Making Your Application Run Well in a Multiscript Environment

A few techniques that will give your application polish on International systems

Nat McCully, Senior Software Engineer, Apple Computer, Inc.

Introduction

Many applications are sold in the U.S. and Europe without any major changes to their codebases for a specific country or region. Sometimes this means that the non-U.S. user runs into oddities of design or implementation that aren't quite right for his or her language or region, because the code assumes a U.S.-centric design. Using "," for the thousands separator in a number field, or "/" for a short date separator in a date are two examples. These defects are not that serious, and in fact are avoidable if the program uses the MacOS International Utilities functions to extract region-specific data from resources in the System, like the correct thousands separator or date separator.

Once the product is to be distributed in a region which uses a different script system from that in the U.S., things can get a bit more complicated. The U.S. uses the Roman script system to display text and other data, as does most of Europe. Japan, however, uses the Japanese script system to display text and other data, and therefore some products that assume Roman script behavior will not function properly, or worse, will not function at all on a Japanese system.

This paper will illustrate some techniques you can use in your application so it will run properly in a multi-script environment. These techniques allow your code to be easily localizable into any language in any script system, which will increase your possible user base and therefore your product's revenue potential.

What is 'Mojibake?'

Mojibake ("moh-jee-bah-keh") is a Japanese word for when a run of text is displayed in the wrong script system, and produces garbage characters that don't make any sense. An example is below:

Text in correct (Japanese) font	Text in wrong (Roman) font
文字化け	ïðéöâªÇØ

Figure 1: *Mojibake Example*

This problem is one of the most common with applications that support multiple fonts. What has happened is that each byte in the text stream above has not changed, but the font used to render it has. This problem is a side-effect of the way in which the MacOS supports so many languages, by grouping languages and their fonts into script systems. A run of the same raw text data will effectively change its meaning (or lose it completely) depending on which font is used to display it.

This means that unless the application makes an effort to protect the user from mojibake, it will likely happen, and this is a bad thing. The user may think that their data has become corrupted, and may panic, telling all his or her friends how buggy your software is.

It turns out that protecting the user from mojibake is not such a big deal. It can be easily defined and scoped so you will

always know what the 'right' thing is to do when you are handling text in multiple fonts in a multiscript environment. For example, you only need to worry about mojibake when:

- The user's system has more than one script system installed
- There are characters in the Extended ASCII (hi-ASCII) range (> ASCII 127) in the text

Further, there are four possible situations in which mojibake can occur:

A. When the user chooses a keyscript different from the script of the current font and begins to type hi-ASCII characters.

B. When the user selects text and chooses a font from a different script than the text's current font AND there are hi-ASCII characters in the selection.

C. When there is hi-ASCII in the text of the user interface of your application and you default to drawing it in a font that can change (i.e. the appFond) depending on the main script of the system.

D. When you are importing text without font information and must set the font yourself to some default, e.g. plain text import, or opening a document with fonts not installed in the current system AND there is hi-ASCII in the text stream being imported.

Notice that in all four of those cases there is a common attribute: Hi-ASCII text. All fonts in all scripts (except for special fonts in the Roman script, like Symbol) share the same lo-ASCII characters, but have different hi-ASCII characters depending on the script of the font. There are some exceptions to this when a country has a standard different from the ASCII standard for character codes 0-127, but for most purposes you as the application developer can assume that if the character code is <=127 and it is not part of a multi-byte character, it will display the same glyph no matter what font you are in. This is a powerful piece of information, because it will guide how you implement several of the features mentioned in this paper.

Font Input Locking

This feature protects the user from mojibake as they are inputting text, as in situation A, above. The desired behavior in that case is if the user is typing in a keyscript that is different from the script of the insertion font, then you must temporarily "lock" the font to a font from the same script as the keyscript. This font can be a user preference, or you can simply use the appFond or sysFond for that script.

Most applications that have word-processing functionality buffer the text as it is typed in, to maximize typing efficiency. This means you have a tight loop that repeatedly gets keyDown events off the event queue and stores the resulting text in a buffer. When the user is using an Input Method program to type in Japanese, once they hit the return key to send to text to your application, you will get a whole stream of keyDown events in rapid succession, each keyDown representing a byte in the stream of 2-byte characters. Once you buffer the characters, you would insert the text from that buffer into the document as a chunk into the current stylerun, which has a font attribute. It is the font attribute that concerns us here. Sometimes the text being inserted will be incompatible with the current insertion font, and you will have to create a new stylerun with a font that will render the text properly.

Session-based Font Input Locking

Session-based Font Input Locking is appropriate when the user is typing in a non-Roman keyscript, where all the characters will be hi-ASCII, or multibyte characters which mix hi-ASCII and lo-ASCII together. In that case, all the

incoming characters will need to be "locked" to a font from the same script as the keyscript. As soon as the user switches to a different keyscript, you can restore the old insertion font.

Character-based Font Input Locking

Sometimes you only want to "lock" the font of some of the characters in the buffer, leaving the others in the insertion font. This is the case when the user is typing in the Roman keyscript, but the insertion font is non-Roman. Most of the characters will be lo-ASCII, but the user can also hold the option key down and type a single hi-ASCII character, which will need to be "font locked" to a Roman font to display the proper glyph. In this case, only that hi-ASCII character needs to be in its own stylerun, and you can restore the original insertion font when the user types the next lo-ASCII character. (See Figure 2). Another case where Character-based Font Input Locking is appropriate is when the user is typing in the Japanese keyscript in a Roman font run, but using the single-byte input mode of their Input Method, resulting in lo-ASCII characters that do not need to be font locked. However, as soon as the user switches to a mode which allows input of single-byte hi-ASCII or 2-byte characters, you will need to lock those characters to a Japanese font. At all times during this input session, the keyscript will be Japanese, so you cannot assume all text input in Japanese keyscript needs font locking when in a Roman font.

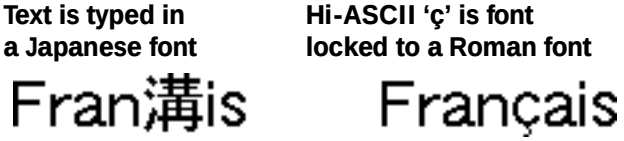

Figure 2: *Character-based Font Input Locking*

When organizing your code to support font input locking, it helps to break down the problem into separate parts of filtering for hi-ASCII (either in a text buffer or as keyDown events), and creating new styleruns for text that needs to be font locked. The following code (Listing 1 and Listing 2) are simple examples of how to do this.

Listing 1: *Finding Hi-ASCII Characters in a Buffer*

BufferHasHiASCII
Loop through a buffer and stop when you find a hi-ASCII character.

```
Boolean BufferHasHiASCII(uchar * buffer,
long length)
{
  long index;

  for (index = 0; index < length; index++)
  {
    if (buffer[index] & 0x80)
      return TRUE;
  }
  return FALSE;
}
```

Listing 2: *StyleRun Management Sample*

FindStyleRunIndex
Each StyleRun structure has a starting character position (startPos) and a font attribute. You would also have size, face, etc. in a real application.

```
typedef struct tagStyleRun {
  long  startPos;
  short font;
} StyleRun;

// Given an ordered array of StyleRun structures,
// find the one that
// corresponds to the given charPos.
long FindStyleRunIndex(StyleRun ** styles,
  long numStyles, long charPos)
{
  longindex;

  for (index = 0; index < numStyles;
index++)
  {
    if ((*styles)[index].startPos <=
charPos)
      break;
  }
  return index;
}
```

ExtendStyleRun
When extending the length of the current stylerun, all styleruns following it must have their startPos member incremented so they match up with the text.

```
void ExtendStyleRun(StyleRun ** styles,
  long numStyles, long index, long amount)
{
  ++index;// the next style has the start
          // we need to increment
  while (index < numStyles)
    (*styles)[index++].startPos += amount;
}
```

AddNewStyleRun
Adding styleruns to your text is at the core of font locking functionality. All styleruns are stored in a Handle, and are in order according to text order. You must bump the size of the Handle and then insert your new style in at the appropriate place.

```
Boolean AddNewStyleRun(StyleRun ** styles,
  long * numStyles, long index, * newStyle,
  long runLength)
{
  // First try to increase the stylerun array handle
  if (SetHandleSize((Handle)styles,
    GetHandleSize((Handle)styles) +
    sizeof(StyleRun))
  {
    // Then make room for the new stylerun in the array
    BlockMove((Ptr)&(*styles)[index],
      (Ptr)&(*styles)[index + 1],
      sizeof(StyleRun) *
      (*numStyles - index));

    // Then copy the new stylerun into the array
    BlockMove((Ptr)newStyle,
      (Ptr)&(*styles)[index],
      sizeof(StyleRun));

    // increment size of array handle and index
    (*numStyles)++;

    // Increment start positions of each run after
    // the new run
    ExtendStyleRun(styles, *numStyles,
      index, runLength);

    return TRUE;
  }
  else
    return FALSE;
}
```

Now let's put these things to use. Say you have a function that creates a buffer of text as the user is typing, and when the user is done, you process it for font locking, add it to the document's data, and then draw it onscreen. Note that much of this code is over-simplified to illustrate the concepts in this paper; you should improve upon it before using it in a real application.

Listing 3 shows how to make a keyDown loop that gets keys until the user stops typing rapidly or they change the keyscript:

Listing 3: *Fast Key Loop to Make Key Event Buffer*

DoKey
Loop until no more keyDown events or until keyscript changes.

```
void DoKey(Str255 buffer,
  ScriptCode * keyScript)
{
  EventRecord  theEvent;
  uchar     char;

  *keyScript =
    GetScriptManagerVariable(smKeyScript);

  while (
    EventAvail(keyDownMask, &theEvent))
  {
    if (buffer[0] < 255 &&
      *keyScript ==
     GetScriptManagerVariable(smKeyScript))
    {
      (void)GetNextEvent(keyDownMask,
        &theEvent);
      char = (uchar)(theEvent.message
        & charCodeMask);
      buffer[buffer[0]++] = char;
    }
    else
      break;
  }
}
```

Listing 4 shows how to take the buffer and keyscript information from **Listing 3**, and perform Font Input Locking before adding the typed text to your document:

Listing 4: *Font Input Locking*

ProcessKeyDownWithFontLocking
Actually perform Font Input Locking using the functions introduced earlier.

```
StyleRun **  gStyles;      // styleruns in
                           // text
long         gNumStyles;
uchar **     gText;        // text data
long         gTextLength;
long         gCurCharPos;  // where we are
                           // in the text
```

```
Boolean ProcessKeyDownWithFontLocking
  (Str255 buffer, ScriptCode keyScript)
{
  Boolean     fontLocked = FALSE;
  ScriptCode  fontScript;
  long        curStyleIndex =
      GetStyleRunIndex(gStyles,
        gNumStyles, gCurCharPos);
  StyleRun *  curStylePtr;
  StyleRun    newStyle;
  short       curFont, goodFont;
  long        lastRunLength = buffer[0];
  long        index, subRunStart,
                subRunLength = 0L;

  // Add the text buffer to our text data handle...
  if (SetHandleSize((Handle)gText,
    GetHandleSize((Handle)gText) +
    (long)buffer[0]))
  {
    BlockMove((Ptr)&(*gText)[gCurCharPos],
      (Ptr)&(*gText)[gCurCharPos +
      buffer[0]],
      gTextLength - gCurCharPos);
    BlockMove((Ptr)&buffer[1],
      (Ptr)&(*gText)[gCurCharPos],
      (long)buffer[0]);
    gTextLength += buffer[0];
  }
  else
    return FALSE;

  // If more than one script system is installed
  // and we have hi-ASCII
  // we need to make new styleruns for each 'locked' run
  if (GetScriptManagerVariable(smEnabled) >
    1 && BufferHasHiASCII(
    (uchar *)&buffer[1], (long)buffer[0]))
  {
    curStylePtr =
      &((*gStyles)[curStyleIndex]);
    curFont = curStylePtr->font;
    fontScript = FontToScript(curFont);

    // make styleruns for hi-ASCII and 2-byte chars in run
    if (keyScript != fontScript)
    {
      BlockMove((Ptr)curStylePtr,
        (Ptr)&newStyle, sizeof(StyleRun));
      goodFont = GetScriptVariable(
        keyScript, smScriptAppFond);

      // loop thru sub-runs in buffer
      for (index = 1;
        index <= (long)buffer[0]; index++)
      {
        // find first sub-run that needs font locking
        subRunStart = index;
        while (index <= (long)buffer[0] &&
              !(buffer[index] & 0x80))
          index++;

        // if lo-ASCII starts buffer, extend current
        // stylerun by amount of lo-ASCII characters
        if (!(buffer[subRunStart] & 0x80))
          ExtendStyleRun(gStyles,
            gNumStyles, subRunStart);
        curStyleIndex++;

        newStyle.startPos += index;
        newStyle.font = goodFont;

        // Find length of run that needs locking
        while (((buffer[index] & 0x80) ||
          CharacterByteType(
          (Ptr)&buffer[1], index,
          keyScript) != smSingleByte)) &&
          index <= buffer[0])
        {
          index++;
          subRunLength++;
        }

        AddNewStyle(gStyles, &gNumStyles,
          curStyleIndex, &newStyle,
          subRunLength);
        lastRunLength = subRunLength;
        subRunLength = 0;
      }
      fontLocked = TRUE;
    }
  }

  // Increment all styleruns' indices after the current one.
  ExtendStyleRun(gStyles, gNumStyles,
    curStyleIndex, lastRunLength);

  return fontLocked;
}
```

Font Change Locking

Font Change Locking is the term for when the user chooses a new font for a selected run of text, but some of the text in the run is hi-ASCII and in a font from a different script than the chosen font, so that text is not changed to the new font. This remedies mojibake in situation B, above.

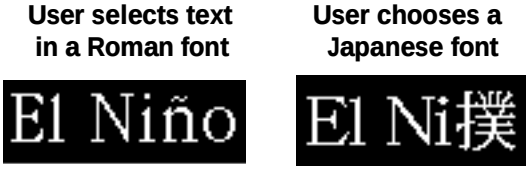

Figure 3: *Mojibake Without Font Change Locking*

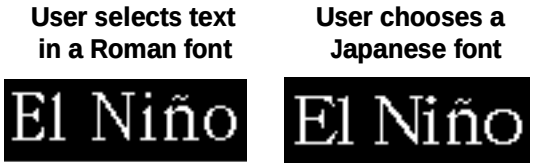

Figure 4: *Font Change Locking 'locks' the 'ñ' to a Roman font*

The basic Font Change Locking algorithm is:

- Check if font(s) in selection are of a different script than the chosen font.
- If so, check if there are any hi-ASCII or multibyte characters in the selection.
- If there are, create new styleruns for each run of "locked" characters.
- Change the font of all characters that are not "locked" to the chosen font.

Forcing the Font Change

Always allow the user to turn Font Change Locking off and force the font to be whatever they set it to, perhaps by holding down a modifier key when they choose a font. This is useful when you import a large amount of raw text data and can't be sure which script system is appropriate to display the text in. You may guess wrong, in which case the user will have to force-set the font to be the proper one.

Font Menu – Keyboard Synchronization

This feature prevents mojibake by automatically setting the keyscript to a script compatible with the font the user selects, when there is no text selection. Here is the basic algorithm when the user chooses a new font from the Font menu:

- Check if there is no text selection (insertion point is flashing).
- If so, check if the current keyscript is non-Roman.
- If so, check if the script of the font chosen is different from the keyscript.
- If so, change the keyscript to that of the chosen font.

This is not the same as changing the current keyscript based on where the user clicks in multiscript text, sometimes called "Font–Keyboard Synchronization." That is not recommended to be a default behavior, and if you implement a feature like that, you should make it a user preference.

Sorting the Font Menu

You should sort all fonts in your font menu according to their script, but how you order the scripts is up to you. There is one thing that you should consider:

Sort the script of the application's user interface translation first, i.e. if the UI is in English or French or some other Roman script language, then the Roman script fonts should be at the top of the Font menu. If the UI is in Japanese, then the Japanese fonts should be at the top. This is because the user is most likely to use the language of the UI localization in their documents, and they need those fonts to be the most accessible, i.e. users of French ClarisWorks will want fonts that can be used in French text at the top; users of Japanese ClarisWorks want the Japanese fonts at the top. Note that the UI localization script can be different from the System script, which is always sorted first by the system. After placing the localization script's fonts first, you can use the MacOS API ScriptOrder() to figure out which scripts should come before which others.

Support of Underline in 2-byte Fonts

QuickDraw on the current version (8.1) of MacOS does not support underline on fonts from the Japanese, Simplified Chinese, Traditional Chinese, or Korean scripts. This is because QuickDraw draws its underline at the baseline of the glyph, according to the measurements in the font's fontMetrics record. The baseline on 2-byte glyphs in fonts from the unsupported

scripts falls on top of part of the character, so instead of drawing the underline lower, QuickDraw bails on drawing the underline at all.

In **Figure 5**, two characters are drawn in a Japanese font (Heisei Mincho 72 pt) and a comparative character is drawn next to them in a Roman font (Palatino 72 pt). The 'M' in Palatino font has QuickDraw underline applied, and that line is extended backwards underneath the two Heisei Mincho font characters. As you can see, the QuickDraw underline would have obscured part of the 2-byte character. The second underline has been drawn further down in the descent area of the font, and is consistent with underline placement of other Japanese applications. That is the line your application would draw.

Figure 5: *Comparing 2-byte Baseline to "Correct" Underline Position*

In some fonts, like Osaka (the appFond and sysFond of the Japanese version of MacOS), the underline may appear to be a little far from the characters if placed in the middle of the descent area, but most Japanese fonts do not have so large a descent value as Osaka, so the line ends up much closer to the text for a more natural look. Of course, the best solution would be for Japanese and other 2-byte fonts to have a special adornment table that told the application where best to draw the underline. Using the baseline for both underline location and placement of 1-byte Roman glyphs does not work in a 2-byte font.

Here is some sample code that draws the underline 50% into the descent area of the font if QuickDraw will not draw the underline for you. It assumes the port has already been set up with the correct font and pen size for drawing underlines.

Listing 5: *Drawing Your Own Underline*

DrawUnderlinedText

```
void DrawUnderlinedText(
  Ptr textPtr, short offset, short length)
{
  Point     pt1, pt2;
  FMetricRec fontMetrics;
  ScriptCode script = FontScript();

  TextFace(underline); // for fonts that
                       // are supported
  GetPen(&pt1);
  DrawText(textPtr, offset, length);
  GetPen(&pt2);

  if (script == smJapanese ||
    script == smTradChinese ||
    script == smSimpChinese ||
    script == smKorean)
  {
    FontMetrics(&fontMetrics);
    MoveTo(pt1.h, pt1.v +
      fontMetrics.descent >> 17);
    Line(pt2.h - pt1.h, 0);
    MoveTo(pt2.h, pt2.v); // restore pen
                          // location
  }
}
```

In a future version of the system software, it has been said that QuickDraw will finally support drawing underline on Japanese and other 2-byte fonts as a user preference.

Truncating Strings

String truncation has been made very simple with the new MacOS APIs TruncString() and TruncText(). These functions are used for strings or text that appear in your user interface where the main reason for truncation is that the string may not fit in the given pixel width. However, sometimes you don't care about the pixel width of the text, but you want to make sure you don't cut text off in the middle of a 2-byte character boundary.

The function below takes a string and a desired truncation length and a script, and truncates the string on the nearest character

boundary less than or equal to the truncation length.

Listing 6: *Multibyte-safe String Truncation*

SmartTruncateString

```
void SmartTruncateString(Str255 string,
  uchar truncLen, ScriptCode script)
{
  if (CharacterByteType((Ptr)&string[1],
    (short)truncLen, script)
    == smFirstByte)
  {
    string[0] = truncLen - 1;
  }
  else
    string[0] = truncLen;
}
```

Searching WorldScript Text

Most search algorithms do a straight byte-compare of text and do not look at script information. This will not produce correct results for the same reason that mojibake occurs: Text will change meaning depending on what font (script) it is displayed in. Therefore, you should put an additional check on a found text run to make sure it is the same meaning (script) as the search string. You can use the International Text Utilities functions CompareString(), CompareText(), etc., but the behavior of these functions may not always be what you want (for example, sometimes diacritical marks are ignored, making the strings “Rosé” and “rose” return equality), and they require the use of a ‘itl2’ sorting resource to do most of the work.

In the following function, you pass a search string (from a Find dialog, for example) and script information for that string, and a text buffer with style information (from the body of your document, for example). The function does a byte compare first, then if the style information and script information are not NULL, it takes them into account and returns TRUE if the search string has been found, and returns the offset into the text buffer at which the string was found in a parameter.

Listing 6: *WorldScript-savvy Byte Compare*

SmartByteCompare

```
Boolean SmartByteCompare(Str255 searchStr,
  ScriptCode script, uchar * textBuffer,
  long length, StyleRun ** styles,
  long numStyles, long * foundOffset)
{
  Boolean  found = FALSE;
  uchar    index1;
  long     index2;

  *foundOffset = -1L;

  for (index2 = 0; index2 < length;
    index2++)
  {
    for (index1 = 1;
      index1 <= searchStr[0]; index1++)
    {
      if (searchStr[index1] !=
          textBuffer[index2])
        break;
      else
        index2++;
    }
    if (index1 == searchStr[0])
    {
      found = TRUE;
      break;
    }
  }

  // Now check the script of the stylerun of the found text.
  // In real life, you'd also check if the found text crossed
  // multiple styleruns and check each of them for the script.
  // Also, you would continue searching if the first found
  // result
  // turned out not to be the same script as the search string.
  if (found == TRUE && styles != NULL)
  {
    long     styleIndex;
    short      font;

    styleIndex = FindStyleRunIndex(
      styles, numStyles, index2 - index1);
    font = (*styles)[styleIndex].font;
    if (script != FontToScript(font))
      found = FALSE;
  }
  return found;
}
```

Conclusion

We have examined a few ways you can make your application run better in multiscript systems right out of the box, with no additional localization. Protecting the user from garbage characters, or mojibake as it's known in Japanese, is something every application should try to do, even if it is only being sold in a particular locale. Every technique explained in this paper revolves around making your code become script-aware, that is, always to pass script information along with any text you may be manipulating internally in your application. If text has a font, it has a script. When there are multiple script systems installed on the user's system, you want your application to behave just as gracefully as when there is only a single script installed.

Bibliography and Related Reading

McCully, Nat. "Supporting Multi-byte Text In Your Application," MacTech Magazine Vol. 14, No. 1. Westlake Village, CA: Xplain Corporation, January, 1998.

Apple Computer, Inc. Inside Macintosh: Text, Menlo Park, CA: Addison Wesley, March 1993.

Apple Computer, Inc. "Technote OV 20, Internationalization Checklist," Cupertino, CA: Apple Computer, Inc, November 1993.

Lunde, Ken. Understanding Japanese Information Processing, Sebastopol, CA: O'Reilly & Associates, September, 1993.

See also Ken Lunde's home page at <http://www.ora.com/people/authors/lunde/>. It has more information about multi-byte text processing on computers.

About the author...

Prior to joining Apple, Nat McCully was at Claris in the Japanese Development Group for six and a half years. He has worked on numerous Japanese products, including MacWrite II–J, Filemaker Pro–J, Claris Impact–J, ClarisDraw–J, and ClarisWorks–J. He speaks, reads and writes Japanese, and enjoys traveling in Japan. He is currently working on the next release of AppleWorks for Macintosh and Windows computers. In his free time he rescues orphaned kittens from hi-tech companies.