes of *iSrcTextP* to convert.

← *oDstTextP*
The same text as in *iSrcTextP* but converted to a suitable format for searching. *oDstTextP* must not be the same address as *iSrcTextP*.

→ *iDstSize*
The length in bytes of the area pointed to by *oDstTextP*.

Returns The number of bytes from *iSrcTextP* that were converted.

Comments Use this function to normalize the string to search for before using `TxtFindString()` to perform a search that is internal to your application. If you are using `TxtFindString()` in response to the `sysAppLaunchCmdFind` launch code, the string that the launch code passes in is already properly normalized for the search.

This function normalizes the string to be searched for. The method by which a search string is normalized varies depending on the version of Palm OS and the character encoding supported by the device.

If necessary to prevent overflow of the destination buffer, not all of *iSrcTextP* is converted.

You must make sure that the parameter *iSrcTextP* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character. If it doesn't, results are unpredictable.

TxtPreviousCharSize Macro

Purpose Returns the size of the character before the specified offset within a text buffer.

Declared In `TextMgr.h`

Prototype `#define TxtPreviousCharSize (iTextP, iOffset)`

Parameters

→ *iTextP*
Pointer to the text buffer.

→ *iOffset*
A valid offset into the buffer *iTextP*. This offset must point to an inter-character boundary.

Returns The size in bytes of the character preceding *iOffset* in *iTextP*. Returns 0 if *iOffset* is at the start of the buffer (that is, *iOffset* is 0).

Comments You must make sure that the parameter *iTextP* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

This macro is often slower to use than `TxtNextCharSize()` because it must determine the appropriate character boundaries if the byte immediately before the offset is valid in more than one location (start, middle, or end) of a multi-byte character. To do this, it must work backwards toward the beginning of the string until it finds an unambiguous byte.

See Also `TxtGetPreviousChar()`

TxtReplaceStr Function

Purpose Replaces a substring of a given format with another string.

Declared In `TextMgr.h`

Prototype
```
uint16_t TxtReplaceStr (char *iStringP,
    size_t iMaxLen, const char *iParamStringP,
    uint16_t iParamNum)
```

Parameters

↔ *iStringP*
The string in which to perform the replacing.

→ *iMaxLen*
The maximum length in bytes that *iStringP* can become.

→ *iParamStringP*
The string that ^*iParamNum* should be replaced with. If `NULL`, no changes are made.

→ *iParamNum*
A single-digit number (0 to 9).

Returns The number of occurrences found and replaced.

Raises a fatal error message if *iParamNum* is greater than 9.

Comments This function searches *iStringP* for occurrences of the string ^*iParamNum*, where *iParamNum* is any digit from 0 to 9. When it finds the string, it replaces it with *iParamStringP*. Multiple instances are replaced as long as the resulting string doesn't contain more than *iMaxLen* bytes, not counting the terminating null.

You can set the *iParamStringP* parameter to `NULL` to determine the required length of *iStringP* before actually doing the replacing. **`TxtReplaceStr()`** returns the number of occurrences it finds of ^*iParamNum*. Multiply this value by the length of the *iParamStr* you intend to use to determine the appropriate length of *iStringP*.

You must make sure that the parameter *iStringP* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

TxtSetNextChar Function

Purpose Sets a character within a text buffer.

Declared In `TextMgr.h`

Prototype
```
size_t TxtSetNextChar (char *iTextP,
    size_t iOffset, wchar32_t iChar)
```

Parameters

↔ *`iTextP`*
Pointer to a text buffer.

→ *`iOffset`*
A valid offset into the buffer *`iTextP`*. This offset must point to an inter-character boundary.

→ *`iChar`*
The character to replace the character at *`iOffset`* with. Must not be a virtual character.

Returns The size of *`iChar`*.

Comments This function replaces the character in *`iTextP`* at the location *`iOffset`* with the character *`iChar`*. Note that there must be enough space at *`iOffset`* to write the character.

You can use `TxtCharSize()` to determine the size of *`iChar`*.

You must make sure that the parameter *`iTextP`* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

See Also `TxtGetNextChar()`

TxtStrEncoding Function

Purpose Returns the encoding required to represent a string.

Declared In `TextMgr.h`

Prototype
```
CharEncodingType TxtStrEncoding
    (const char *iStringP)
```

Parameters

→ *`iStringP`*
A string.

Returns A `CharEncodingType` value that indicates the encoding required to represent *`iStringP`*. If any character in the string isn't recognizable, then `charEncodingUnknown` is returned.

Comments The encoding for the string is the maximum encoding of any character in that string. For example, if a two-character string contains a blank space and a ü, the appropriate encoding is `charEncodingISO8859_1`. The blank space's minimum encoding is ASCII. The minimum encoding for the ü is ISO 8859-1. The maximum of these two encodings is ISO 8859-1.

Use this function for informational purposes only. Your code should not assume that the character encoding returned by this function is the Palm OS system's character encoding. (Instead use `LmGetSystemLocale()`.)

See Also `TxtCharEncoding()`, `TxtMaxEncoding()`

TxtTransliterate Function

Purpose Converts the specified number of bytes in a text buffer using the specified operation.

Declared In `TextMgr.h`

Prototype

```
status_t TxtTransliterate (const char *iSrcTextP,
    size_t iSrcLength, char *oDstTextP,
    size_t *ioDstLength,
    TranslitOpType iTranslitOp)
```

Parameters

→ *`iSrcTextP`*
Pointer to a text buffer.

→ *`iSrcLength`*
The length in bytes of *`iSrcTextP`*.

← *`oDstTextP`*
The output buffer containing the converted characters.

↔ *`ioDstLength`*
Upon entry, the maximum length of *`oDstTextP`*. Upon return, the actual length of *`oDstTextP`*.

→ `iTranslitOp`

A 16-bit unsigned value that specifies which transliteration operation is to be performed. See `TranslitOpType` for the possible values for this field.

You can ensure that you have enough space for the output by OR-ing your chosen operation with `translitOpPreprocess`.

Returns One of the following values:

`errNone`

Success

`txtErrUknownTranslitOp`

`iTranslitOp`'s value is not recognized

`txtErrTranslitOverrun`

`iSrcTextP` and *`oDstTextP`* point to the same memory location and the operation would cause the function to overwrite unprocessed data in the input buffer.

`txtErrTranslitOverflow`

`oDstTextP` is not large enough to contain the converted string.

`txtErrTranslitUnderflow`

The end of the source buffer contains a partial character.

Comments *`iSrcTextP`* and *`oDstTextP`* may point to the same location if you want to perform the operation in place. However, you should be careful that the space required for *`oDstTextP`* is not larger than *`iSrcTextP`* so that you don't generate a `txtErrTranslitOverrun` error.

For example, suppose on a Shift JIS encoded system, you want to convert a series of single-byte Japanese Katakana symbols to double-byte Katakana symbols. You cannot perform this operation in place because it replaces a single-byte character with a multi-byte character. When the first converted character is written to the buffer, it overwrites the second input character. Thus, a text overrun has occurred.

You must make sure that the parameter *`iSrcTextP`* points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

Example The following code shows how to convert a string to uppercase.

```
outSize = buf2Len;
error = TxtTransliterate(buf1, buf1len, &buf2, &outSize,
   translitOpUpperCase|translitOpPreprocess);
if (outSize > buf2len)
    /* allocate more memory for buf2 */
error = TxtTransliterate(buf1, buf1Len, &buf2, &outSize,
   translitOpUpperCase);
```

TxtTruncateString Function

Purpose Determines if a string fits within a given number of bytes. If not, truncates the string.

Declared In `TextMgr.h`

Prototype
```
Boolean TxtTruncateString (char *iDstString,
    const char *iSrcString, size_t iMaxLength,
    Boolean iAddEllipsis)
```

Parameters

← *`iDstString`*
The null-terminated string truncated if necessary so that it is no more than *`iMaxLength`* bytes long.

→ *`iSrcString`*
A null-terminated string.

→ *`iMaxLength`*
The maximum length of *`iDstString`* including the null terminator.

→ *`iAddEllipsis`*
If `true`, an ellipsis character is the last character of *`iDstString`* if *`iSrcString`* had to be truncated. If `false`, *`iSrcString`* is truncated at the last character that fits in *`iDstString`*.

Returns `true` if the string was truncated, or `false` if the string can fit without truncation.

Comments This function determines whether *`iSrcString`* can be copied into a string with the specified length without being truncated. If it can, `TxtTruncateString()` returns `false` and copies *`iSrcString`* into *`iDstString`*. If the string must be truncated, this function copies one less than the number of characters that can fit in *`iMaxLength`* into *`iDstString`* and then appends an ellipsis (...) character.

See Also `FntWidthToOffset()`, `WinDrawTruncChars()`, `TxtGetTruncationOffset()`

TxtWordBounds Function

Purpose Finds the boundaries of a word of text that contains the character starting at the specified offset.

Declared In `TextMgr.h`

Prototype

```
Boolean TxtWordBounds (const char *iTextP,
    size_t iLength, size_t iOffset,
    size_t *oWordStart, size_t *oWordEnd)
```

Parameters

→ *`iTextP`*
Pointer to a text buffer.

→ *`iLength`*
The length in bytes of the text pointed to by *`iTextP`*.

→ *`iOffset`*
A valid offset into the text buffer *`iTextP`*. This offset must point to the beginning of a character.

← *`oWordStart`*
The starting offset of the text word.

← *`oWordEnd`*
The ending offset of the text word.

Returns `true` if a word is found. Returns `false` if the word doesn't exist or is punctuation or whitespace.

Comments Assuming the ASCII encoding, if the text buffer contains the string "Hi! How are you?" and you pass 5 as the offset, `TxtWordBounds()` returns the start and end of the word containing the character at offset 5, which is the character "o". Thus, *`oWordStart`* and *`oWordEnd`* would point to the start and end of the word "How".

You must make sure that the parameter `iTextP` points to the start of a valid character. That is, it must point to the first byte of a multi-byte character or it must point to a single-byte character; if it doesn't, results are unpredictable.

See Also `TxtCharBounds()`, `TxtCharIsDelim()`, `TxtGetWordWrapOffset()`

Part III
Appendixes

This part contains supplementary localization material. It covers:

A

Language-specific Information

This appendix contains information about language-specific implementations of Palm OS®. Read it if you are localizing to these languages to determine the correct programming practices for these languages.

Notes on the Japanese Implementation

This section describes programming practices for applications that are to be localized for Japanese use. It covers:

Japanese Character Encoding

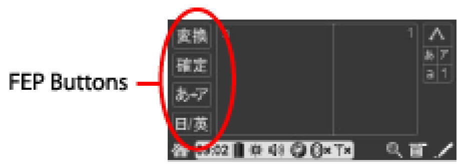
The character encoding used on Japanese systems is based on Microsoft code page 932. The complete 932 character set (JIS level 1 and 2) is supported in both the standard and large font sizes. The bold versions of these two fonts contain bolded versions of the glyphs found in the 7-bit ASCII range, but on some devices, the single-byte Katakana characters and the multi-byte characters are not bolded.

Japanese Character Input

On Japanese devices, users can enter Japanese text using Latin (ASCII) characters, and special software called a front-end processor

(FEP) transliterates this text into Hiragana or Katakana characters. The user can then ask the FEP to phonetically convert Hiragana characters into a mixture of Hiragana and Kanji (Kana-Kanji conversion).

Figure 8.1 Handwriting recognition pinlet on a Japanese device



The Graffiti® 2 handwriting recognition pinlet for Japanese ROMs has four buttons that control the FEP transliteration and conversion process. These four FEP buttons are arranged vertically to the left of the handwriting recognition area. The top-most FEP button tells the FEP to attempt Kana-Kanji conversion on the inline text. The next button confirms the conversion results and removes the first clause from the inline text. The third button toggles the transliteration mode between Hiragana and Katakana. The last button toggles the FEP on and off.

Japanese text entry is always inline, which means that transliteration and conversion happen directly inside of a field. The field code passes events to the FEP, which then returns information about the appropriate text to display.

During inline conversion, the Graffiti 2 space stroke acts as a shortcut for the conversion FEP button and the return stroke acts as a shortcut for the confirm FEP button.

Displaying Japanese Strings on UI Objects

To conserve screen space, you should use half-width Katakana characters on user interface elements (such as buttons, menu items, labels, and pop-up lists) whenever the string contains only Katakana characters. If the string contains a mix of Katakana and either Hiragana, Kanji, or Romaji, then use the full-width Katakana characters instead.

Displaying Error Messages

You may have code that uses the macros `DbgOnlyFatalErrorIf()` and `DbgOnlyFatalError()` to determine error conditions. If the error condition occurs, the system displays the file name and line number at which the error occurred along with the message that you passed to the macro. Often these messages are hard-coded strings. On Japanese systems, Palm OS traps the messages passed to these two macros and displays a generic message explaining that an error has occurred.

You should only use `DbgOnlyFatalErrorIf()` and `DbgOnlyFatalError()` for totally unexpected errors. Do not use them for errors that you believe your end users will see. If you wish to inform your users of an error, use a localizable resource to display the error message instead of `DbgOnlyFatalErrorIf()` or `DbgOnlyFatalError()`.

Index

M

N

O

P

U

V

W